VectSharp

2.4.0

Generated by Doxygen 1.9.5

1 VectSharp: a light library for C# vector graphics 1
1.1 Introduction . . . 1
1.2 Installing VectSharp . . . 2
1.3 Usage . . . 2
1.4 Font libraries . . . 3
1.5 Creating new output layers . . . 4
1.6 Compiling VectSharp from source . . . 4
1.6.1 Windows . . . 5
1.6.2 macOS and Linux . . . 5
1.7 Note about VectSharp.MuPDFUtils and .NET Framework . . . 5

2 Namespace Index 7
2.1 Package List . . . 7

3 Hierarchical Index 9
3.1 Class Hierarchy . . . 9

4 Class Index 13
4.1 Class List . . . 13

5 File Index 19
5.1 File List . . . 19

6 Namespace Documentation 21
6.1 VectSharp Namespace Reference . . . 21
6.1.1 Enumeration Type Documentation . . . 23
6.1.1.1 LineCaps . . . 23
6.1.1.2 LineJoins . . . 23
6.1.1.3 PixelFormats . . . 23
6.1.1.4 Script . . . 24
6.1.1.5 SegmentType . . . 24
6.1.1.6 TextAnchors . . . 24
6.1.1.7 TextBaselines . . . 24
6.1.1.8 UnbalancedStackActions . . . 24
6.2 VectSharp.Canvas Namespace Reference . . . 25
6.3 VectSharp.Filters Namespace Reference . . . 25
6.4 VectSharp.Fonts Namespace Reference . . . 26
6.5 VectSharp.ImageSharpUtils Namespace Reference . . . 26
6.6 VectSharp.Markdown Namespace Reference . . . 26
6.7 VectSharp.MarkdownCanvas Namespace Reference . . . 26
6.8 VectSharp.MuPDFUtils Namespace Reference . . . 27
6.9 VectSharp.PDF Namespace Reference . . . 27
6.10 VectSharp.Raster Namespace Reference . . . 27
6.11 VectSharp.Raster.ImageSharp Namespace Reference . . . 27

6.11.1 Enumeration Type Documentation . . . 27
6.11.1.1 OutputFormats . . . 28
6.12 VectSharp.SVG Namespace Reference . . . 28
6.13 VectSharp.ThreeD Namespace Reference . . . 28

7 Class Documentation **29**
7.1 VectSharp.ThreeD.AmbientLightSource Class Reference . . . 29
7.1.1 Detailed Description . . . 31
7.1.2 Constructor & Destructor Documentation . . . 31
7.1.2.1 AmbientLightSource() . . . 31
7.1.3 Member Function Documentation . . . 31
7.1.3.1 GetLightAt() . . . 32
7.1.3.2 GetObstruction() . . . 32
7.1.4 Property Documentation . . . 32
7.1.4.1 CastsShadow . . . 33
7.1.4.2 Intensity . . . 33
7.2 VectSharp.Canvas.AnimatedCanvas Class Reference . . . 33
7.2.1 Detailed Description . . . 34
7.2.2 Member Function Documentation . . . 34
7.2.2.1 Render() . . . 34
7.2.3 Member Data Documentation . . . 34
7.2.3.1 CurrentFrameProperty . . . 35
7.2.3.2 FrameCountProperty . . . 35
7.2.3.3 FrameRateProperty . . . 35
7.2.3.4 IsPlayingProperty . . . 36
7.2.4 Property Documentation . . . 36
7.2.4.1 CurrentFrame . . . 36
7.2.4.2 FrameCount . . . 36
7.2.4.3 FrameRate . . . 36
7.2.4.4 IsPlaying . . . 37
7.3 VectSharp.AnimatedPNG Class Reference . . . 37
7.3.1 Detailed Description . . . 37
7.3.2 Member Enumeration Documentation . . . 37
7.3.2.1 InterframeCompression . . . 38
7.3.3 Member Function Documentation . . . 38
7.3.3.1 Create() . . . 38
7.4 VectSharp.Animation Class Reference . . . 38
7.4.1 Detailed Description . . . 39
7.4.2 Constructor & Destructor Documentation . . . 39
7.4.2.1 Animation() . . . 39
7.4.3 Member Function Documentation . . . 40
7.4.3.1 AddFrame() . . . 40

7.4.3.2 GetFrameAtAbsolute() . . . 40
7.4.3.3 GetFrameAtRelative() . . . 41
7.4.3.4 RemoveLastFrame() . . . 41
7.4.4 Property Documentation . . . 41
7.4.4.1 Background . . . 41
7.4.4.2 Duration . . . 42
7.4.4.3 Frames . . . 42
7.4.4.4 Height . . . 42
7.4.4.5 LinearisationResolution . . . 42
7.4.4.6 RepeatCount . . . 42
7.4.4.7 Transitions . . . 43
7.4.4.8 Width . . . 43
7.5 VectSharp.ThreeD.AreaLightSource Class Reference . . . 43
7.5.1 Detailed Description . . . 44
7.5.2 Constructor & Destructor Documentation . . . 45
7.5.2.1 AreaLightSource() . . . 45
7.5.3 Member Function Documentation . . . 45
7.5.3.1 GetLightAt() . . . 45
7.5.3.2 GetObstruction() . . . 46
7.5.4 Property Documentation . . . 46
7.5.4.1 CastsShadow . . . 46
7.5.4.2 Center . . . 47
7.5.4.3 Direction . . . 47
7.5.4.4 DistanceAttenuationExponent . . . 47
7.5.4.5 Intensity . . . 47
7.5.4.6 PenumbraAttenuationExponent . . . 47
7.5.4.7 PenumbraRadius . . . 48
7.5.4.8 Radius . . . 48
7.5.4.9 ShadowSamplingPointCount . . . 48
7.5.4.10 SourceDistance . . . 48
7.6 VectSharp.Canvas.AvaloniaContextInterpreter Class Reference . . . 48
7.6.1 Detailed Description . . . 49
7.6.2 Member Enumeration Documentation . . . 49
7.6.2.1 TextOptions . . . 49
7.6.3 Member Function Documentation . . . 49
7.6.3.1 PaintToAnimatedCanvas() . . . 50
7.6.3.2 PaintToCanvas() [1/4] . . . 50
7.6.3.3 PaintToCanvas() [2/4] . . . 51
7.6.3.4 PaintToCanvas() [3/4] . . . 51
7.6.3.5 PaintToCanvas() [4/4] . . . 52
7.7 VectSharp.TrueTypeFile.Bearings Struct Reference . . . 53
7.7.1 Detailed Description . . . 53

7.7.2 Member Data Documentation . . . 53
7.7.2.1 LeftSideBearing . . . 53
7.7.2.2 RightSideBearing . . . 53
7.8 VectSharp.Filters.BoxBlurFilter Class Reference . . . 54
7.8.1 Detailed Description . . . 55
7.8.2 Constructor & Destructor Documentation . . . 55
7.8.2.1 BoxBlurFilter() . . . 55
7.8.3 Member Function Documentation . . . 55
7.8.3.1 Filter() . . . 55
7.8.4 Property Documentation . . . 56
7.8.4.1 BottomRightMargin . . . 56
7.8.4.2 BoxRadius . . . 56
7.8.4.3 TopLeftMargin . . . 56
7.9 VectSharp.Brush Class Reference . . . 57
7.9.1 Detailed Description . . . 57
7.9.2 Member Function Documentation . . . 57
7.9.2.1 MultiplyOpacity() . . . 57
7.9.2.2 operator Brush() . . . 58
7.10 VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord Struct Reference . . . 58
7.10.1 Detailed Description . . . 58
7.10.2 Constructor & Destructor Documentation . . . 59
7.10.2.1 ClassRangeRecord() . . . 59
7.10.3 Member Data Documentation . . . 59
7.10.3.1 Class . . . 59
7.10.3.2 EndGlyphID . . . 59
7.10.3.3 StartGlyphID . . . 59
7.11 VectSharp.Colour Struct Reference . . . 60
7.11.1 Detailed Description . . . 61
7.11.2 Member Function Documentation . . . 62
7.11.2.1 Equals() [1/2] . . . 62
7.11.2.2 Equals() [2/2] . . . 62
7.11.2.3 FromCSSString() . . . 62
7.11.2.4 FromHSL() . . . 62
7.11.2.5 FromLab() . . . 63
7.11.2.6 FromRgb() [1/3] . . . 63
7.11.2.7 FromRgb() [2/3] . . . 64
7.11.2.8 FromRgb() [3/3] . . . 64
7.11.2.9 FromRgba() [1/6] . . . 65
7.11.2.10 FromRgba() [2/6] . . . 65
7.11.2.11 FromRgba() [3/6] . . . 66
7.11.2.12 FromRgba() [4/6] . . . 66
7.11.2.13 FromRgba() [5/6] . . . 67

7.11.2.14 FromRgba() [6/6] . . . 67
7.11.2.15 FromXYZ() . . . 68
7.11.2.16 GetHashCode() . . . 68
7.11.2.17 operator"!=() . . . 68
7.11.2.18 operator==() . . . 68
7.11.2.19 ToCSSString() . . . 68
7.11.2.20 ToHSL() . . . 69
7.11.2.21 ToLab() . . . 69
7.11.2.22 ToXYZ() . . . 69
7.11.2.23 WithAlpha() [1/4] . . . 69
7.11.2.24 WithAlpha() [2/4] . . . 70
7.11.2.25 WithAlpha() [3/4] . . . 70
7.11.2.26 WithAlpha() [4/4] . . . 71
7.11.3 Member Data Documentation . . . 71
7.11.3.1 A . . . 71
7.11.3.2 a . . . 71
7.11.3.3 B . . . 71
7.11.3.4 G . . . 72
7.11.3.5 H . . . 72
7.11.3.6 L . . . 72
7.11.3.7 R . . . 72
7.11.3.8 S . . . 73
7.11.3.9 X . . . 73
7.11.3.10 Y . . . 73
7.12 VectSharp.ThreeD.ColourMaterial Class Reference . . . 73
7.12.1 Detailed Description . . . 74
7.12.2 Constructor & Destructor Documentation . . . 74
7.12.2.1 ColourMaterial() . . . 74
7.12.3 Member Function Documentation . . . 74
7.12.3.1 GetColour() . . . 75
7.12.4 Property Documentation . . . 75
7.12.4.1 Colour . . . 75
7.13 VectSharp.Filters.ColourMatrix Class Reference . . . 75
7.13.1 Detailed Description . . . 77
7.13.2 Constructor & Destructor Documentation . . . 78
7.13.2.1 ColourMatrix() . . . 78
7.13.3 Member Function Documentation . . . 78
7.13.3.1 Apply() [1/5] . . . 78
7.13.3.2 Apply() [2/5] . . . 79
7.13.3.3 Apply() [3/5] . . . 79
7.13.3.4 Apply() [4/5] . . . 80
7.13.3.5 Apply() [5/5] . . . 80

7.13.3.6 LuminanceToAlpha() . . . 80
7.13.3.7 LuminanceToColour() . . . 81
7.13.3.8 operator∗() . . . 81
7.13.3.9 ToColour() . . . 82
7.13.3.10 WithAlpha() . . . 82
7.13.4 Member Data Documentation . . . 83
7.13.4.1 AlphaInversion . . . 83
7.13.4.2 GreyScale . . . 83
7.13.4.3 Identity . . . 83
7.13.4.4 Inversion . . . 84
7.13.4.5 InvertedAlphaShift . . . 84
7.13.4.6 Pastel . . . 84
7.13.5 Property Documentation . . . 84
7.13.5.1 A1 . . . 84
7.13.5.2 A2 . . . 85
7.13.5.3 A3 . . . 85
7.13.5.4 A4 . . . 85
7.13.5.5 A5 . . . 85
7.13.5.6 B1 . . . 85
7.13.5.7 B2 . . . 86
7.13.5.8 B3 . . . 86
7.13.5.9 B4 . . . 86
7.13.5.10 B5 . . . 86
7.13.5.11 G1 . . . 86
7.13.5.12 G2 . . . 87
7.13.5.13 G3 . . . 87
7.13.5.14 G4 . . . 87
7.13.5.15 G5 . . . 87
7.13.5.16 R1 . . . 87
7.13.5.17 R2 . . . 88
7.13.5.18 R3 . . . 88
7.13.5.19 R4 . . . 88
7.13.5.20 R5 . . . 88
7.13.5.21 this[int y, int x] . . . 88
7.14 VectSharp.Filters.ColourMatrixFilter Class Reference . . . 89
7.14.1 Detailed Description . . . 90
7.14.2 Constructor & Destructor Documentation . . . 90
7.14.2.1 ColourMatrixFilter() . . . 90
7.14.3 Member Function Documentation . . . 90
7.14.3.1 Filter() . . . 90
7.14.4 Property Documentation . . . 91
7.14.4.1 BottomRightMargin . . . 91

7.14.4.2 ColourMatrix . . . 91
7.14.4.3 TopLeftMargin . . . 91
7.15 VectSharp.Colours Class Reference . . . 92
7.15.1 Detailed Description . . . 98
7.15.2 Member Data Documentation . . . 98
7.15.2.1 AliceBlue . . . 98
7.15.2.2 AntiqueWhite . . . 98
7.15.2.3 Aqua . . . 98
7.15.2.4 Aquamarine . . . 98
7.15.2.5 Azure . . . 99
7.15.2.6 Beige . . . 99
7.15.2.7 Bisque . . . 99
7.15.2.8 Black . . . 99
7.15.2.9 BlanchedAlmond . . . 99
7.15.2.10 Blue . . . 100
7.15.2.11 BlueViolet . . . 100
7.15.2.12 Brown . . . 100
7.15.2.13 BurlyWood . . . 100
7.15.2.14 CadetBlue . . . 100
7.15.2.15 Chartreuse . . . 101
7.15.2.16 Chocolate . . . 101
7.15.2.17 Coral . . . 101
7.15.2.18 CornflowerBlue . . . 101
7.15.2.19 Cornsilk . . . 101
7.15.2.20 Crimson . . . 102
7.15.2.21 Cyan . . . 102
7.15.2.22 DarkBlue . . . 102
7.15.2.23 DarkCyan . . . 102
7.15.2.24 DarkGoldenRod . . . 102
7.15.2.25 DarkGray . . . 103
7.15.2.26 DarkGreen . . . 103
7.15.2.27 DarkGrey . . . 103
7.15.2.28 DarkKhaki . . . 103
7.15.2.29 DarkMagenta . . . 103
7.15.2.30 DarkOliveGreen . . . 104
7.15.2.31 DarkOrange . . . 104
7.15.2.32 DarkOrchid . . . 104
7.15.2.33 DarkRed . . . 104
7.15.2.34 DarkSalmon . . . 104
7.15.2.35 DarkSeaGreen . . . 105
7.15.2.36 DarkSlateBlue . . . 105
7.15.2.37 DarkSlateGray . . . 105

7.15.2.38 DarkSlateGrey . . . 105
7.15.2.39 DarkTurquoise . . . 105
7.15.2.40 DarkViolet . . . 106
7.15.2.41 DeepPink . . . 106
7.15.2.42 DeepSkyBlue . . . 106
7.15.2.43 DimGray . . . 106
7.15.2.44 DimGrey . . . 106
7.15.2.45 DodgerBlue . . . 107
7.15.2.46 FireBrick . . . 107
7.15.2.47 FloralWhite . . . 107
7.15.2.48 ForestGreen . . . 107
7.15.2.49 Fuchsia . . . 107
7.15.2.50 Gainsboro . . . 108
7.15.2.51 GhostWhite . . . 108
7.15.2.52 Gold . . . 108
7.15.2.53 GoldenRod . . . 108
7.15.2.54 Gray . . . 108
7.15.2.55 Green . . . 109
7.15.2.56 GreenYellow . . . 109
7.15.2.57 Grey . . . 109
7.15.2.58 HoneyDew . . . 109
7.15.2.59 HotPink . . . 109
7.15.2.60 IndianRed . . . 110
7.15.2.61 Indigo . . . 110
7.15.2.62 Ivory . . . 110
7.15.2.63 Khaki . . . 110
7.15.2.64 Lavender . . . 110
7.15.2.65 LavenderBlush . . . 111
7.15.2.66 LawnGreen . . . 111
7.15.2.67 LemonChiffon . . . 111
7.15.2.68 LightBlue . . . 111
7.15.2.69 LightCoral . . . 111
7.15.2.70 LightCyan . . . 112
7.15.2.71 LightGoldenRodYellow . . . 112
7.15.2.72 LightGray . . . 112
7.15.2.73 LightGreen . . . 112
7.15.2.74 LightGrey . . . 112
7.15.2.75 LightPink . . . 113
7.15.2.76 LightSalmon . . . 113
7.15.2.77 LightSeaGreen . . . 113
7.15.2.78 LightSkyBlue . . . 113
7.15.2.79 LightSlateGray . . . 113

7.15.2.80 LightSlateGrey . . . 114
7.15.2.81 LightSteelBlue . . . 114
7.15.2.82 LightYellow . . . 114
7.15.2.83 Lime . . . 114
7.15.2.84 LimeGreen . . . 114
7.15.2.85 Linen . . . 115
7.15.2.86 Magenta . . . 115
7.15.2.87 Maroon . . . 115
7.15.2.88 MediumAquaMarine . . . 115
7.15.2.89 MediumBlue . . . 115
7.15.2.90 MediumOrchid . . . 116
7.15.2.91 MediumPurple . . . 116
7.15.2.92 MediumSeaGreen . . . 116
7.15.2.93 MediumSlateBlue . . . 116
7.15.2.94 MediumSpringGreen . . . 116
7.15.2.95 MediumTurquoise . . . 117
7.15.2.96 MediumVioletRed . . . 117
7.15.2.97 MidnightBlue . . . 117
7.15.2.98 MintCream . . . 117
7.15.2.99 MistyRose . . . 117
7.15.2.100 Moccasin . . . 118
7.15.2.101 NavajoWhite . . . 118
7.15.2.102 Navy . . . 118
7.15.2.103 OldLace . . . 118
7.15.2.104 Olive . . . 118
7.15.2.105 OliveDrab . . . 119
7.15.2.106 Orange . . . 119
7.15.2.107 OrangeRed . . . 119
7.15.2.108 Orchid . . . 119
7.15.2.109 PaleGoldenRod . . . 119
7.15.2.110 PaleGreen . . . 120
7.15.2.111 PaleTurquoise . . . 120
7.15.2.112 PaleVioletRed . . . 120
7.15.2.113 PapayaWhip . . . 120
7.15.2.114 PeachPuff . . . 120
7.15.2.115 Peru . . . 121
7.15.2.116 Pink . . . 121
7.15.2.117 Plum . . . 121
7.15.2.118 PowderBlue . . . 121
7.15.2.119 Purple . . . 121
7.15.2.120 RebeccaPurple . . . 122
7.15.2.121 Red . . . 122

7.15.2.122 RosyBrown . . . 122
7.15.2.123 RoyalBlue . . . 122
7.15.2.124 SaddleBrown . . . 122
7.15.2.125 Salmon . . . 123
7.15.2.126 SandyBrown . . . 123
7.15.2.127 SeaGreen . . . 123
7.15.2.128 SeaShell . . . 123
7.15.2.129 Sienna . . . 123
7.15.2.130 Silver . . . 124
7.15.2.131 SkyBlue . . . 124
7.15.2.132 SlateBlue . . . 124
7.15.2.133 SlateGray . . . 124
7.15.2.134 SlateGrey . . . 124
7.15.2.135 Snow . . . 125
7.15.2.136 SpringGreen . . . 125
7.15.2.137 SteelBlue . . . 125
7.15.2.138 Tan . . . 125
7.15.2.139 Teal . . . 125
7.15.2.140 Thistle . . . 126
7.15.2.141 Tomato . . . 126
7.15.2.142 Turquoise . . . 126
7.15.2.143 Violet . . . 126
7.15.2.144 Wheat . . . 126
7.15.2.145 White . . . 127
7.15.2.146 WhiteSmoke . . . 127
7.15.2.147 Yellow . . . 127
7.15.2.148 YellowGreen . . . 127
7.16 VectSharp.Filters.CompositeLocationInvariantFilter Class Reference . . . 128
7.16.1 Detailed Description . . . 129
7.16.2 Constructor & Destructor Documentation . . . 129
7.16.2.1 CompositeLocationInvariantFilter() [1/2] . . . 129
7.16.2.2 CompositeLocationInvariantFilter() [2/2] . . . 129
7.16.3 Member Function Documentation . . . 129
7.16.3.1 Filter() . . . 130
7.16.4 Property Documentation . . . 130
7.16.4.1 BottomRightMargin . . . 130
7.16.4.2 Filters . . . 130
7.16.4.3 TopLeftMargin . . . 131
7.17 VectSharp.AnimatedPNG.CompressedFrame Class Reference . . . 131
7.17.1 Detailed Description . . . 132
7.17.2 Constructor & Destructor Documentation . . . 132
7.17.2.1 CompressedFrame() [1/2] . . . 132

7.17.2.2 CompressedFrame() [2/2] . . . 132
7.17.3 Member Function Documentation . . . 133
7.17.3.1 Dispose() . . . 133
7.17.4 Property Documentation . . . 133
7.17.4.1 Duration . . . 133
7.18 VectSharp.Filters.ConvolutionFilter Class Reference . . . 134
7.18.1 Detailed Description . . . 135
7.18.2 Constructor & Destructor Documentation . . . 135
7.18.2.1 ConvolutionFilter() . . . 135
7.18.3 Member Function Documentation . . . 136
7.18.3.1 Filter() . . . 136
7.18.4 Property Documentation . . . 136
7.18.4.1 Bias . . . 136
7.18.4.2 BottomRightMargin . . . 136
7.18.4.3 Kernel . . . 137
7.18.4.4 Normalisation . . . 137
7.18.4.5 PreserveAlpha . . . 137
7.18.4.6 Scale . . . 137
7.18.4.7 TopLeftMargin . . . 137
7.19 VectSharp.DefaultFontLibrary Class Reference . . . 138
7.19.1 Detailed Description . . . 138
7.19.2 Member Function Documentation . . . 139
7.19.2.1 ResolveFontFamily() [1/2] . . . 139
7.19.2.2 ResolveFontFamily() [2/2] . . . 139
7.20 VectSharp.Font.DetailedFontMetrics Class Reference . . . 140
7.20.1 Detailed Description . . . 140
7.20.2 Property Documentation . . . 140
7.20.2.1 AdvanceWidth . . . 140
7.20.2.2 Bottom . . . 141
7.20.2.3 Height . . . 141
7.20.2.4 LeftSideBearing . . . 141
7.20.2.5 RightSideBearing . . . 141
7.20.2.6 Top . . . 141
7.20.2.7 Width . . . 142
7.21 VectSharp.DisposableIntPtr Class Reference . . . 142
7.21.1 Detailed Description . . . 142
7.21.2 Constructor & Destructor Documentation . . . 143
7.21.2.1 DisposableIntPtr() . . . 143
7.21.3 Member Function Documentation . . . 143
7.21.3.1 Dispose() . . . 143
7.21.4 Member Data Documentation . . . 143
7.21.4.1 InternalPointer . . . 143

7.22 VectSharp.Document Class Reference . . . 144
7.22.1 Detailed Description . . . 144
7.22.2 Constructor & Destructor Documentation . . . 144
7.22.2.1 Document() . . . 144
7.22.3 Member Data Documentation . . . 144
7.22.3.1 Pages . . . 144
7.23 VectSharp.Canvas.FilterOption Class Reference . . . 145
7.23.1 Detailed Description . . . 145
7.23.2 Member Enumeration Documentation . . . 145
7.23.2.1 FilterOperations . . . 145
7.23.3 Constructor & Destructor Documentation . . . 146
7.23.3.1 FilterOption() . . . 146
7.23.4 Member Data Documentation . . . 146
7.23.4.1 Default . . . 146
7.23.5 Property Documentation . . . 146
7.23.5.1 Operation . . . 147
7.23.5.2 RasterisationResolution . . . 147
7.23.5.3 RasterisationResolutionRelative . . . 147
7.24 VectSharp.PDF.PDFContextInterpreter.FilterOption Class Reference . . . 147
7.24.1 Detailed Description . . . 148
7.24.2 Member Enumeration Documentation . . . 148
7.24.2.1 FilterOperations . . . 148
7.24.3 Constructor & Destructor Documentation . . . 148
7.24.3.1 FilterOption() . . . 148
7.24.4 Member Data Documentation . . . 149
7.24.4.1 Default . . . 149
7.24.5 Property Documentation . . . 149
7.24.5.1 Operation . . . 149
7.24.5.2 RasterisationResolution . . . 149
7.24.5.3 RasterisationResolutionRelative . . . 150
7.25 VectSharp.SVG.SVGContextInterpreter.FilterOption Class Reference . . . 150
7.25.1 Detailed Description . . . 151
7.25.2 Member Enumeration Documentation . . . 151
7.25.2.1 FilterOperations . . . 151
7.25.3 Constructor & Destructor Documentation . . . 151
7.25.3.1 FilterOption() . . . 151
7.25.4 Member Data Documentation . . . 151
7.25.4.1 Default . . . 152
7.25.5 Property Documentation . . . 152
7.25.5.1 Operation . . . 152
7.25.5.2 RasterisationResolution . . . 152
7.25.5.3 RasterisationResolutionRelative . . . 152

7.26 VectSharp.Filters.FilterWithRasterisableParameter Class Reference . . . 153
7.26.1 Detailed Description . . . 153
7.26.2 Member Function Documentation . . . 153
7.26.2.1 Dispose() . . . 154
7.26.2.2 RasteriseParameter() . . . 154
7.27 VectSharp.Font Class Reference . . . 154
7.27.1 Detailed Description . . . 155
7.27.2 Constructor & Destructor Documentation . . . 156
7.27.2.1 Font() [1/3] . . . 156
7.27.2.2 Font() [2/3] . . . 156
7.27.2.3 Font() [3/3] . . . 156
7.27.3 Member Function Documentation . . . 158
7.27.3.1 MeasureText() . . . 158
7.27.3.2 MeasureTextAdvanced() . . . 158
7.27.4 Member Data Documentation . . . 159
7.27.4.1 EnableKerning . . . 159
7.27.5 Property Documentation . . . 159
7.27.5.1 Ascent . . . 159
7.27.5.2 Descent . . . 159
7.27.5.3 FontFamily . . . 159
7.27.5.4 FontSize . . . 160
7.27.5.5 Underline . . . 160
7.27.5.6 WinAscent . . . 160
7.27.5.7 YMax . . . 160
7.27.5.8 YMin . . . 160
7.28 VectSharp.FontFamily Class Reference . . . 161
7.28.1 Detailed Description . . . 162
7.28.2 Member Enumeration Documentation . . . 162
7.28.2.1 StandardFontFamilies . . . 162
7.28.3 Constructor & Destructor Documentation . . . 163
7.28.3.1 FontFamily() [1/4] . . . 163
7.28.3.2 FontFamily() [2/4] . . . 163
7.28.3.3 FontFamily() [3/4] . . . 163
7.28.3.4 FontFamily() [4/4] . . . 164
7.28.4 Member Function Documentation . . . 164
7.28.4.1 ResolveFontFamily() [1/4] . . . 164
7.28.4.2 ResolveFontFamily() [2/4] . . . 164
7.28.4.3 ResolveFontFamily() [3/4] . . . 165
7.28.4.4 ResolveFontFamily() [4/4] . . . 165
7.28.5 Member Data Documentation . . . 166
7.28.5.1 StandardFamilies . . . 166
7.28.5.2 StandardFontFamilyResources . . . 166

7.28.6 Property Documentation . . . 166
7.28.6.1 DefaultFontLibrary . . . 166
7.28.6.2 FamilyName . . . 167
7.28.6.3 FileName . . . 167
7.28.6.4 IsBold . . . 167
7.28.6.5 IsItalic . . . 167
7.28.6.6 IsOblique . . . 167
7.28.6.7 IsStandardFamily . . . 168
7.28.6.8 TrueTypeFile . . . 168
7.29 VectSharp.FontFamilyCreationException Class Reference . . . 168
7.29.1 Detailed Description . . . 169
7.29.2 Constructor & Destructor Documentation . . . 169
7.29.2.1 FontFamilyCreationException() . . . 169
7.29.3 Property Documentation . . . 169
7.29.3.1 FontFamily . . . 169
7.30 VectSharp.FontLibrary Class Reference . . . 170
7.30.1 Detailed Description . . . 171
7.30.2 Member Function Documentation . . . 171
7.30.2.1 ResolveFontFamily() [1/4] . . . 171
7.30.2.2 ResolveFontFamily() [2/4] . . . 172
7.30.2.3 ResolveFontFamily() [3/4] . . . 172
7.30.2.4 ResolveFontFamily() [4/4] . . . 173
7.31 VectSharp.Font.FontUnderline Class Reference . . . 173
7.31.1 Detailed Description . . . 173
7.31.2 Property Documentation . . . 174
7.31.2.1 FollowItalicAngle . . . 174
7.31.2.2 LineCap . . . 174
7.31.2.3 Position . . . 174
7.31.2.4 SkipDescenders . . . 174
7.31.2.5 Thickness . . . 175
7.32 VectSharp.Markdown.FormattedString Struct Reference . . . 175
7.32.1 Detailed Description . . . 175
7.32.2 Constructor & Destructor Documentation . . . 175
7.32.2.1 FormattedString() . . . 175
7.32.3 Property Documentation . . . 176
7.32.3.1 Colour . . . 176
7.32.3.2 IsBold . . . 176
7.32.3.3 IsItalic . . . 176
7.32.3.4 Text . . . 177
7.33 VectSharp.FormattedText Class Reference . . . 177
7.33.1 Detailed Description . . . 177
7.33.2 Constructor & Destructor Documentation . . . 178

7.33.2.1 FormattedText() . . . 178
7.33.3 Member Function Documentation . . . 178
7.33.3.1 Format() [1/2] . . . 178
7.33.3.2 Format() [2/2] . . . 179
7.33.4 Property Documentation . . . 180
7.33.4.1 Brush . . . 180
7.33.4.2 Font . . . 181
7.33.4.3 Script . . . 181
7.33.4.4 Text . . . 181
7.34 VectSharp.FormattedTextExtensions Class Reference . . . 181
7.34.1 Detailed Description . . . 181
7.34.2 Member Function Documentation . . . 181
7.34.2.1 Measure() . . . 181
7.35 VectSharp.Frame Class Reference . . . 182
7.35.1 Detailed Description . . . 182
7.35.2 Constructor & Destructor Documentation . . . 182
7.35.2.1 Frame() . . . 182
7.35.3 Property Documentation . . . 183
7.35.3.1 Duration . . . 183
7.35.3.2 Graphics . . . 183
7.36 VectSharp.Filters.GaussianBlurFilter Class Reference . . . 183
7.36.1 Detailed Description . . . 184
7.36.2 Constructor & Destructor Documentation . . . 184
7.36.2.1 GaussianBlurFilter() . . . 184
7.36.3 Member Function Documentation . . . 185
7.36.3.1 Filter() . . . 185
7.36.4 Property Documentation . . . 185
7.36.4.1 BottomRightMargin . . . 185
7.36.4.2 StandardDeviation . . . 185
7.36.4.3 TopLeftMargin . . . 186
7.37 VectSharp.GradientBrush Class Reference . . . 186
7.37.1 Detailed Description . . . 186
7.37.2 Property Documentation . . . 187
7.37.2.1 GradientStops . . . 187
7.38 VectSharp.GradientStop Struct Reference . . . 187
7.38.1 Detailed Description . . . 187
7.38.2 Constructor & Destructor Documentation . . . 187
7.38.2.1 GradientStop() . . . 187
7.38.3 Member Function Documentation . . . 188
7.38.3.1 MultiplyOpacity() . . . 188
7.38.4 Property Documentation . . . 188
7.38.4.1 Colour . . . 188

7.38.4.2 Offset . . . 189
7.39 VectSharp.GradientStops Class Reference . . . 189
7.39.1 Detailed Description . . . 190
7.39.2 Constructor & Destructor Documentation . . . 190
7.39.2.1 GradientStops() [1/2] . . . 190
7.39.2.2 GradientStops() [2/2] . . . 190
7.39.3 Member Function Documentation . . . 190
7.39.3.1 GetColourAt() . . . 191
7.39.3.2 GetEnumerator() . . . 191
7.39.4 Member Data Documentation . . . 191
7.39.4.1 StopTolerance . . . 191
7.39.5 Property Documentation . . . 191
7.39.5.1 Count . . . 191
7.39.5.2 this[int index] . . . 192
7.40 VectSharp.Graphics Class Reference . . . 192
7.40.1 Detailed Description . . . 195
7.40.2 Member Function Documentation . . . 195
7.40.2.1 CopyToIGraphicsContext() . . . 195
7.40.2.2 Crop() [1/2] . . . 196
7.40.2.3 Crop() [2/2] . . . 196
7.40.2.4 DrawGraphics() [1/6] . . . 196
7.40.2.5 DrawGraphics() [2/6] . . . 197
7.40.2.6 DrawGraphics() [3/6] . . . 197
7.40.2.7 DrawGraphics() [4/6] . . . 198
7.40.2.8 DrawGraphics() [5/6] . . . 198
7.40.2.9 DrawGraphics() [6/6] . . . 198
7.40.2.10 DrawRasterImage() [1/5] . . . 199
7.40.2.11 DrawRasterImage() [2/5] . . . 199
7.40.2.12 DrawRasterImage() [3/5] . . . 200
7.40.2.13 DrawRasterImage() [4/5] . . . 201
7.40.2.14 DrawRasterImage() [5/5] . . . 201
7.40.2.15 FillPath() . . . 201
7.40.2.16 FillRectangle() [1/2] . . . 202
7.40.2.17 FillRectangle() [2/2] . . . 202
7.40.2.18 FillText() [1/4] . . . 203
7.40.2.19 FillText() [2/4] . . . 203
7.40.2.20 FillText() [3/4] . . . 204
7.40.2.21 FillText() [4/4] . . . 204
7.40.2.22 FillTextOnPath() . . . 205
7.40.2.23 FillTextUnderline() [1/4] . . . 205
7.40.2.24 FillTextUnderline() [2/4] . . . 206
7.40.2.25 FillTextUnderline() [3/4] . . . 206

7.40.2.26 FillTextUnderline() [4/4] . . . 207
7.40.2.27 GetBounds() . . . 207
7.40.2.28 GetTags() . . . 207
7.40.2.29 Linearise() . . . 208
7.40.2.30 MeasureText() [1/2] . . . 208
7.40.2.31 MeasureText() [2/2] . . . 208
7.40.2.32 Restore() . . . 209
7.40.2.33 Rotate() . . . 209
7.40.2.34 RotateAt() . . . 209
7.40.2.35 Save() . . . 210
7.40.2.36 Scale() . . . 210
7.40.2.37 SetClippingPath() [1/3] . . . 210
7.40.2.38 SetClippingPath() [2/3] . . . 211
7.40.2.39 SetClippingPath() [3/3] . . . 211
7.40.2.40 StrokePath() . . . 212
7.40.2.41 StrokeRectangle() [1/2] . . . 212
7.40.2.42 StrokeRectangle() [2/2] . . . 213
7.40.2.43 StrokeText() [1/4] . . . 213
7.40.2.44 StrokeText() [2/4] . . . 214
7.40.2.45 StrokeText() [3/4] . . . 215
7.40.2.46 StrokeText() [4/4] . . . 215
7.40.2.47 StrokeTextOnPath() . . . 216
7.40.2.48 StrokeTextUnderline() [1/4] . . . 217
7.40.2.49 StrokeTextUnderline() [2/4] . . . 217
7.40.2.50 StrokeTextUnderline() [3/4] . . . 218
7.40.2.51 StrokeTextUnderline() [4/4] . . . 219
7.40.2.52 Transform() [1/3] . . . 219
7.40.2.53 Transform() [2/3] . . . 220
7.40.2.54 Transform() [3/3] . . . 220
7.40.2.55 Translate() [1/2] . . . 221
7.40.2.56 Translate() [2/2] . . . 221
7.40.2.57 TryRasterise() . . . 222
7.40.3 Member Data Documentation . . . 222
7.40.3.1 RasterisationMethod . . . 222
7.40.4 Property Documentation . . . 222
7.40.4.1 UnbalancedStackAction . . . 223
7.41 VectSharp.GraphicsPath Class Reference . . . 223
7.41.1 Detailed Description . . . 225
7.41.2 Member Function Documentation . . . 225
7.41.2.1 AddPath() . . . 225
7.41.2.2 AddSmoothSpline() . . . 225
7.41.2.3 AddText() [1/2] . . . 226

7.41.2.4 AddText() [2/2] . . . 226
7.41.2.5 AddTextOnPath() . . . 227
7.41.2.6 AddTextUnderline() . . . 227
7.41.2.7 Arc() [1/2] . . . 228
7.41.2.8 Arc() [2/2] . . . 228
7.41.2.9 Close() . . . 229
7.41.2.10 CubicBezierTo() [1/2] . . . 229
7.41.2.11 CubicBezierTo() [2/2] . . . 230
7.41.2.12 Discretise() . . . 230
7.41.2.13 EllipticalArc() . . . 230
7.41.2.14 GetBounds() . . . 231
7.41.2.15 GetFigures() . . . 231
7.41.2.16 GetLinearisationPointsNormals() . . . 231
7.41.2.17 GetNormalAtAbsolute() . . . 232
7.41.2.18 GetNormalAtRelative() . . . 232
7.41.2.19 GetPointAtAbsolute() . . . 233
7.41.2.20 GetPointAtRelative() . . . 233
7.41.2.21 GetPoints() . . . 233
7.41.2.22 GetTangentAtAbsolute() . . . 234
7.41.2.23 GetTangentAtRelative() . . . 234
7.41.2.24 Linearise() . . . 234
7.41.2.25 LineTo() [1/2] . . . 235
7.41.2.26 LineTo() [2/2] . . . 235
7.41.2.27 MeasureLength() . . . 236
7.41.2.28 MoveTo() [1/2] . . . 236
7.41.2.29 MoveTo() [2/2] . . . 236
7.41.2.30 QuadraticBezierTo() [1/2] . . . 237
7.41.2.31 QuadraticBezierTo() [2/2] . . . 237
7.41.2.32 Transform() . . . 238
7.41.2.33 Triangulate() . . . 238
7.41.3 Property Documentation . . . 238
7.41.3.1 Segments . . . 239
7.42 VectSharp.Markdown.HTTPUtils Class Reference . . . 239
7.42.1 Detailed Description . . . 239
7.42.2 Member Function Documentation . . . 239
7.42.2.1 ResolveImageURI() . . . 240
7.42.3 Member Data Documentation . . . 240
7.42.3.1 path . . . 240
7.42.4 Property Documentation . . . 240
7.42.4.1 LogDownloads . . . 240
7.43 VectSharp.IEasing Interface Reference . . . 241
7.43.1 Detailed Description . . . 241

7.43.2 Member Function Documentation . . . 241
7.43.2.1 Ease() . . . 241
7.44 VectSharp.Filters.IFilter Interface Reference . . . 242
7.44.1 Detailed Description . . . 242
7.44.2 Property Documentation . . . 243
7.44.2.1 BottomRightMargin . . . 243
7.44.2.2 TopLeftMargin . . . 243
7.45 VectSharp.Filters.IFilterWithLocation Interface Reference . . . 243
7.45.1 Detailed Description . . . 244
7.45.2 Member Function Documentation . . . 244
7.45.2.1 Filter() . . . 244
7.46 VectSharp.Filters.IFilterWithRasterisableParameter Interface Reference . . . 244
7.46.1 Detailed Description . . . 245
7.46.2 Member Function Documentation . . . 245
7.46.2.1 RasteriseParameter() . . . 245
7.47 VectSharp.IFontLibrary Interface Reference . . . 246
7.47.1 Detailed Description . . . 247
7.47.2 Member Function Documentation . . . 247
7.47.2.1 ResolveFontFamily() [1/4] . . . 247
7.47.2.2 ResolveFontFamily() [2/4] . . . 247
7.47.2.3 ResolveFontFamily() [3/4] . . . 248
7.47.2.4 ResolveFontFamily() [4/4] . . . 248
7.48 VectSharp.IGraphicsContext Interface Reference . . . 249
7.48.1 Detailed Description . . . 250
7.48.2 Member Function Documentation . . . 250
7.48.2.1 CubicBezierTo() . . . 250
7.48.2.2 DrawFilteredGraphics() . . . 251
7.48.2.3 DrawRasterImage() . . . 251
7.48.2.4 FillText() . . . 252
7.48.2.5 LineTo() . . . 252
7.48.2.6 MoveTo() . . . 252
7.48.2.7 Rectangle() . . . 253
7.48.2.8 Rotate() . . . 253
7.48.2.9 Scale() . . . 253
7.48.2.10 SetFillStyle() [1/2] . . . 254
7.48.2.11 SetFillStyle() [2/2] . . . 254
7.48.2.12 SetLineDash() . . . 254
7.48.2.13 SetStrokeStyle() [1/2] . . . 254
7.48.2.14 SetStrokeStyle() [2/2] . . . 255
7.48.2.15 StrokeText() . . . 255
7.48.2.16 Transform() . . . 255
7.48.2.17 Translate() . . . 256

7.48.3 Property Documentation . . . 256
7.48.3.1 FillStyle . . . 256
7.48.3.2 Font . . . 256
7.48.3.3 Height . . . 257
7.48.3.4 LineCap . . . 257
7.48.3.5 LineJoin . . . 257
7.48.3.6 LineWidth . . . 257
7.48.3.7 StrokeStyle . . . 257
7.48.3.8 Tag . . . 258
7.48.3.9 TextBaseline . . . 258
7.48.3.10 Width . . . 258
7.49 VectSharp.ThreeD.ILightSource Interface Reference . . . 258
7.49.1 Detailed Description . . . 259
7.49.2 Member Function Documentation . . . 259
7.49.2.1 GetLightAt() . . . 259
7.49.2.2 GetObstruction() . . . 259
7.49.3 Property Documentation . . . 260
7.49.3.1 CastsShadow . . . 260
7.50 VectSharp.Filters.ILocationInvariantFilter Interface Reference . . . 260
7.50.1 Detailed Description . . . 261
7.50.2 Member Function Documentation . . . 261
7.50.2.1 Filter() . . . 261
7.51 VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter Class Reference . . . 261
7.51.1 Detailed Description . . . 262
7.51.2 Member Function Documentation . . . 262
7.51.2.1 Rasterise() . . . 263
7.51.2.2 SaveAsAnimatedGIF() [1/3] . . . 263
7.51.2.3 SaveAsAnimatedGIF() [2/3] . . . 264
7.51.2.4 SaveAsAnimatedGIF() [3/3] . . . 264
7.51.2.5 SaveAsAnimatedPNG() [1/2] . . . 265
7.51.2.6 SaveAsAnimatedPNG() [2/2] . . . 265
7.51.2.7 SaveAsImage() [1/3] . . . 266
7.51.2.8 SaveAsImage() [2/3] . . . 266
7.51.2.9 SaveAsImage() [3/3] . . . 267
7.51.2.10 SaveAsRawBytes() [1/2] . . . 267
7.51.2.11 SaveAsRawBytes() [2/2] . . . 268
7.52 VectSharp.ImageSharpUtils.ImageURIParser Class Reference . . . 268
7.52.1 Detailed Description . . . 269
7.52.2 Member Function Documentation . . . 269
7.52.2.1 Parser() . . . 269
7.53 VectSharp.MuPDFUtils.ImageURIParser Class Reference . . . 269
7.53.1 Detailed Description . . . 270

7.53.2 Member Function Documentation . . . 270
7.53.2.1 Parser() . . . 270
7.54 VectSharp.ThreeD.IMaterial Interface Reference . . . 270
7.54.1 Detailed Description . . . 271
7.54.2 Member Function Documentation . . . 271
7.54.2.1 GetColour() . . . 271
7.55 VectSharp.ThreeD.IScene Interface Reference . . . 272
7.55.1 Detailed Description . . . 273
7.55.2 Member Function Documentation . . . 273
7.55.2.1 AddElement() . . . 273
7.55.2.2 AddRange() . . . 273
7.55.2.3 Replace() [1/2] . . . 273
7.55.2.4 Replace() [2/2] . . . 274
7.55.3 Property Documentation . . . 274
7.55.3.1 SceneElements . . . 274
7.55.3.2 SceneLock . . . 274
7.56 VectSharp.ThreeD.LightIntensity Struct Reference . . . 275
7.56.1 Detailed Description . . . 275
7.56.2 Constructor & Destructor Documentation . . . 275
7.56.2.1 LightIntensity() . . . 275
7.56.3 Member Function Documentation . . . 276
7.56.3.1 Deconstruct() . . . 276
7.56.4 Member Data Documentation . . . 276
7.56.4.1 Direction . . . 276
7.56.4.2 Intensity . . . 276
7.57 VectSharp.LinearEasing Class Reference . . . 277
7.57.1 Detailed Description . . . 277
7.57.2 Constructor & Destructor Documentation . . . 277
7.57.2.1 LinearEasing() . . . 277
7.57.3 Member Function Documentation . . . 278
7.57.3.1 Ease() . . . 278
7.58 VectSharp.LinearGradientBrush Class Reference . . . 278
7.58.1 Detailed Description . . . 279
7.58.2 Constructor & Destructor Documentation . . . 279
7.58.2.1 LinearGradientBrush() [1/2] . . . 279
7.58.2.2 LinearGradientBrush() [2/2] . . . 280
7.58.3 Member Function Documentation . . . 280
7.58.3.1 MultiplyOpacity() . . . 280
7.58.3.2 RelativeTo() . . . 281
7.58.4 Property Documentation . . . 281
7.58.4.1 EndPoint . . . 281
7.58.4.2 StartPoint . . . 282

7.59 VectSharp.LineDash Struct Reference . . . 282
7.59.1 Detailed Description . . . 282
7.59.2 Constructor & Destructor Documentation . . . 282
7.59.2.1 LineDash() . . . 282
7.59.3 Member Data Documentation . . . 283
7.59.3.1 Phase . . . 283
7.59.3.2 SolidLine . . . 283
7.59.3.3 UnitsOff . . . 283
7.59.3.4 UnitsOn . . . 283
7.60 VectSharp.Markdown.Margins Class Reference . . . 284
7.60.1 Detailed Description . . . 284
7.60.2 Constructor & Destructor Documentation . . . 284
7.60.2.1 Margins() . . . 284
7.60.3 Property Documentation . . . 285
7.60.3.1 Bottom . . . 285
7.60.3.2 Left . . . 285
7.60.3.3 Right . . . 285
7.60.3.4 Top . . . 285
7.61 VectSharp.MarkdownCanvas.MarkdownCanvasControl Class Reference . . . 286
7.61.1 Detailed Description . . . 287
7.61.2 Constructor & Destructor Documentation . . . 287
7.61.2.1 MarkdownCanvasControl() . . . 287
7.61.3 Member Data Documentation . . . 287
7.61.3.1 DocumentProperty . . . 288
7.61.3.2 DocumentSourceProperty . . . 288
7.61.3.3 MaxRenderWidthProperty . . . 288
7.61.3.4 MinRenderWidthProperty . . . 288
7.61.3.5 MinVariationProperty . . . 289
7.61.3.6 TextConversionOptionsProperty . . . 289
7.61.4 Property Documentation . . . 289
7.61.4.1 Document . . . 289
7.61.4.2 DocumentSource . . . 289
7.61.4.3 MaxRenderWidth . . . 290
7.61.4.4 MinRenderWidth . . . 290
7.61.4.5 MinVariation . . . 290
7.61.4.6 Renderer . . . 290
7.61.4.7 TextConversionOption . . . 291
7.62 VectSharp.Markdown.MarkdownRenderer Class Reference . . . 291
7.62.1 Detailed Description . . . 294
7.62.2 Member Enumeration Documentation . . . 295
7.62.2.1 VerticalAlignment . . . 295
7.62.3 Member Function Documentation . . . 295

7.62.3.1 Render() [1/2] . . . 295
7.62.3.2 Render() [2/2] . . . 295
7.62.3.3 RenderSinglePage() [1/2] . . . 296
7.62.3.4 RenderSinglePage() [2/2] . . . 296
7.62.4 Property Documentation . . . 297
7.62.4.1 AllowPageBreak . . . 297
7.62.4.2 BackgroundColour . . . 297
7.62.4.3 BaseFontSize . . . 297
7.62.4.4 BaseImageUri . . . 298
7.62.4.5 BaseLinkUri . . . 298
7.62.4.6 BoldFontFamily . . . 298
7.62.4.7 BoldItalicFontFamily . . . 298
7.62.4.8 BoldUnderlineThickness . . . 298
7.62.4.9 Bullets . . . 299
7.62.4.10 CodeBlockBackgroundColour . . . 299
7.62.4.11 CodeFont . . . 299
7.62.4.12 CodeFontBold . . . 299
7.62.4.13 CodeFontBoldItalic . . . 300
7.62.4.14 CodeFontItalic . . . 300
7.62.4.15 CodeInlineBackgroundColour . . . 300
7.62.4.16 CodeInlineMargin . . . 300
7.62.4.17 ForegroundColour . . . 300
7.62.4.18 HeaderFontSizeMultipliers . . . 301
7.62.4.19 HeaderLineColour . . . 301
7.62.4.20 HeaderLineThicknesses . . . 301
7.62.4.21 ImageMarginTolerance . . . 301
7.62.4.22 ImageMultiplier . . . 302
7.62.4.23 ImageSideMargin . . . 302
7.62.4.24 ImageUnitMultiplier . . . 302
7.62.4.25 ImageUriResolver . . . 302
7.62.4.26 IndentWidth . . . 303
7.62.4.27 InsertedColour . . . 303
7.62.4.28 ItalicFontFamily . . . 303
7.62.4.29 LinkColour . . . 303
7.62.4.30 LinkUriResolver . . . 303
7.62.4.31 Margins . . . 304
7.62.4.32 MarkedColour . . . 304
7.62.4.33 MathFontScalingFactor . . . 304
7.62.4.34 PageSize . . . 304
7.62.4.35 QuoteBlockBackgroundColour . . . 304
7.62.4.36 QuoteBlockBarColour . . . 305
7.62.4.37 QuoteBlockBarWidth . . . 305

7.62.4.38 QuoteBlockIndentWidth . . . 305
7.62.4.39 RasterImageLoader . . . 305
7.62.4.40 RegularFontFamily . . . 305
7.62.4.41 SpaceAfterHeading . . . 306
7.62.4.42 SpaceAfterLine . . . 306
7.62.4.43 SpaceAfterParagraph . . . 306
7.62.4.44 SpaceBeforeHeading . . . 306
7.62.4.45 SpaceBeforeParagaph . . . 306
7.62.4.46 SubscriptShift . . . 307
7.62.4.47 SubSuperscriptFontSize . . . 307
7.62.4.48 SuperscriptShift . . . 307
7.62.4.49 SyntaxHighlighter . . . 307
7.62.4.50 TableCellMargins . . . 307
7.62.4.51 TableHeaderRowSeparatorColour . . . 308
7.62.4.52 TableHeaderRowSeparatorThickness . . . 308
7.62.4.53 TableHeaderSeparatorThickness . . . 308
7.62.4.54 TableRowSeparatorColour . . . 308
7.62.4.55 TableVAlign . . . 308
7.62.4.56 TaskListCheckedBullet . . . 309
7.62.4.57 TaskListUncheckedBullet . . . 309
7.62.4.58 ThematicBreakLineColour . . . 309
7.62.4.59 ThematicBreakThickness . . . 310
7.62.4.60 UnderlineThickness . . . 310
7.63 VectSharp.ThreeD.MaskedLightSource Class Reference . . . 310
7.63.1 Detailed Description . . . 311
7.63.2 Constructor & Destructor Documentation . . . 311
7.63.2.1 MaskedLightSource() [1/2] . . . 312
7.63.2.2 MaskedLightSource() [2/2] . . . 312
7.63.3 Member Function Documentation . . . 313
7.63.3.1 GetLightAt() . . . 313
7.63.3.2 GetObstruction() . . . 313
7.63.4 Property Documentation . . . 313
7.63.4.1 AngleAttenuationExponent . . . 314
7.63.4.2 CastsShadow . . . 314
7.63.4.3 Direction . . . 314
7.63.4.4 Distance . . . 314
7.63.4.5 DistanceAttenuationExponent . . . 314
7.63.4.6 Intensity . . . 315
7.63.4.7 Origin . . . 315
7.63.4.8 Position . . . 315
7.64 VectSharp.Filters.MaskFilter Class Reference . . . 315
7.64.1 Detailed Description . . . 316

7.64.2 Constructor & Destructor Documentation . . . 316
7.64.2.1 MaskFilter() . . . 316
7.64.3 Member Function Documentation . . . 316
7.64.3.1 Filter() . . . 317
7.64.4 Property Documentation . . . 317
7.64.4.1 BottomRightMargin . . . 317
7.64.4.2 Mask . . . 317
7.64.4.3 TopLeftMargin . . . 318
7.65 VectSharp.Fonts.Nimbus Class Reference . . . 318
7.65.1 Detailed Description . . . 318
7.65.2 Property Documentation . . . 318
7.65.2.1 Library . . . 318
7.66 VectSharp.ThreeD.ObjectFactory Class Reference . . . 319
7.66.1 Detailed Description . . . 319
7.66.2 Member Function Documentation . . . 319
7.66.2.1 CreateCube() . . . 320
7.66.2.2 CreateCuboid() . . . 320
7.66.2.3 CreatePoints() . . . 321
7.66.2.4 CreatePolygon() . . . 321
7.66.2.5 CreatePrism() . . . 322
7.66.2.6 CreateRectangle() [1/2] . . . 323
7.66.2.7 CreateRectangle() [2/2] . . . 323
7.66.2.8 CreateSphere() . . . 324
7.66.2.9 CreateTetrahedron() . . . 325
7.66.2.10 CreateWireframe() . . . 325
7.67 VectSharp.Page Class Reference . . . 326
7.67.1 Detailed Description . . . 326
7.67.2 Constructor & Destructor Documentation . . . 326
7.67.2.1 Page() . . . 326
7.67.3 Member Function Documentation . . . 327
7.67.3.1 Crop() [1/3] . . . 327
7.67.3.2 Crop() [2/3] . . . 327
7.67.3.3 Crop() [3/3] . . . 328
7.67.4 Property Documentation . . . 328
7.67.4.1 Background . . . 328
7.67.4.2 Graphics . . . 328
7.67.4.3 Height . . . 328
7.67.4.4 Width . . . 329
7.68 VectSharp.TrueTypeFile.PairKerning Class Reference . . . 329
7.68.1 Detailed Description . . . 329
7.68.2 Property Documentation . . . 329
7.68.2.1 Glyph1Advance . . . 329

7.68.2.2 Glyph1Placement . . . 330
7.68.2.3 Glyph2Advance . . . 330
7.68.2.4 Glyph2Placement . . . 330
7.69 VectSharp.ThreeD.ParallelLightSource Class Reference . . . 330
7.69.1 Detailed Description . . . 331
7.69.2 Constructor & Destructor Documentation . . . 331
7.69.2.1 ParallelLightSource() . . . 331
7.69.3 Member Function Documentation . . . 332
7.69.3.1 GetLightAt() . . . 332
7.69.3.2 GetObstruction() . . . 332
7.69.4 Property Documentation . . . 333
7.69.4.1 CastsShadow . . . 333
7.69.4.2 Direction . . . 333
7.69.4.3 Intensity . . . 333
7.69.4.4 ReverseDirection . . . 334
7.70 VectSharp.SVG.Parser Class Reference . . . 334
7.70.1 Detailed Description . . . 334
7.70.2 Member Function Documentation . . . 334
7.70.2.1 FromFile() . . . 334
7.70.2.2 FromStream() . . . 335
7.70.2.3 FromString() . . . 335
7.70.2.4 ParseSVGURI() . . . 336
7.70.3 Member Data Documentation . . . 336
7.70.3.1 ParseImageURI . . . 336
7.71 VectSharp.PDF.PDFContextInterpreter Class Reference . . . 336
7.71.1 Detailed Description . . . 337
7.71.2 Member Enumeration Documentation . . . 337
7.71.2.1 TextOptions . . . 337
7.71.3 Member Function Documentation . . . 337
7.71.3.1 SaveAsPDF() [1/2] . . . 337
7.71.3.2 SaveAsPDF() [2/2] . . . 338
7.72 VectSharp.ThreeD.PhongMaterial Class Reference . . . 338
7.72.1 Detailed Description . . . 339
7.72.2 Constructor & Destructor Documentation . . . 340
7.72.2.1 PhongMaterial() . . . 340
7.72.3 Member Function Documentation . . . 340
7.72.3.1 GetColour() . . . 340
7.72.4 Property Documentation . . . 341
7.72.4.1 AmbientReflectionCoefficient . . . 341
7.72.4.2 Colour . . . 341
7.72.4.3 DiffuseReflectionCoefficient . . . 341
7.72.4.4 SpecularReflectionCoefficient . . . 341

7.72.4.5 SpecularShininess . . . 342
7.73 VectSharp.Point Struct Reference . . . 342
7.73.1 Detailed Description . . . 343
7.73.2 Constructor & Destructor Documentation . . . 343
7.73.2.1 Point() . . . 343
7.73.3 Member Function Documentation . . . 343
7.73.3.1 Bounds() [1/2] . . . 343
7.73.3.2 Bounds() [2/2] . . . 344
7.73.3.3 IsEqual() . . . 344
7.73.3.4 Max() . . . 344
7.73.3.5 Min() . . . 345
7.73.3.6 Modulus() . . . 345
7.73.3.7 Normalize() . . . 346
7.73.4 Member Data Documentation . . . 346
7.73.4.1 X . . . 346
7.73.4.2 Y . . . 346
7.74 VectSharp.ThreeD.PointLightSource Class Reference . . . 346
7.74.1 Detailed Description . . . 347
7.74.2 Constructor & Destructor Documentation . . . 347
7.74.2.1 PointLightSource() . . . 347
7.74.3 Member Function Documentation . . . 348
7.74.3.1 GetLightAt() . . . 348
7.74.3.2 GetObstruction() . . . 348
7.74.4 Property Documentation . . . 349
7.74.4.1 CastsShadow . . . 349
7.74.4.2 DistanceAttenuationExponent . . . 349
7.74.4.3 Intensity . . . 349
7.74.4.4 Position . . . 350
7.75 VectSharp.RadialGradientBrush Class Reference . . . 350
7.75.1 Detailed Description . . . 351
7.75.2 Constructor & Destructor Documentation . . . 351
7.75.2.1 RadialGradientBrush() [1/2] . . . 351
7.75.2.2 RadialGradientBrush() [2/2] . . . 352
7.75.3 Member Function Documentation . . . 352
7.75.3.1 MultiplyOpacity() . . . 352
7.75.4 Property Documentation . . . 352
7.75.4.1 Centre . . . 353
7.75.4.2 FocalPoint . . . 353
7.75.4.3 Radius . . . 353
7.76 VectSharp.TrueTypeFile.CoverageTable.RangeRecord Struct Reference . . . 353
7.76.1 Detailed Description . . . 353
7.76.2 Constructor & Destructor Documentation . . . 354

7.76.2.1 RangeRecord() . . . 354
7.76.3 Member Data Documentation . . . 354
7.76.3.1 EndGlyphID . . . 354
7.76.3.2 StartCoverageIndex . . . 354
7.76.3.3 StartGlyphID . . . 354
7.77 VectSharp.Raster.Raster Class Reference . . . 354
7.77.1 Detailed Description . . . 355
7.77.2 Member Function Documentation . . . 355
7.77.2.1 Rasterise() . . . 355
7.77.2.2 SaveAsAnimatedPNG() [1/2] . . . 356
7.77.2.3 SaveAsAnimatedPNG() [2/2] . . . 356
7.77.2.4 SaveAsPNG() [1/2] . . . 357
7.77.2.5 SaveAsPNG() [2/2] . . . 357
7.77.2.6 SaveAsRawBytes() . . . 358
7.78 VectSharp.RasterImage Class Reference . . . 358
7.78.1 Detailed Description . . . 360
7.78.2 Constructor & Destructor Documentation . . . 360
7.78.2.1 RasterImage() [1/3] . . . 360
7.78.2.2 RasterImage() [2/3] . . . 360
7.78.2.3 RasterImage() [3/3] . . . 361
7.78.3 Member Function Documentation . . . 361
7.78.3.1 ClearPNGCache() . . . 361
7.78.3.2 Dispose() . . . 362
7.78.4 Property Documentation . . . 362
7.78.4.1 DataHolder . . . 362
7.78.4.2 HasAlpha . . . 362
7.78.4.3 Height . . . 362
7.78.4.4 Id . . . 362
7.78.4.5 ImageDataAddress . . . 363
7.78.4.6 Interpolate . . . 363
7.78.4.7 PNGStream . . . 363
7.78.4.8 Width . . . 363
7.79 VectSharp.ImageSharpUtils.RasterImageFile Class Reference . . . 364
7.79.1 Detailed Description . . . 364
7.79.2 Constructor & Destructor Documentation . . . 364
7.79.2.1 RasterImageFile() . . . 364
7.80 VectSharp.MuPDFUtils.RasterImageFile Class Reference . . . 365
7.80.1 Detailed Description . . . 365
7.80.2 Constructor & Destructor Documentation . . . 366
7.80.2.1 RasterImageFile() . . . 366
7.81 VectSharp.ImageSharpUtils.RasterImageStream Class Reference . . . 366
7.81.1 Detailed Description . . . 367

7.81.2 Constructor & Destructor Documentation . . . 367
7.81.2.1 RasterImageStream() [1/2] . . . 367
7.81.2.2 RasterImageStream() [2/2] . . . 368
7.82 VectSharp.MuPDFUtils.RasterImageStream Class Reference . . . 368
7.82.1 Detailed Description . . . 369
7.82.2 Constructor & Destructor Documentation . . . 369
7.82.2.1 RasterImageStream() [1/2] . . . 369
7.82.2.2 RasterImageStream() [2/2] . . . 370
7.83 VectSharp.Rectangle Struct Reference . . . 370
7.83.1 Detailed Description . . . 371
7.83.2 Constructor & Destructor Documentation . . . 372
7.83.2.1 Rectangle() [1/3] . . . 372
7.83.2.2 Rectangle() [2/3] . . . 373
7.83.2.3 Rectangle() [3/3] . . . 373
7.83.3 Member Function Documentation . . . 373
7.83.3.1 Intersection() . . . 374
7.83.3.2 IsNaN() . . . 374
7.83.3.3 Union() [1/3] . . . 374
7.83.3.4 Union() [2/3] . . . 375
7.83.3.5 Union() [3/3] . . . 375
7.83.4 Member Data Documentation . . . 375
7.83.4.1 Location . . . 376
7.83.4.2 NaN . . . 376
7.83.4.3 Size . . . 376
7.83.5 Property Documentation . . . 376
7.83.5.1 Centre . . . 376
7.84 VectSharp.Canvas.RenderAction Class Reference . . . 376
7.84.1 Detailed Description . . . 378
7.84.2 Member Enumeration Documentation . . . 378
7.84.2.1 ActionTypes . . . 378
7.84.3 Member Function Documentation . . . 378
7.84.3.1 BringToFront() . . . 379
7.84.3.2 ImageAction() . . . 379
7.84.3.3 PathAction() . . . 379
7.84.3.4 SendToBack() . . . 380
7.84.3.5 TextAction() . . . 380
7.84.4 Property Documentation . . . 381
7.84.4.1 ActionType . . . 381
7.84.4.2 ClippingPath . . . 381
7.84.4.3 Fill . . . 381
7.84.4.4 Geometry . . . 381
7.84.4.5 ImageDestination . . . 382

7.84.4.6 ImageId . . . 382
7.84.4.7 ImageSource . . . 382
7.84.4.8 InverseTransform . . . 382
7.84.4.9 Parent . . . 382
7.84.4.10 Stroke . . . 383
7.84.4.11 Tag . . . 383
7.84.4.12 Text . . . 383
7.84.4.13 Transform . . . 383
7.84.5 Event Documentation . . . 383
7.84.5.1 PointerEnter . . . 383
7.84.5.2 PointerLeave . . . 384
7.84.5.3 PointerPressed . . . 384
7.84.5.4 PointerReleased . . . 384
7.85 VectSharp.ResourceFontFamily Class Reference . . . 384
7.85.1 Detailed Description . . . 385
7.85.2 Constructor & Destructor Documentation . . . 385
7.85.2.1 ResourceFontFamily() . . . 385
7.85.3 Member Data Documentation . . . 385
7.85.3.1 ResourceName . . . 386
7.86 VectSharp.ThreeD.Scene Class Reference . . . 386
7.86.1 Detailed Description . . . 387
7.86.2 Constructor & Destructor Documentation . . . 387
7.86.2.1 Scene() . . . 387
7.86.3 Member Function Documentation . . . 387
7.86.3.1 AddElement() . . . 387
7.86.3.2 AddRange() . . . 388
7.86.3.3 Replace() [1/2] . . . 388
7.86.3.4 Replace() [2/2] . . . 388
7.86.4 Property Documentation . . . 389
7.86.4.1 SceneElements . . . 389
7.86.4.2 SceneLock . . . 389
7.87 VectSharp.Segment Class Reference . . . 389
7.87.1 Detailed Description . . . 390
7.87.2 Member Function Documentation . . . 390
7.87.2.1 Clone() . . . 390
7.87.2.2 GetLinearisationTangents() . . . 390
7.87.2.3 GetPointAt() . . . 391
7.87.2.4 GetTangentAt() . . . 391
7.87.2.5 Linearise() . . . 392
7.87.2.6 Measure() . . . 392
7.87.2.7 Transform() . . . 392
7.87.3 Property Documentation . . . 393

7.87.3.1 Point . . . 393
7.87.3.2 Points . . . 393
7.87.3.3 Type . . . 393
7.88 VectSharp.SimpleFontLibrary Class Reference . . . 394
7.88.1 Detailed Description . . . 395
7.88.2 Constructor & Destructor Documentation . . . 395
7.88.2.1 SimpleFontLibrary() [1/4] . . . 395
7.88.2.2 SimpleFontLibrary() [2/4] . . . 395
7.88.2.3 SimpleFontLibrary() [3/4] . . . 396
7.88.2.4 SimpleFontLibrary() [4/4] . . . 396
7.88.3 Member Function Documentation . . . 397
7.88.3.1 Add() [1/4] . . . 397
7.88.3.2 Add() [2/4] . . . 398
7.88.3.3 Add() [3/4] . . . 398
7.88.3.4 Add() [4/4] . . . 398
7.88.3.5 ResolveFontFamily() [1/2] . . . 399
7.88.3.6 ResolveFontFamily() [2/2] . . . 399
7.89 VectSharp.Size Struct Reference . . . 399
7.89.1 Detailed Description . . . 400
7.89.2 Constructor & Destructor Documentation . . . 400
7.89.2.1 Size() . . . 400
7.89.3 Member Data Documentation . . . 400
7.89.3.1 Height . . . 400
7.89.3.2 Width . . . 401
7.90 VectSharp.Canvas.SKMultiLayerRenderCanvas Class Reference . . . 401
7.90.1 Detailed Description . . . 402
7.90.2 Constructor & Destructor Documentation . . . 403
7.90.2.1 SKMultiLayerRenderCanvas() [1/3] . . . 403
7.90.2.2 SKMultiLayerRenderCanvas() [2/3] . . . 403
7.90.2.3 SKMultiLayerRenderCanvas() [3/3] . . . 404
7.90.3 Member Function Documentation . . . 404
7.90.3.1 AddLayer() . . . 404
7.90.3.2 Dispose() . . . 405
7.90.3.3 InsertLayer() . . . 405
7.90.3.4 InvalidateDirty() . . . 405
7.90.3.5 InvalidateZIndex() . . . 405
7.90.3.6 MoveLayer() . . . 405
7.90.3.7 RemoveLayer() . . . 406
7.90.3.8 Render() . . . 406
7.90.3.9 RenderAtResolution() . . . 406
7.90.3.10 SwitchLayers() . . . 407
7.90.3.11 UpdateLayer() . . . 407

7.90.3.12 UpdateWith() . . . 407
7.90.4 Member Data Documentation . . . 408
7.90.4.1 LayerTransforms . . . 408
7.90.4.2 RenderActions . . . 408
7.90.4.3 RenderLock . . . 408
7.90.5 Property Documentation . . . 409
7.90.5.1 PageHeight . . . 409
7.90.5.2 PageWidth . . . 409
7.90.5.3 Spinner . . . 409
7.91 VectSharp.Canvas.SKRenderAction Class Reference . . . 409
7.91.1 Detailed Description . . . 411
7.91.2 Member Enumeration Documentation . . . 412
7.91.2.1 ActionTypes . . . 412
7.91.3 Member Function Documentation . . . 412
7.91.3.1 ClipAction() . . . 412
7.91.3.2 Dispose() . . . 412
7.91.3.3 DrawFilteredGraphicsAction() . . . 413
7.91.3.4 ImageAction() . . . 413
7.91.3.5 InvalidateAll() . . . 413
7.91.3.6 InvalidateHitTestPath() . . . 414
7.91.3.7 InvalidateVisual() . . . 414
7.91.3.8 InvalidateZIndex() . . . 415
7.91.3.9 PathAction() . . . 415
7.91.3.10 RestoreAction() . . . 415
7.91.3.11 SaveAction() . . . 416
7.91.3.12 TextAction() . . . 416
7.91.3.13 TransformAction() . . . 417
7.91.4 Property Documentation . . . 417
7.91.4.1 ActionType . . . 417
7.91.4.2 Disposed . . . 417
7.91.4.3 Filter . . . 418
7.91.4.4 Font . . . 418
7.91.4.5 Graphics . . . 418
7.91.4.6 ImageDestination . . . 418
7.91.4.7 ImageId . . . 418
7.91.4.8 ImageSource . . . 419
7.91.4.9 Paint . . . 419
7.91.4.10 Parent . . . 419
7.91.4.11 Path . . . 419
7.91.4.12 Payload . . . 419
7.91.4.13 Tag . . . 420
7.91.4.14 Text . . . 420

7.91.4.15 TextX . . . 420
7.91.4.16 TextY . . . 420
7.91.4.17 Transform . . . 420
7.91.4.18 ZIndex . . . 421
7.91.5 Event Documentation . . . 421
7.91.5.1 PointerEnter . . . 421
7.91.5.2 PointerLeave . . . 421
7.91.5.3 PointerPressed . . . 421
7.91.5.4 PointerReleased . . . 422
7.92 VectSharp.Canvas.SKRenderContext Class Reference . . . 422
7.92.1 Detailed Description . . . 422
7.93 VectSharp.Canvas.SKRenderContextInterpreter Class Reference . . . 422
7.93.1 Detailed Description . . . 423
7.93.2 Member Function Documentation . . . 423
7.93.2.1 CopyToSKRenderContext() [1/3] . . . 424
7.93.2.2 CopyToSKRenderContext() [2/3] . . . 424
7.93.2.3 CopyToSKRenderContext() [3/3] . . . 425
7.93.2.4 PaintToSKCanvas() [1/6] . . . 426
7.93.2.5 PaintToSKCanvas() [2/6] . . . 426
7.93.2.6 PaintToSKCanvas() [3/6] . . . 427
7.93.2.7 PaintToSKCanvas() [4/6] . . . 428
7.93.2.8 PaintToSKCanvas() [5/6] . . . 429
7.93.2.9 PaintToSKCanvas() [6/6] . . . 429
7.93.2.10 Rasterise() . . . 430
7.94 VectSharp.SolidColourBrush Class Reference . . . 431
7.94.1 Detailed Description . . . 432
7.94.2 Constructor & Destructor Documentation . . . 432
7.94.2.1 SolidColourBrush() . . . 432
7.94.3 Member Function Documentation . . . 432
7.94.3.1 MultiplyOpacity() . . . 432
7.94.3.2 operator SolidColourBrush() . . . 433
7.94.4 Property Documentation . . . 433
7.94.4.1 A . . . 433
7.94.4.2 B . . . 433
7.94.4.3 Colour . . . 434
7.94.4.4 G . . . 434
7.94.4.5 R . . . 434
7.95 VectSharp.SplineEasing Class Reference . . . 434
7.95.1 Detailed Description . . . 435
7.95.2 Constructor & Destructor Documentation . . . 435
7.95.2.1 SplineEasing() . . . 435
7.95.3 Member Function Documentation . . . 435

7.95.3.1 Ease() . . . 436
7.95.4 Property Documentation . . . 436
7.95.4.1 ControlPoint1 . . . 436
7.95.4.2 ControlPoint2 . . . 436
7.96 VectSharp.ThreeD.SpotlightLightSource Class Reference . . . 437
7.96.1 Detailed Description . . . 438
7.96.2 Constructor & Destructor Documentation . . . 438
7.96.2.1 SpotlightLightSource() . . . 438
7.96.3 Member Function Documentation . . . 439
7.96.3.1 GetLightAt() . . . 439
7.96.3.2 GetObstruction() . . . 439
7.96.4 Property Documentation . . . 439
7.96.4.1 AngleAttenuationExponent . . . 440
7.96.4.2 BeamWidthAngle . . . 440
7.96.4.3 CastsShadow . . . 440
7.96.4.4 CutoffAngle . . . 440
7.96.4.5 Direction . . . 440
7.96.4.6 DistanceAttenuationExponent . . . 441
7.96.4.7 Intensity . . . 441
7.96.4.8 Position . . . 441
7.97 VectSharp.SVG.SVGContextInterpreter Class Reference . . . 441
7.97.1 Detailed Description . . . 442
7.97.2 Member Enumeration Documentation . . . 442
7.97.2.1 TextOptions . . . 442
7.97.3 Member Function Documentation . . . 443
7.97.3.1 SaveAsAnimatedSVG() [1/3] . . . 443
7.97.3.2 SaveAsAnimatedSVG() [2/3] . . . 443
7.97.3.3 SaveAsAnimatedSVG() [3/3] . . . 444
7.97.3.4 SaveAsAnimatedSVGWithFrames() [1/3] . . . 445
7.97.3.5 SaveAsAnimatedSVGWithFrames() [2/3] . . . 445
7.97.3.6 SaveAsAnimatedSVGWithFrames() [3/3] . . . 446
7.97.3.7 SaveAsSVG() [1/3] . . . 447
7.97.3.8 SaveAsSVG() [2/3] . . . 447
7.97.3.9 SaveAsSVG() [3/3] . . . 448
7.98 VectSharp.Markdown.SyntaxHighlighter Class Reference . . . 448
7.98.1 Detailed Description . . . 449
7.98.2 Member Function Documentation . . . 449
7.98.2.1 GetSyntaxHighlightedLines() . . . 449
7.99 VectSharp.Transition Class Reference . . . 449
7.99.1 Detailed Description . . . 450
7.99.2 Constructor & Destructor Documentation . . . 450
7.99.2.1 Transition() . . . 450

7.99.3 Property Documentation . . . 450
7.99.3.1 Duration . . . 450
7.99.3.2 Easings . . . 451
7.99.3.3 OverallEasing . . . 451
7.100 VectSharp.TrueTypeFile Class Reference . . . 451
7.100.1 Detailed Description . . . 453
7.100.2 Member Function Documentation . . . 453
7.100.2.1 Destroy() . . . 453
7.100.2.2 Get1000EmAscent() . . . 454
7.100.2.3 Get1000EmDescent() . . . 454
7.100.2.4 Get1000EmGlyphBearings() . . . 454
7.100.2.5 Get1000EmGlyphVerticalMetrics() . . . 455
7.100.2.6 Get1000EmGlyphWidth() [1/2] . . . 455
7.100.2.7 Get1000EmGlyphWidth() [2/2] . . . 455
7.100.2.8 Get1000EmKerning() [1/2] . . . 456
7.100.2.9 Get1000EmKerning() [2/2] . . . 456
7.100.2.10 Get1000EmUnderlineIntersections() . . . 457
7.100.2.11 Get1000EmUnderlinePosition() . . . 457
7.100.2.12 Get1000EmUnderlineThickness() . . . 457
7.100.2.13 Get1000EmWinAscent() . . . 458
7.100.2.14 Get1000EmXMax() . . . 458
7.100.2.15 Get1000EmXMin() . . . 458
7.100.2.16 Get1000EmYMax() . . . 458
7.100.2.17 Get1000EmYMin() . . . 459
7.100.2.18 GetFirstCharIndex() . . . 459
7.100.2.19 GetFontFamilyName() . . . 459
7.100.2.20 GetFontName() . . . 459
7.100.2.21 GetFullFontFamilyName() . . . 460
7.100.2.22 GetGlyphIndex() . . . 460
7.100.2.23 GetGlyphPath() [1/2] . . . 460
7.100.2.24 GetGlyphPath() [2/2] . . . 461
7.100.2.25 GetItalicAngle() . . . 461
7.100.2.26 GetLastCharIndex() . . . 461
7.100.2.27 IsBold() . . . 462
7.100.2.28 IsFixedPitch() . . . 462
7.100.2.29 IsItalic() . . . 462
7.100.2.30 IsOblique() . . . 462
7.100.2.31 IsScript() . . . 463
7.100.2.32 IsSerif() . . . 463
7.100.2.33 SubsetFont() . . . 463
7.100.3 Property Documentation . . . 464
7.100.3.1 FontStream . . . 464

7.101 VectSharp.TrueTypeFile.TrueTypePoint Struct Reference . . . 464
7.101.1 Detailed Description . . . 464
7.101.2 Member Data Documentation . . . 464
7.101.2.1 IsOnCurve . . . 464
7.101.2.2 X . . . 465
7.101.2.3 Y . . . 465
7.102 VectSharp.UnbalancedStackException Class Reference . . . 465
7.102.1 Detailed Description . . . 465
7.103 VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.UnknownFormatException Class Reference . . . 466
7.103.1 Detailed Description . . . 466
7.103.2 Property Documentation . . . 466
7.103.2.1 Format . . . 466
7.104 VectSharp.TrueTypeFile.VerticalMetrics Struct Reference . . . 467
7.104.1 Detailed Description . . . 467
7.104.2 Member Data Documentation . . . 467
7.104.2.1 YMax . . . 467
7.104.2.2 YMin . . . 467

8 File Documentation **469**
8.1 AnimatedCanvas.cs . . . 469
8.2 AvaloniaContext.cs . . . 473
8.3 RenderingParameters.cs . . . 507
8.4 SKMultiLayerRenderCanvas.cs . . . 509
8.5 SKRenderContext.cs . . . 524
8.6 Nimbus.cs . . . 548
8.7 HtmlTag.cs . . . 549
8.8 Line.cs . . . 554
8.9 MarkdownContext.cs . . . 556
8.10 MarkdownRenderer.cs . . . 559
8.11 SyntaxHighlighting.cs . . . 592
8.12 Word.cs . . . 596
8.13 ImageCache.cs . . . 598
8.14 MarkdownCanvas.axaml.cs . . . 600
8.15 ImageUriParser.cs . . . 604
8.16 ImageURIParser.cs . . . 605
8.17 RasterImages.cs . . . 607
8.18 RasterImages.cs . . . 610
8.19 PDFContext.cs . . . 613
8.20 GradientBrushApplicator.cs . . . 651
8.21 ImageSharpContext.cs . . . 654
8.22 Raster.cs . . . 674
8.23 SVGContext.cs . . . 678

8.24 SVGParser.cs 724
8.25 Lights.cs 761
8.26 Materials.cs 772
8.27 ObjectFactory.cs 774
8.28 Scene.cs 781
8.29 Animation.cs 782
8.30 Brush.cs 804
8.31 Colour.cs 810
8.32 Document.cs 818
8.33 Enums.cs 820
8.34 BoxBlurFilter.cs 822
8.35 ColourMatrixFilter.cs 827
8.36 CompositeFilter.cs 836
8.37 ConvolutionFilter.cs 838
8.38 Filters.cs 842
8.39 GaussianBlurFilter.cs 845
8.40 MaskFilter.cs 849
8.41 Font.cs 851
8.42 FontLibrary.cs 860
8.43 FormattedText.cs 868
8.44 Graphics.cs 877
8.45 Graphics.Text.cs 899
8.46 GraphicsAction.cs 912
8.47 GraphicsPath.cs 915
8.48 ImageFormats.cs 959
8.49 Point.cs 973
8.50 RasterImage.cs 977
8.51 Segment.cs 981
8.52 SmoothSpline.cs 990
8.53 StandardColours.cs 992
8.54 TrueType.cs 1001

Index **1061**

Chapter 1

VectSharp: a light library for C# vector graphics

1.1 Introduction

VectSharp is a library to create vector graphics (including text) in C#, without too many dependencies.

VectSharp is written using .NET Core, and is available for Mac, Windows and Linux. Since version 2.0.0, it is released under an LGPLv3 license. It includes 14 standard fonts, originally released under an ASL-2.0 license.

It includes an abstract layer on top of which output layers can be written. Currently, there are five available output layers:

- **VectSharp.PDF** produces PDF documents.
- **VectSharp.Canvas** produces an `Avalonia.Controls.Canvas` object (`https://avaloniaui.net/docs/controls/canvas`) containing the rendered graphics objects.
- **VectSharp.SVG** produces vector graphics in SVG format.
- **VectSharp.Raster** produces raster images in PNG format, (this is done by rendering the image to a PDF document, and then using the `MuPDFCore` library to render the PDF). Since version 2.0.0, VectSharp.Raster is released under an AGPLv3 licence.
- **VectSharp.Raster.ImageSharp** produces raster images in multiple formats (BMP, GIF, JPEG, PBM, PNG, TGA, TIFF, WebP) using the `SixLabors.ImageSharp` library.

VectSharp.Raster and VectSharp.Raster.ImageSharp are somewhat overlapping, as both of them can be used to create PNG images. However, VectSharp.Raster is much faster, though it only supports the PNG format. Instead, VectSharp.Raster.ImageSharp is slower, but supports more formats and has a more permissive licence. Another difference is that VectSharp.Raster carries a native dependency (through MuPDFCore), while VectSharp.ImageSharp does not.

Furthermore:

- **`VectSharp.ThreeD`** adds support for 3D vector and raster graphics.

- **VectSharp.Markdown** can be used to transform Markdown documents into VectSharp objects, that can then be exported e.g. as PDF or SVG files, or displayed in an Avalonia `Canvas`. **VectSharp.MarkdownCanvas** uses VectSharp.Markdown to render Markdown documents in Avalonia applications (an example of this is in the `MarkdownViewerDemo project`).
- **VectSharp.MuPDFUtils**, released under an AGPLv3 licence, contains some utility functions that use `MuPDFCore` to make it possible to include in VectSharp graphics images in various formats.
- **VectSharp.ImageSharpUtils** adds the same capabilities as VectSharp.MuPDFUtils, using ImageSharp instead of MuPDFCore; as a result, it is released under a more permissive LGPLv3 licence.
- **VectSharp.Fonts.Nimbus** is a package released under a GPLv3 license, which contains the standard fonts that were used in VectSharp before version 2.0.0. Since these fonts are released under a GPL license, they had to be replaced when the VectSharp license changed to LGPL. See the [Font libraries](section) below for information on how to re-enable these fonts.
- The **Animation** class (provided in the base VectSharp package) can be used to create animations that can be saved as animated GIFs (using VectSharp.Raster.ImageSharp), SVGs (using VectSharp.SVG) and PNGs (using VectSharp.Raster or VectSharp.Raster.ImageSharp).

1.2 Installing VectSharp

To include VectSharp in your project, you will need one of the output layer NuGet packages: `VectSharp.PDF`, `VectSharp.Canvas`, `VectSharp.Raster`, `VectSharp.Raster.ImageSharp`, or `VectSharp.SVG`. You will need `VectSharp.ThreeD` to work with 3D graphics. You may want the `VectSharp.MuPDFUtils` package if you wish to manipulate raster images, and the `VectSharp.Fonts.Nimbus` if you want to restore the GPL-licensed fonts used in previous versions of the library.

1.3 Usage

You can find detailed documentation for the VectSharp library, **including interactive examples**, at the `documentation website`. A comprehensive API reference is also available, both as a `website` and as a `PDF manual`.

In general, working with VectSharp involves: creating a `Document`, adding `Page`s, drawing to the `Page`s' `Graphics` objects and, finally, exporting them to a PDF document, `Canvas`, PNG image or SVG document.

- Create a `Document`:
```
using VectSharp;
// ...
Document doc = new Document();
```
- Add a `Page`:
```
doc.Pages.Add(new Page(1000, 1000));
```
- Draw to the `Page`'s `Graphics` object:
```
Graphics gpr = doc.Pages.Last().Graphics;
gpr.FillRectangle(100, 100, 800, 800, Colour.FromRgb(128, 128, 128));
```
- Save as PDF document:
```
using VectSharp.PDF;
//...
doc.SaveAsPDF(@"Sample.pdf");
```
- Export the graphics to a `Canvas`:
```
using VectSharp.Canvas;
//...
Avalonia.Controls.Canvas can = doc.Pages.Last().PaintToCanvas();
```

- Export the graphics to a `Canvas`, using a multi-layer, multi-threaded, triple-buffered renderer based on SkiaSharp (which provides the best performance if you wish e.g. to place the canvas within a `ZoomBorder`):
```
using VectSharp.Canvas;
//...
// A single page
Avalonia.Controls.Canvas can = doc.Pages.Last().PaintToSKCanvas();
// The whole document - each page will correspond to a layer
Avalonia.Controls.Canvas can = doc.PaintToSKCanvas();
```
- Save as a PNG image:
```
using VectSharp.Raster;
//...
doc.Pages.Last().SaveAsPNG(@"Sample.png");
```
- Save as a JPEG image:
```
using VectSharp.Raster.ImageSharp;
//...
doc.Pages.Last().SaveAsImage(@"Sample.jpg");
```
- Save as an SVG document:
```
using VectSharp.SVG;
//...
doc.Pages.Last().SaveAsSVG(@"Sample.svg");
```
- PDF and SVG documents support both internal and external links:
```
using VectSharp;
using VectSharp.PDF;
using VectSharp.SVG;
//...
Document document = new Document();
Page page = new Page(1000, 1000);
document.Pages.Add(page);
page.Graphics.FillRectangle(100, 100, 800, 50, Colour.FromRgb(128, 128, 128), tag: "linkToGitHub");
page.Graphics.FillRectangle(100, 300, 800, 50, Colour.FromRgb(255, 0, 0), tag: "linkToBlueRectangle");
page.Graphics.FillRectangle(100, 850, 800, 50, Colour.FromRgb(0, 0, 255), tag: "blueRectangle");
Dictionary<string, string> links = new Dictionary<string, string>() { { "linkToGitHub",
 "https://github.com/" }, { "linkToBlueRectangle", "#blueRectangle" } };
page.SaveAsSVG(@"Links.svg", linkDestinations: links);
document.SaveAsPDF(@"Links.pdf", linkDestinations: links);
```

  This code produces a document with three rectangles: the grey one at the top links to the GitHub home page, while the red one in the middle is a hyperlink to the blue one at the bottom. Links in PDF documents can refer to objects that are in a different page than the one containing the link.

The public classes and methods are `fully documented` (with interactive examples created using Blazor), and you can find a (much) more detailed code example in `MainWindow.xaml.cs`. A detailed guide about 3D graphics in VectSharp.ThreeD is available in the `VectSharp.ThreeD` folder. Further example code for animations is available in the `DemoAnimation project`.

1.4 Font libraries

Since version 2.0.0, font names are resolved using a "font library". This is a class that implements the `VectSharp.IFontLibrary` interface, providing methods to obtain a `FontFamily` object from a `string` or a `FontFamily.StandardFontFamilies` enumeration. The default font library included in VectSharp uses the embedded fonts (Arimo, Tinos, Cousine) as the standard font families.

In practice, assuming you want to use the default font library, you have the following options to create a `FontFamily` object:
```
using VectSharp;
// ...
FontFamily helvetica = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica); // Will
       resolve to the Arimo font family.
FontFamily times = FontFamily.ResolveFontFamily("Times-Roman"); // Will resolve to the Tinos font family.
```

These replace the `FontFamily(string)` and `FontFamily(StandardFontFamilies)` constructors of previous versions of VectSharp. Overloads of this method let you specify a list of "fallback" fonts that will be used if the first font you specify is not available.

If you wish, you can replace the default font library with a different one; this will change the way font families are resolved. For example, after installing the `VectSharp.Fonts.Nimbus` NuGet package, you can do:

```
using VectSharp;
// ...
FontFamily.DefaultFontLibrary = VectSharp.Fonts.Nimbus.Library;
FontFamily helvetica = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica); // Will
        resolve to the Nimbus Sans L font family.
FontFamily times = FontFamily.ResolveFontFamily("Times-Roman"); // Will resolve to the Nimbus Roman No 9 L
        font family.
```

This will let you re-enable the fonts that were used in previous versions of VectSharp.

You can also use multiple font libraries in the same project. Again, assuming you have installed the VectSharp.Fonts.Nimbus NuGet package:

```
using VectSharp;
FontFamily helvetica1 = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica); // Will
        resolve to the Arimo font family.
FontFamily times1 = FontFamily.ResolveFontFamily("Times-Roman"); // Will resolve to the Tinos font family.
FontFamily helvetica2 = VectSharp.Fonts.Nimbus.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica);
        // Will resolve to the Nimbus Sans L font family.
FontFamily times2 = VectSharp.Fonts.Nimbus.ResolveFontFamily("Times-Roman"); // Will resolve to the Nimbus
        Roman No 9 L font family.
```

Finally, you can create your own font library class (which could implement things such as dowloading fonts from Google Fonts, or finding them in the user's system font directory...) by creating a class that implements the `IFont↩Library` interface or that extends the `FontLibrary` class (in this latter case, you get a default implementation for the `ResolveFontFamily` overloads that use a list of fallback fonts).

1.5 Creating new output layers

VectSharp can be easily extended to provide additional output layers. To do so:

1. Create a new class implementing the `IGraphicsContext` interface.
2. Provide an extension method to either the `Page` or `Document` types.
3. Somewhere in the extension method, call the `CopyToIGraphicsContext` method on the `Graphics` object of the `Page`s.
4. Opportunely save or return the rendered result.

1.6 Compiling VectSharp from source

The VectSharp source code includes an example project (*VectSharp.Demo*) presenting how VectSharp can be used to produce graphics.

To be able to compile VectSharp from source, you will need to install the latest `.NET SDK` for your operating system.

You can use `Microsoft Visual Studio` to compile the program. The following instructions will cover compiling VectSharp from the command line, instead.

First of all, you will need to download the VectSharp source code: `VectSharp.tar.gz` and extract it somewhere.

1.6.1 Windows

Open a command-line window in the folder where you have extracted the source code, and type:

```
BuildDemo <Target>
```

Where `<Target>` can be one of `Win-x64`, `Linux-x64` or `Mac-x64` depending on which platform you wish to generate executables for.

In the Release folder and in the appropriate subfolder for the target platform you selected, you will find the compiled program.

1.6.2 macOS and Linux

Open a terminal in the folder where you have extracted the source code, and type:

```
./BuildDemo.sh <Target>
```

Where `<Target>` can be one of `Win-x64`, `Linux-x64` or `Mac-x64` depending on which platform you wish to generate executables for.

In the Release folder and in the appropriate subfolder for the target platform you selected, you will find the compiled program.

If you receive an error about permissions being denied, try typing `chmod +x BuildDemo.sh` first.

1.7 Note about VectSharp.MuPDFUtils and .NET Framework

If you wish to use VectSharp.MuPDFUtils in a .NET Framework project, you will need to manually copy the native MuPDFWrapper library for the platform you are using to the executable directory (this is done automatically if you target .NET core).

One way to obtain the appropriate library files is:

1. Manually download the NuGet package for `MuPFDCore` (click on the "Download package" link on the right).
2. Rename the `.nupkg` file so that it has a `.zip` extension.
3. Extract the zip file.
4. Within the extracted folder, the library files are in the `runtimes/xxx-yyy/native/` folder, where `xxx` is either `linux`, `osx` or `win`, depending on the platform you are using, and `yyy` is `x64`, `x86` or `arm64` depending on the architecture.

Make sure you copy the appropriate file to the same folder as the executable!

Chapter 2

Namespace Index

2.1 Package List

Here are the packages with brief descriptions (if available):

VectSharp 21
VectSharp.Canvas 25
VectSharp.Filters 25
VectSharp.Fonts 26
VectSharp.ImageSharpUtils 26
VectSharp.Markdown 26
VectSharp.MarkdownCanvas 26
VectSharp.MuPDFUtils 27
VectSharp.PDF 27
VectSharp.Raster 27
VectSharp.Raster.ImageSharp 27
VectSharp.SVG 28
VectSharp.ThreeD 28

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

VectSharp.AnimatedPNG . . . 37
VectSharp.Animation . . . 38
VectSharp.Canvas.AvaloniaContextInterpreter . . . 48
VectSharp.TrueTypeFile.Bearings . . . 53
VectSharp.Brush . . . 57
  VectSharp.GradientBrush . . . 186
    VectSharp.LinearGradientBrush . . . 278
    VectSharp.RadialGradientBrush . . . 350
  VectSharp.SolidColourBrush . . . 431
Avalonia.Controls.Canvas
  VectSharp.Canvas.AnimatedCanvas . . . 33
  VectSharp.Canvas.SKMultiLayerRenderCanvas . . . 401
VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord . . . 58
VectSharp.Filters.ColourMatrix . . . 75
VectSharp.Colours . . . 92
VectSharp.Font.DetailedFontMetrics . . . 140
VectSharp.Document . . . 144
Exception
  VectSharp.FontFamilyCreationException . . . 168
  VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.UnknownFormatException . . . 466
  VectSharp.UnbalancedStackException . . . 465
VectSharp.Canvas.FilterOption . . . 145
VectSharp.PDF.PDFContextInterpreter.FilterOption . . . 147
VectSharp.SVG.SVGContextInterpreter.FilterOption . . . 150
VectSharp.Font . . . 154
VectSharp.FontFamily . . . 161
  VectSharp.ResourceFontFamily . . . 384
VectSharp.Font.FontUnderline . . . 173
VectSharp.Markdown.FormattedString . . . 175
VectSharp.FormattedText . . . 177
VectSharp.FormattedTextExtensions . . . 181
VectSharp.Frame . . . 182
VectSharp.GradientStop . . . 187
VectSharp.Graphics . . . 192
VectSharp.GraphicsPath . . . 223

VectSharp.Markdown.HTTPUtils . . . 239
IDisposable
VectSharp.AnimatedPNG.CompressedFrame . . . 131
VectSharp.Canvas.SKMultiLayerRenderCanvas . . . 401
VectSharp.Canvas.SKRenderAction . . . 409
VectSharp.DisposableIntPtr . . . 142
VectSharp.Filters.FilterWithRasterisableParameter . . . 153
VectSharp.Filters.MaskFilter . . . 315
VectSharp.RasterImage . . . 358
VectSharp.ImageSharpUtils.RasterImageFile . . . 364
VectSharp.ImageSharpUtils.RasterImageStream . . . 366
VectSharp.MuPDFUtils.RasterImageFile . . . 365
VectSharp.MuPDFUtils.RasterImageStream . . . 368
VectSharp.IEasing . . . 241
VectSharp.LinearEasing . . . 277
VectSharp.SplineEasing . . . 434
IEquatable
VectSharp.Colour . . . 60
VectSharp.Filters.IFilter . . . 242
VectSharp.Filters.IFilterWithLocation . . . 243
VectSharp.Filters.MaskFilter . . . 315
VectSharp.Filters.ILocationInvariantFilter . . . 260
VectSharp.Filters.BoxBlurFilter . . . 54
VectSharp.Filters.ColourMatrixFilter . . . 89
VectSharp.Filters.CompositeLocationInvariantFilter . . . 128
VectSharp.Filters.ConvolutionFilter . . . 134
VectSharp.Filters.GaussianBlurFilter . . . 183
VectSharp.Filters.IFilterWithRasterisableParameter . . . 244
VectSharp.Filters.FilterWithRasterisableParameter . . . 153
VectSharp.IFontLibrary . . . 246
VectSharp.FontLibrary . . . 170
VectSharp.DefaultFontLibrary . . . 138
VectSharp.SimpleFontLibrary . . . 394
VectSharp.IGraphicsContext . . . 249
VectSharp.ThreeD.ILightSource . . . 258
VectSharp.ThreeD.AmbientLightSource . . . 29
VectSharp.ThreeD.AreaLightSource . . . 43
VectSharp.ThreeD.MaskedLightSource . . . 310
VectSharp.ThreeD.ParallelLightSource . . . 330
VectSharp.ThreeD.PointLightSource . . . 346
VectSharp.ThreeD.SpotlightLightSource . . . 437
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter . . . 261
VectSharp.ImageSharpUtils.ImageURIParser . . . 268
VectSharp.MuPDFUtils.ImageURIParser . . . 269
VectSharp.ThreeD.IMaterial . . . 270
VectSharp.ThreeD.ColourMaterial . . . 73
VectSharp.ThreeD.PhongMaterial . . . 338
IReadOnlyList
VectSharp.GradientStops . . . 189
VectSharp.ThreeD.IScene . . . 272
VectSharp.ThreeD.Scene . . . 386
VectSharp.ThreeD.LightIntensity . . . 275
VectSharp.LineDash . . . 282
VectSharp.Markdown.Margins . . . 284
VectSharp.Markdown.MarkdownRenderer . . . 291
VectSharp.Fonts.Nimbus . . . 318

VectSharp.ThreeD.ObjectFactory . . . 319
VectSharp.Page . . . 326
VectSharp.TrueTypeFile.PairKerning . . . 329
VectSharp.SVG.Parser . . . 334
VectSharp.PDF.PDFContextInterpreter . . . 336
VectSharp.Point . . . 342
VectSharp.TrueTypeFile.CoverageTable.RangeRecord . . . 353
VectSharp.Raster.Raster . . . 354
VectSharp.Rectangle . . . 370
VectSharp.Canvas.RenderAction . . . 376
VectSharp.Segment . . . 389
VectSharp.Size . . . 399
VectSharp.Canvas.SKRenderContext . . . 422
VectSharp.Canvas.SKRenderContextInterpreter . . . 422
VectSharp.SVG.SVGContextInterpreter . . . 441
VectSharp.Markdown.SyntaxHighlighter . . . 448
VectSharp.Transition . . . 449
VectSharp.TrueTypeFile . . . 451
VectSharp.TrueTypeFile.TrueTypePoint . . . 464
UserControl
  VectSharp.MarkdownCanvas.MarkdownCanvasControl . . . 286
VectSharp.TrueTypeFile.VerticalMetrics . . . 467

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

VectSharp.ThreeD.AmbientLightSource
Represents a uniform ambien light source. 29
VectSharp.Canvas.AnimatedCanvas
An Avalonia.Controls.Canvas containing an animation. 33
VectSharp.AnimatedPNG
Contains methods to create animated PNG image files. 37
VectSharp.Animation
Describes an animation constituted by a number of frames and transitions between them. . . . 38
VectSharp.ThreeD.AreaLightSource
Represents a light source emitting light from a circular area. 43
VectSharp.Canvas.AvaloniaContextInterpreter
Contains methods to render a Page to an Avalonia.Controls.Canvas. 48
VectSharp.TrueTypeFile.Bearings
Represents the left- and right-side bearings of a glyph. 53
VectSharp.Filters.BoxBlurFilter
Represents a filter applying a box blur. 54
VectSharp.Brush
Represents a brush used to fill or stroke graphics elements. This could be a solid colour, or a more complicated gradient or pattern. 57
VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord 58
VectSharp.Colour
Represents an RGB colour. 60
VectSharp.ThreeD.ColourMaterial
Represents a material that always has the same colour, regardless of light. 73
VectSharp.Filters.ColourMatrix
Represents a colour transformation matrix. 75
VectSharp.Filters.ColourMatrixFilter
Represents a filter that applies a Filters.ColourMatrix to the colours of the image. 89
VectSharp.Colours
Standard colours. 92
VectSharp.Filters.CompositeLocationInvariantFilter
Represents a filter that corresponds to applying multiple ILocationInvariantFilters one after the other. 128
VectSharp.AnimatedPNG.CompressedFrame
Represents an individual frame of a PNG animation. 131

VectSharp.Filters.ConvolutionFilter
Represents a filter that applies a matrix convolution to the image. 134
VectSharp.DefaultFontLibrary
A default font library that resolves standard families using the embedded fonts. 138
VectSharp.Font.DetailedFontMetrics
Represents detailed information about the metrics of a text string when drawn with a certain font. 140
VectSharp.DisposableIntPtr
An IDisposable wrapper around an IntPtr that frees the allocated memory when it is disposed. 142
VectSharp.Document
Represents a collection of pages. 144
VectSharp.Canvas.FilterOption
Determines how and whether image filters are rasterised. 145
VectSharp.PDF.PDFContextInterpreter.FilterOption
Determines how and whether image filters are rasterised. 147
VectSharp.SVG.SVGContextInterpreter.FilterOption
Determines how and whether image filters are rasterised. 150
VectSharp.Filters.FilterWithRasterisableParameter
Represents a filter with a parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter. 153
VectSharp.Font
Represents a typeface with a specific size. 154
VectSharp.FontFamily
Represents a typeface. 161
VectSharp.FontFamilyCreationException
An exception that occurs while creating a FontFamily. 168
VectSharp.FontLibrary
Abstract class with a default implementation of font family fallbacks. 170
VectSharp.Font.FontUnderline
Represents options to underline text. 173
VectSharp.Markdown.FormattedString
Represents a string with associated formatting information. 175
VectSharp.FormattedText
Represents a run of text that should be drawn with the same style. 177
VectSharp.FormattedTextExtensions
Contains extension methods for collections of FormattedText objects. 181
VectSharp.Frame
A key frame for an animation. 182
VectSharp.Filters.GaussianBlurFilter
Represents a filter that applies a Gaussian blur effect. 183
VectSharp.GradientBrush
Represents a brush painting with a gradient. 186
VectSharp.GradientStop
Represents a colour stop in a gradient. 187
VectSharp.GradientStops
Represents a read-only list of GradientStops. 189
VectSharp.Graphics
Represents an abstract drawing surface. 192
VectSharp.GraphicsPath
Represents a graphics path that can be filled or stroked. 223
VectSharp.Markdown.HTTPUtils
Contains utilities to resolve absolute and relative URIs. 239
VectSharp.IEasing
Describes a function used to transform the transition speed. 241
VectSharp.Filters.IFilter
Represents a filter. Do not implement this interface directly; instead, implement ILocationInvariantFilter or IFilterWithLocation. 242

VectSharp.Filters.IFilterWithLocation
Represents a filter whose results depend on the position of the subject image on the graphics surface. 243
VectSharp.Filters.IFilterWithRasterisableParameter
Represents a filter with a parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter. The FilterWithRasterisableParameter abstract class provides a default implementation of this interface. 244
VectSharp.IFontLibrary
Represents a font library with methods to create FontFamily objects from a string or from FontFamily.StandardFontFamilies. 246
VectSharp.IGraphicsContext
This interface should be implemented by classes intended to provide graphics output capability to a Graphics object. 249
VectSharp.ThreeD.ILightSource
Represents a light source. 258
VectSharp.Filters.ILocationInvariantFilter
Represents a filter that can be applied to an image regardless of its location on the graphics surface. 260
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter
Contains methods to render a Page to an Image. 261
VectSharp.ImageSharpUtils.ImageURIParser
Provides a method to parse an image URI into a page. 268
VectSharp.MuPDFUtils.ImageURIParser
Provides a method to parse an image URI into a page. 269
VectSharp.ThreeD.IMaterial
Represents a material used to the determine the appearance of Triangle3DElement. 270
VectSharp.ThreeD.IScene
Represents a 3D scene. 272
VectSharp.ThreeD.LightIntensity
Represents the intensity of a light source at a particular point. 275
VectSharp.LinearEasing
Describes a linear easing (i.e., no easing). 277
VectSharp.LinearGradientBrush
Represents a brush painting with a linear gradient. 278
VectSharp.LineDash
Represents instructions on how to paint a dashed line. 282
VectSharp.Markdown.Margins
Represents the margins of a page. 284
VectSharp.MarkdownCanvas.MarkdownCanvasControl
A control to display a Markdown document in an Avalonia application. 286
VectSharp.Markdown.MarkdownRenderer
Renders Markdown documents into VectSharp graphics objects. 291
VectSharp.ThreeD.MaskedLightSource
Represents a point light source with a stencil in front of it. 310
VectSharp.Filters.MaskFilter
Represents a filter that uses the luminance of an image to mask another image. 315
VectSharp.Fonts.Nimbus
Contains an IFontLibrary providing access to the Nimbus family of standard fonts (used e.g. by MuPDF). 318
VectSharp.ThreeD.ObjectFactory
A static class containing methods to create complex 3D objects. 319
VectSharp.Page
Represents a Graphics object with a width and height. 326
VectSharp.TrueTypeFile.PairKerning
Contains information describing how the position of two glyphs in a kerning pair should be altered. 329
VectSharp.ThreeD.ParallelLightSource
Represents a parallel light source. 330

VectSharp.SVG.Parser
Contains methods to read an SVG image file. 334
VectSharp.PDF.PDFContextInterpreter
Contains methods to render a Document as a PDF document. 336
VectSharp.ThreeD.PhongMaterial
Represents a material that uses a Phong reflection model to determine the colour of the material based on the light sources that hit it. 338
VectSharp.Point
Represents a point relative to an origin in the top-left corner. 342
VectSharp.ThreeD.PointLightSource
Represents a point light source. 346
VectSharp.RadialGradientBrush
Represents a brush painting with a radial gradient. 350
VectSharp.TrueTypeFile.CoverageTable.RangeRecord 353
VectSharp.Raster.Raster
Contains methods to render a page to a PNG image. 354
VectSharp.RasterImage
Represents a raster image, created from raw pixel data. Consider using the derived classes included in the NuGet package "VectSharp.MuPDFUtils" if you need to load a raster image from a file or a Stream. 358
VectSharp.ImageSharpUtils.RasterImageFile
A RasterImage created from a file. 364
VectSharp.MuPDFUtils.RasterImageFile
A RasterImage created from a file. 365
VectSharp.ImageSharpUtils.RasterImageStream
A RasterImage created from a stream. 366
VectSharp.MuPDFUtils.RasterImageStream
A RasterImage created from a stream. 368
VectSharp.Rectangle
Represents a rectangle. 370
VectSharp.Canvas.RenderAction
Represents a light-weight rendering action. 376
VectSharp.ResourceFontFamily
Represents a FontFamily created from a resource stream. 384
VectSharp.ThreeD.Scene
Represents a 3D scene. 386
VectSharp.Segment
Represents a segment as part of a GraphicsPath. 389
VectSharp.SimpleFontLibrary
A font library that can be used to cache and resolve font family names. 394
VectSharp.Size
Represents the size of an object. 399
VectSharp.Canvas.SKMultiLayerRenderCanvas
Represents a multi-threaded, triple-buffered canvas on which the image is drawn using SkiaSharp. 401
VectSharp.Canvas.SKRenderAction
Represents a light-weight rendering action. 409
VectSharp.Canvas.SKRenderContext
Represents a page that has been prepared for fast rendering using the SkiaSharp renderer. . . 422
VectSharp.Canvas.SKRenderContextInterpreter
Contains methods to render a Page to an Avalonia.Controls.Canvas using the SkiaSharp renderer. 422
VectSharp.SolidColourBrush
Represents a brush painting with a single solid colour. 431
VectSharp.SplineEasing
Describes an easing defined by a Cubic Bezier curve. 434
VectSharp.ThreeD.SpotlightLightSource
Represents a conic spotlight. 437

VectSharp.SVG.SVGContextInterpreter
Contains methods to render a Page as an SVG file. 441
VectSharp.Markdown.SyntaxHighlighter
Contains methods to perform syntax highlighting. 448
VectSharp.Transition
Describes the transition between two successive Frames. 449
VectSharp.TrueTypeFile
Represents a font file in TrueType format. Reference: http://stevehanov.↩ca/blog/?id=143, https://developer.apple.com/fonts/TrueType-↩Reference-Manual/, https://docs.microsoft.com/en-us/typography/opentype/spec/ 451
VectSharp.TrueTypeFile.TrueTypePoint
Represents a point in a TrueType path description. 464
VectSharp.UnbalancedStackException
The exception that is thrown when an unbalanced graphics state stack occurs. 465
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.UnknownFormatException
The exception that is raised when the output file format is not specified and the file name does not have an extension corresponding to a known file format. 466
VectSharp.TrueTypeFile.VerticalMetrics
Represents the maximum heigth above and depth below the baseline of a glyph. 467

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

VectSharp.Canvas/AnimatedCanvas.cs . . . 469
VectSharp.Canvas/AvaloniaContext.cs . . . 473
VectSharp.Canvas/RenderingParameters.cs . . . 507
VectSharp.Canvas/SKMultiLayerRenderCanvas.cs . . . 509
VectSharp.Canvas/SKRenderContext.cs . . . 524
VectSharp.Fonts.Nimbus/Nimbus.cs . . . 548
VectSharp.ImageSharpUtils/ImageUriParser.cs . . . 604
VectSharp.ImageSharpUtils/RasterImages.cs . . . 607
VectSharp.Markdown/HtmlTag.cs . . . 549
VectSharp.Markdown/Line.cs . . . 554
VectSharp.Markdown/MarkdownContext.cs . . . 556
VectSharp.Markdown/MarkdownRenderer.cs . . . 559
VectSharp.Markdown/SyntaxHighlighting.cs . . . 592
VectSharp.Markdown/Word.cs . . . 596
VectSharp.MarkdownCanvas/ImageCache.cs . . . 598
VectSharp.MarkdownCanvas/MarkdownCanvas.axaml.cs . . . 600
VectSharp.MuPDFUtils/ImageURIParser.cs . . . 605
VectSharp.MuPDFUtils/RasterImages.cs . . . 610
VectSharp.PDF/PDFContext.cs . . . 613
VectSharp.Raster.ImageSharp/GradientBrushApplicator.cs . . . 651
VectSharp.Raster.ImageSharp/ImageSharpContext.cs . . . 654
VectSharp.Raster/Raster.cs . . . 674
VectSharp.SVG/SVGContext.cs . . . 678
VectSharp.SVG/SVGParser.cs . . . 724
VectSharp.ThreeD/Lights.cs . . . 761
VectSharp.ThreeD/Materials.cs . . . 772
VectSharp.ThreeD/ObjectFactory.cs . . . 774
VectSharp.ThreeD/Scene.cs . . . 781
VectSharp/Animation.cs . . . 782
VectSharp/Brush.cs . . . 804
VectSharp/Colour.cs . . . 810
VectSharp/Document.cs . . . 818
VectSharp/Enums.cs . . . 820
VectSharp/Font.cs . . . 851
VectSharp/FontLibrary.cs . . . 860

VectSharp/FormattedText.cs 868
VectSharp/Graphics.cs 877
VectSharp/Graphics.Text.cs 899
VectSharp/GraphicsAction.cs 912
VectSharp/GraphicsPath.cs 915
VectSharp/ImageFormats.cs 959
VectSharp/Point.cs 973
VectSharp/RasterImage.cs 977
VectSharp/Segment.cs 981
VectSharp/SmoothSpline.cs 990
VectSharp/StandardColours.cs 992
VectSharp/TrueType.cs 1001
VectSharp/Filters/BoxBlurFilter.cs 822
VectSharp/Filters/ColourMatrixFilter.cs 827
VectSharp/Filters/CompositeFilter.cs 836
VectSharp/Filters/ConvolutionFilter.cs 838
VectSharp/Filters/Filters.cs 842
VectSharp/Filters/GaussianBlurFilter.cs 845
VectSharp/Filters/MaskFilter.cs 849

Chapter 6

Namespace Documentation

6.1 VectSharp Namespace Reference

Classes

- class AnimatedPNG

  Contains methods to create animated PNG image files.
- class Animation

  Describes an animation constituted by a number of frames and transitions between them.
- class Brush

  Represents a brush used to fill or stroke graphics elements. This could be a solid colour, or a more complicated gradient or pattern.
- struct Colour

  Represents an RGB colour.
- class Colours

  Standard colours.
- class DefaultFontLibrary

  A default font library that resolves standard families using the embedded fonts.
- class DisposableIntPtr

  An IDisposable wrapper around an IntPtr that frees the allocated memory when it is disposed.
- class Document

  Represents a collection of pages.
- class Font

  Represents a typeface with a specific size.
- class FontFamily

  Represents a typeface.
- class FontFamilyCreationException

  An exception that occurs while creating a FontFamily.
- class FontLibrary

  Abstract class with a default implementation of font family fallbacks.
- class FormattedText

  Represents a run of text that should be drawn with the same style.
- class FormattedTextExtensions

  Contains extension methods for collections of FormattedText objects.
- class Frame

  A key frame for an animation.

- class GradientBrush

  Represents a brush painting with a gradient.
- struct GradientStop

  Represents a colour stop in a gradient.
- class GradientStops

  Represents a read-only list of GradientStops.
- class Graphics

  Represents an abstract drawing surface.
- class GraphicsPath

  Represents a graphics path that can be filled or stroked.
- interface IEasing

  Describes a function used to transform the transition speed.
- interface IFontLibrary

  Represents a font library with methods to create FontFamily objects from a string or from FontFamily.StandardFontFamilies.
- interface IGraphicsContext

  This interface should be implemented by classes intended to provide graphics output capability to a Graphics object.
- class LinearEasing

  Describes a linear easing (i.e., no easing).
- class LinearGradientBrush

  Represents a brush painting with a linear gradient.
- struct LineDash

  Represents instructions on how to paint a dashed line.
- class Page

  Represents a Graphics object with a width and height.
- struct Point

  Represents a point relative to an origin in the top-left corner.
- class RadialGradientBrush

  Represents a brush painting with a radial gradient.
- class RasterImage

  Represents a raster image, created from raw pixel data. Consider using the derived classes included in the NuGet package "VectSharp.MuPDFUtils" if you need to load a raster image from a file or a Stream.
- struct Rectangle

  Represents a rectangle.
- class ResourceFontFamily

  Represents a FontFamily created from a resource stream.
- class Segment

  Represents a segment as part of a GraphicsPath.
- class SimpleFontLibrary

  A font library that can be used to cache and resolve font family names.
- struct Size

  Represents the size of an object.
- class SolidColourBrush

  Represents a brush painting with a single solid colour.
- class SplineEasing

  Describes an easing defined by a Cubic Bezier curve.
- class Transition

  Describes the transition between two successive Frames.
- class TrueTypeFile

  Represents a font file in TrueType format. Reference: http://stevehanov.ca/blog/?id=143, https://developer.apple.com/fonts/TrueType-Reference-Manual/, https://docs.microsoft.com/en-us/typography/opentype/spec/
- class UnbalancedStackException

  The exception that is thrown when an unbalanced graphics state stack occurs.

Enumerations

- enum TextBaselines

  Represent text baselines.
- enum TextAnchors

  Represents text anchors.
- enum LineCaps

  Represents line caps.
- enum LineJoins

  Represents line joining options.
- enum SegmentType

  Types of Segment.
- enum UnbalancedStackActions

  Represents ways to deal with unbalanced graphics state stacks.
- enum Script

  Represents the position of the text.
- enum PixelFormats

  Represents the pixel format of a raster image.

6.1.1 Enumeration Type Documentation

6.1.1.1 LineCaps

`enum VectSharp.LineCaps`

Represents line caps.

Definition at line 70 of file Enums.cs.

6.1.1.2 LineJoins

`enum VectSharp.LineJoins`

Represents line joining options.

Definition at line 91 of file Enums.cs.

6.1.1.3 PixelFormats

`enum VectSharp.PixelFormats`

Represents the pixel format of a raster image.

Definition at line 27 of file RasterImage.cs.

6.1.1.4 Script

`enum VectSharp.Script`

Represents the position of the text.

Definition at line 29 of file FormattedText.cs.

6.1.1.5 SegmentType

`enum VectSharp.SegmentType`

Types of Segment.

Definition at line 151 of file Enums.cs.

6.1.1.6 TextAnchors

`enum VectSharp.TextAnchors`

Represents text anchors.

Definition at line 49 of file Enums.cs.

6.1.1.7 TextBaselines

`enum VectSharp.TextBaselines`

Represent text baselines.

Definition at line 23 of file Enums.cs.

6.1.1.8 UnbalancedStackActions

`enum VectSharp.UnbalancedStackActions`

Represents ways to deal with unbalanced graphics state stacks.

Definition at line 182 of file Enums.cs.

6.2 VectSharp.Canvas Namespace Reference

Classes

- class AnimatedCanvas

  An Avalonia.Controls.Canvas containing an animation.
- class AvaloniaContextInterpreter

  Contains methods to render a Page to an Avalonia.Controls.Canvas.
- class FilterOption

  Determines how and whether image filters are rasterised.
- class RenderAction

  Represents a light-weight rendering action.
- class SKMultiLayerRenderCanvas

  Represents a multi-threaded, triple-buffered canvas on which the image is drawn using SkiaSharp.
- class SKRenderAction

  Represents a light-weight rendering action.
- class SKRenderContext

  Represents a page that has been prepared for fast rendering using the SkiaSharp renderer.
- class SKRenderContextInterpreter

  Contains methods to render a Page to an Avalonia.Controls.Canvas using the SkiaSharp renderer.

6.3 VectSharp.Filters Namespace Reference

Classes

- class BoxBlurFilter

  Represents a filter applying a box blur.
- class ColourMatrix

  Represents a colour transformation matrix.
- class ColourMatrixFilter

  Represents a filter that applies a Filters.ColourMatrix to the colours of the image.
- class CompositeLocationInvariantFilter

  Represents a filter that corresponds to applying multiple ILocationInvariantFilters one after the other.
- class ConvolutionFilter

  Represents a filter that applies a matrix convolution to the image.
- class FilterWithRasterisableParameter

  Represents a filter with a parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter.
- class GaussianBlurFilter

  Represents a filter that applies a Gaussian blur effect.
- interface IFilter

  Represents a filter. Do not implement this interface directly; instead, implement ILocationInvariantFilter or IFilterWithLocation.
- interface IFilterWithLocation

  Represents a filter whose results depend on the position of the subject image on the graphics surface.
- interface IFilterWithRasterisableParameter

  Represents a filter with a parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter. The FilterWithRasterisableParameter abstract class provides a default implementation of this interface.
- interface ILocationInvariantFilter

  Represents a filter that can be applied to an image regardless of its location on the graphics surface.
- class MaskFilter

  Represents a filter that uses the luminance of an image to mask another image.

6.4 VectSharp.Fonts Namespace Reference

Classes

- class Nimbus

  Contains an IFontLibrary providing access to the Nimbus family of standard fonts (used e.g. by MuPDF).

6.5 VectSharp.ImageSharpUtils Namespace Reference

Classes

- class ImageURIParser

  Provides a method to parse an image URI into a page.
- class RasterImageFile

  A RasterImage created from a file.
- class RasterImageStream

  A RasterImage created from a stream.

6.6 VectSharp.Markdown Namespace Reference

Classes

- struct FormattedString

  Represents a string with associated formatting information.
- class HTTPUtils

  Contains utilities to resolve absolute and relative URIs.
- class Margins

  Represents the margins of a page.
- class MarkdownRenderer

  Renders Markdown documents into VectSharp graphics objects.
- class SyntaxHighlighter

  Contains methods to perform syntax highlighting.

6.7 VectSharp.MarkdownCanvas Namespace Reference

Classes

- class MarkdownCanvasControl

  A control to display a Markdown document in an Avalonia application.

6.8 VectSharp.MuPDFUtils Namespace Reference

Classes

- class ImageURIParser

  Provides a method to parse an image URI into a page.
- class RasterImageFile

  A RasterImage created from a file.
- class RasterImageStream

  A RasterImage created from a stream.

6.9 VectSharp.PDF Namespace Reference

Classes

- class PDFContextInterpreter

  Contains methods to render a Document as a PDF document.

6.10 VectSharp.Raster Namespace Reference

Classes

- class Raster

  Contains methods to render a page to a PNG image.

6.11 VectSharp.Raster.ImageSharp Namespace Reference

Classes

- class ImageSharpContextInterpreter

  Contains methods to render a Page to an Image.

Enumerations

- enum OutputFormats

  Enumeration containing the supported output formats.

6.11.1 Enumeration Type Documentation

6.11.1.1 OutputFormats

`enum VectSharp.Raster.ImageSharp.OutputFormats`

Enumeration containing the supported output formats.

Definition at line 991 of file ImageSharpContext.cs.

6.12 VectSharp.SVG Namespace Reference

Classes

- class Parser

  Contains methods to read an SVG image file.
- class SVGContextInterpreter

  Contains methods to render a Page as an SVG file.

6.13 VectSharp.ThreeD Namespace Reference

Classes

- class AmbientLightSource

  Represents a uniform ambien light source.
- class AreaLightSource

  Represents a light source emitting light from a circular area.
- class ColourMaterial

  Represents a material that always has the same colour, regardless of light.
- interface ILightSource

  Represents a light source.
- interface IMaterial

  Represents a material used to the determine the appearance of Triangle3DElement.
- interface IScene

  Represents a 3D scene.
- struct LightIntensity

  Represents the intensity of a light source at a particular point.
- class MaskedLightSource

  Represents a point light source with a stencil in front of it.
- class ObjectFactory

  A static class containing methods to create complex 3D objects.
- class ParallelLightSource

  Represents a parallel light source.
- class PhongMaterial

  Represents a material that uses a Phong reflection model to determine the colour of the material based on the light sources that hit it.
- class PointLightSource

  Represents a point light source.
- class Scene

  Represents a 3D scene.
- class SpotlightLightSource

  Represents a conic spotlight.

Chapter 7

Class Documentation

7.1 VectSharp.ThreeD.AmbientLightSource Class Reference

Represents a uniform ambien light source.

Inheritance diagram for VectSharp.ThreeD.AmbientLightSource:

Public Member Functions

- AmbientLightSource (double intensity)

  Creates a new AmbientLightSource instance.
- LightIntensity GetLightAt (Point3D point)

  Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	*The Point3DElement at which the light intensity should be computed.*

Returns

- double GetObstruction (Point3D point, IEnumerable< Triangle3DElement > shadowingTriangles)

  Determines the amount of obstruction of the light that results at point due to the specified shadowingTriangles .

Parameters

point	*The Point3D at which the obstruction should be computed.*
shadowingTriangles	*A collection of Triangle3DElement casting shadows.*

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Properties

- double Intensity `[get, set]`

  The intensity of the light.
- bool CastsShadow `[get]`

  Determines whether the light casts a shadow or not.

7.1.1 Detailed Description

Represents a uniform ambien light source.

Definition at line 91 of file Lights.cs.

7.1.2 Constructor & Destructor Documentation

7.1.2.1 AmbientLightSource()

```
VectSharp.ThreeD.AmbientLightSource.AmbientLightSource (
            double intensity )
```

Creates a new AmbientLightSource instance.

Parameters

intensity	The intensity of the light.

Definition at line 105 of file Lights.cs.

7.1.3 Member Function Documentation

7.1.3.1 GetLightAt()

```
LightIntensity VectSharp.ThreeD.AmbientLightSource.GetLightAt (
            Point3D point )
```

Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	The Point3DElement at which the light intensity should be computed.

Returns

Implements VectSharp.ThreeD.ILightSource.

Definition at line 111 of file Lights.cs.

7.1.3.2 GetObstruction()

```
double VectSharp.ThreeD.AmbientLightSource.GetObstruction (
            Point3D point,
            IEnumerable< Triangle3DElement > shadowingTriangles )
```

Determines the amount of obstruction of the light that results at *point* due to the specified *shadowingTriangles* .

Parameters

point	The Point3D at which the obstruction should be computed.
shadowingTriangles	A collection of Triangle3DElement casting shadows.

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 117 of file Lights.cs.

7.1.4 Property Documentation

7.1.4.1 CastsShadow

```
bool VectSharp.ThreeD.AmbientLightSource.CastsShadow  [get]
```

Determines whether the light casts a shadow or not.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 99 of file Lights.cs.

7.1.4.2 Intensity

```
double VectSharp.ThreeD.AmbientLightSource.Intensity  [get], [set]
```

The intensity of the light.

Definition at line 96 of file Lights.cs.

The documentation for this class was generated from the following file:

- VectSharp.ThreeD/Lights.cs

7.2 VectSharp.Canvas.AnimatedCanvas Class Reference

An Avalonia.Controls.Canvas containing an animation.

Inheritance diagram for VectSharp.Canvas.AnimatedCanvas:

Public Member Functions

- override void Render (DrawingContext context)

Static Public Attributes

- static readonly StyledProperty< int > CurrentFrameProperty

  Defines the CurrentFrame property.
- static readonly DirectProperty< AnimatedCanvas, int > FrameCountProperty = AvaloniaProperty.Register↩ Direct<AnimatedCanvas, int>(nameof(FrameCount), o => o.maxFrameIndex)

  Defines the FrameCount property.
- static readonly DirectProperty< AnimatedCanvas, double > FrameRateProperty = AvaloniaProperty.↩ RegisterDirect<AnimatedCanvas, double>(nameof(FrameRate), o => o.FrameRate)

  Defines the FrameRate property.
- static readonly StyledProperty< bool > IsPlayingProperty

  Defines the IsPlaying property.

Properties

- int CurrentFrame `[get, set]`

  The current frame in the animation.
- int FrameCount `[get]`

  The number of frames in the animation.
- double FrameRate `[get]`

  The target frame rate of the animation.
- bool IsPlaying `[get, set]`

  The current frame in the animation.

7.2.1 Detailed Description

An Avalonia.Controls.Canvas containing an animation.

Definition at line 28 of file AnimatedCanvas.cs.

7.2.2 Member Function Documentation

7.2.2.1 Render()

```
override void VectSharp.Canvas.AnimatedCanvas.Render (
            DrawingContext context )
```

Definition at line 203 of file AnimatedCanvas.cs.

7.2.3 Member Data Documentation

7.2.3.1 CurrentFrameProperty

```
readonly StyledProperty<int> VectSharp.Canvas.AnimatedCanvas.CurrentFrameProperty  [static]
```

Initial value:

```
= AvaloniaProperty.Register<AnimatedCanvas, int>(nameof(CurrentFrame), coerce: (ownerObject, val) =>
        {
            AnimatedCanvas owner = (AnimatedCanvas)ownerObject;
            if (owner.maxFrameIndex > 0)
            {
                if (val < owner.maxFrameIndex)
                {
                    return val;
                }
                else
                {
                    owner.IsPlaying = false;
                    return owner.maxFrameIndex - 1;
                }
            }
            else
            {
                return val % owner.RenderActions.Length;
            }
        })
```

Defines the CurrentFrame property.

Definition at line 33 of file AnimatedCanvas.cs.

7.2.3.2 FrameCountProperty

```
readonly DirectProperty<AnimatedCanvas, int> VectSharp.Canvas.AnimatedCanvas.FrameCount↩
Property = AvaloniaProperty.RegisterDirect<AnimatedCanvas, int>(nameof(FrameCount), o => o.↩
maxFrameIndex)  [static]
```

Defines the FrameCount property.

Definition at line 67 of file AnimatedCanvas.cs.

7.2.3.3 FrameRateProperty

```
readonly DirectProperty<AnimatedCanvas, double> VectSharp.Canvas.AnimatedCanvas.FrameRate↩
Property = AvaloniaProperty.RegisterDirect<AnimatedCanvas, double>(nameof(FrameRate), o =>
o.FrameRate)  [static]
```

Defines the FrameRate property.

Definition at line 91 of file AnimatedCanvas.cs.

7.2.3.4 IsPlayingProperty

```
readonly StyledProperty<bool> VectSharp.Canvas.AnimatedCanvas.IsPlayingProperty  [static]
```

Initial value:

```
= AvaloniaProperty.Register<AnimatedCanvas, bool>(nameof(IsPlaying), false, coerce: (ownerObject, val) =>
        {
            AnimatedCanvas owner = (AnimatedCanvas)ownerObject;
            if (owner.IsPlaying && !val)
            {
                owner.RefreshTimer.Dispose();
            }
            else if (!owner.IsPlaying && val)
            {
                owner.RefreshTimer = new System.Threading.Timer(_ =>
                {
                    _ = Avalonia.Threading.Dispatcher.UIThread.InvokeAsync(() =>
                    {
                        owner.CurrentFrame++;
                    });
                }, null, 0, (long)(1000.0 / owner.FrameRate));
            }
            return val;
        })
```

Defines the IsPlaying property.

Definition at line 115 of file AnimatedCanvas.cs.

7.2.4 Property Documentation

7.2.4.1 CurrentFrame

```
int VectSharp.Canvas.AnimatedCanvas.CurrentFrame  [get], [set]
```

The current frame in the animation.

Definition at line 58 of file AnimatedCanvas.cs.

7.2.4.2 FrameCount

```
int VectSharp.Canvas.AnimatedCanvas.FrameCount  [get]
```

The number of frames in the animation.

Definition at line 74 of file AnimatedCanvas.cs.

7.2.4.3 FrameRate

```
double VectSharp.Canvas.AnimatedCanvas.FrameRate  [get]
```

The target frame rate of the animation.

Definition at line 98 of file AnimatedCanvas.cs.

7.2.4.4 IsPlaying

```
bool VectSharp.Canvas.AnimatedCanvas.IsPlaying  [get], [set]
```

The current frame in the animation.

Definition at line 140 of file AnimatedCanvas.cs.

The documentation for this class was generated from the following file:

- VectSharp.Canvas/AnimatedCanvas.cs

7.3 VectSharp.AnimatedPNG Class Reference

Contains methods to create animated PNG image files.

Classes

- class CompressedFrame

  Represents an individual frame of a PNG animation.

Public Types

- enum InterframeCompression

  Types of inter-frame compression.

Static Public Member Functions

- static unsafe void Create (Stream outputStream, int width, int height, bool hasAlpha, IReadOnlyList< CompressedFrame > compressedFrames, int repeatCount)

  Create a new animated PNG image, outputting it to the specified stream.

7.3.1 Detailed Description

Contains methods to create animated PNG image files.

Definition at line 30 of file ImageFormats.cs.

7.3.2 Member Enumeration Documentation

7.3.2.1 InterframeCompression

```
enum VectSharp.AnimatedPNG.InterframeCompression
```

Types of inter-frame compression.

Definition at line 35 of file ImageFormats.cs.

7.3.3 Member Function Documentation

7.3.3.1 Create()

```
static unsafe void VectSharp.AnimatedPNG.Create (
            Stream outputStream,
            int width,
            int height,
            bool hasAlpha,
            IReadOnlyList< CompressedFrame > compressedFrames,
            int repeatCount )  [static]
```

Create a new animated PNG image, outputting it to the specified stream.

Parameters

outputStream	The stream to which the animated PNG image will be written.
width	The width of the image in pixels.
height	The height of the image in pixels.
hasAlpha	If the frames of the image have an alpha channel, set this to `true`; otherwise, set it to `false`.
compressedFrames	The frames that will be used to create the animated PNG image.
repeatCount	The number of times that the animation should loop. Set this to 0 for an infinitely repeating animation.

Definition at line 181 of file ImageFormats.cs.

The documentation for this class was generated from the following file:

- VectSharp/ImageFormats.cs

7.4 VectSharp.Animation Class Reference

Describes an animation constituted by a number of frames and transitions between them.

Public Member Functions

- Animation (double width, double height, double linearisationResolution)

  Creates a new Animation with the specified width, height and linearisation resolution.
- Page GetFrameAtAbsolute (double time)

  Obtains the (interpolated) frame that should be displayed after the specified time has passed since the start of the animation.
- Page GetFrameAtRelative (double relativeTime)

  Obtains the (interpolated) frame that should be displayed after the specified relative time has passed since the start of the animation.
- void AddFrame (Frame frame, Transition transition=null)

  Adds a new frame to the animation, with the specified transition.
- void RemoveLastFrame ()

  Removes the last frame from the animation (and the corresponding transition).

Properties

- ImmutableList< Frame > Frames = ImmutableList<Frame>.Empty `[get]`

  The key frames of the animation.
- ImmutableList< Transition > Transitions = ImmutableList<Transition>.Empty `[get]`

  The transitions between successive frames of the animation. This array always contains one fewer element than Frames.
- double Width `[get, set]`

  The width of the animation.
- double Height `[get, set]`

  The height of the animation.
- Colour Background = Colour.FromRgba(255, 255, 255, 0) `[get, set]`

  The background colour of the animation.
- double LinearisationResolution `[get]`

  The absolute length between successive samples to use when linearising GraphicsPaths.
- double Duration = 0 `[get]`

  The total duration of the animation (not including the number of repeats).
- int RepeatCount = 0 `[get, set]`

  The number of times that the animation should repeat.

7.4.1 Detailed Description

Describes an animation constituted by a number of frames and transitions between them.

Definition at line 1421 of file Animation.cs.

7.4.2 Constructor & Destructor Documentation

7.4.2.1 Animation()

```
VectSharp.Animation.Animation (
            double width,
            double height,
            double linearisationResolution )
```

Creates a new Animation with the specified width, height and linearisation resolution.

Parameters

width	The width of the animation.
height	The height of the animation.
linearisationResolution	The absolute length between successive samples to use when linearising GraphicsPaths.

Definition at line 1472 of file Animation.cs.

7.4.3 Member Function Documentation

7.4.3.1 AddFrame()

```
void VectSharp.Animation.AddFrame (
            Frame frame,
            Transition transition = null )
```

Adds a new frame to the animation, with the specified transition.

Parameters

frame	The new frame to add to the animation.
transition	The transition that should be applied between the previous frame and the new frame. This parameter is ignored for the first frame. If this is `null`, the animation will abruptly change from one frame to the next.

Definition at line 1597 of file Animation.cs.

7.4.3.2 GetFrameAtAbsolute()

```
Page VectSharp.Animation.GetFrameAtAbsolute (
            double time )
```

Obtains the (interpolated) frame that should be displayed after the specified time has passed since the start of the animation.

Parameters

time	The time since the start of the animation (in milliseconds).

Returns

A Page containing the interpolated frame that should be displayed after the specified time has passed since the start of the animation.

Definition at line 1484 of file Animation.cs.

7.4.3.3 GetFrameAtRelative()

```
Page VectSharp.Animation.GetFrameAtRelative (
            double relativeTime )
```

Obtains the (interpolated) frame that should be displayed after the specified relative time has passed since the start of the animation.

Parameters

relativeTime	The time since the start of the animation (ranging from 0 for the start of the animation, to 1 for the end of the animation).

Returns

A Page containing the interpolated frame that should be displayed after the specified time has passed since the start of the animation.

Definition at line 1587 of file Animation.cs.

7.4.3.4 RemoveLastFrame()

```
void VectSharp.Animation.RemoveLastFrame ( )
```

Removes the last frame from the animation (and the corresponding transition).

Exceptions

ArgumentOutOfRangeException	Thrown if there are no frames in the animation.

Definition at line 1619 of file Animation.cs.

7.4.4 Property Documentation

7.4.4.1 Background

```
Colour VectSharp.Animation.Background = Colour.FromRgba(255, 255, 255, 0)  [get], [set]
```

The background colour of the animation.

Definition at line 1449 of file Animation.cs.

7.4.4.2 Duration

```
double VectSharp.Animation.Duration = 0  [get]
```

The total duration of the animation (not including the number of repeats).

Definition at line 1459 of file Animation.cs.

7.4.4.3 Frames

```
ImmutableList<Frame> VectSharp.Animation.Frames = ImmutableList<Frame>.Empty  [get]
```

The key frames of the animation.

Definition at line 1426 of file Animation.cs.

7.4.4.4 Height

```
double VectSharp.Animation.Height  [get], [set]
```

The height of the animation.

Definition at line 1444 of file Animation.cs.

7.4.4.5 LinearisationResolution

```
double VectSharp.Animation.LinearisationResolution  [get]
```

The absolute length between successive samples to use when linearising GraphicsPaths.

Definition at line 1454 of file Animation.cs.

7.4.4.6 RepeatCount

```
int VectSharp.Animation.RepeatCount = 0  [get], [set]
```

The number of times that the animation should repeat.

Definition at line 1464 of file Animation.cs.

7.4.4.7 Transitions

```
ImmutableList<Transition> VectSharp.Animation.Transitions = ImmutableList<Transition>.Empty
[get]
```

The transitions between successive frames of the animation. This array always contains one fewer element than Frames.

Definition at line 1431 of file Animation.cs.

7.4.4.8 Width

```
double VectSharp.Animation.Width  [get], [set]
```

The width of the animation.

Definition at line 1439 of file Animation.cs.

The documentation for this class was generated from the following file:

- VectSharp/Animation.cs

7.5 VectSharp.ThreeD.AreaLightSource Class Reference

Represents a light source emitting light from a circular area.

Inheritance diagram for VectSharp.ThreeD.AreaLightSource:

Public Member Functions

- AreaLightSource (double intensity, Point3D center, double radius, double penumbraRadius, Normalized↩ Vector3D direction, double sourceDistance, int shadowSamplingPointCount)

  Creates a new AreaLightSource instance.
- LightIntensity GetLightAt (Point3D point)

  Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	*The Point3DElement at which the light intensity should be computed.*

Returns

- double GetObstruction (Point3D point, IEnumerable< Triangle3DElement > shadowingTriangles)

  Determines the amount of obstruction of the light that results at point due to the specified shadowingTriangles .

Parameters

point	*The Point3D at which the obstruction should be computed.*
shadowingTriangles	*A collection of Triangle3DElement casting shadows.*

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Properties

- bool CastsShadow = true `[get, set]`

  Determines whether the light casts a shadow or not.
- Point3D Center `[get]`

  The centre of the light-emitting area.
- NormalizedVector3D Direction `[get]`

  The direction of the light's main axis, i.e. the normal to the plane containing the light-emitting area.
- double Radius `[get]`

  The radius of the light emitting area.
- double PenumbraRadius `[get]`

  The radius of the penumbra area.
- double Intensity `[get, set]`

  The base intensity of the light.
- double SourceDistance `[get]`

  The distance between the focal point of the light and the light's Center.
- double DistanceAttenuationExponent = 2 `[get, set]`

  An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.
- double PenumbraAttenuationExponent = 1 `[get, set]`

  An exponent determining how fast the light attenuates between the light-emitting area radius and the penumbra radius.
- int ShadowSamplingPointCount `[get]`

  The number of points to use when determining the amount of light that is obstructed at a certain point.

7.5.1 Detailed Description

Represents a light source emitting light from a circular area.

Definition at line 579 of file Lights.cs.

7.5.2 Constructor & Destructor Documentation

7.5.2.1 AreaLightSource()

```
VectSharp.ThreeD.AreaLightSource.AreaLightSource (
            double intensity,
            Point3D center,
            double radius,
            double penumbraRadius,
            NormalizedVector3D direction,
            double sourceDistance,
            int shadowSamplingPointCount )
```

Creates a new AreaLightSource instance.

Parameters

intensity	The base intensity of the light.
center	The centre of the light-emitting area.
radius	The radius of the light-emitting area.
penumbraRadius	The radius of the penumbra area.
direction	The direction of the light.
sourceDistance	The distance between the focal point of the light and the light's center.
shadowSamplingPointCount	The number of points to use when determining the amount of light that is obstructed at a certain point.

Definition at line 643 of file Lights.cs.

7.5.3 Member Function Documentation

7.5.3.1 GetLightAt()

```
LightIntensity VectSharp.ThreeD.AreaLightSource.GetLightAt (
            Point3D point )
```

Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	The Point3DElement at which the light intensity should be computed.

Returns

Implements VectSharp.ThreeD.ILightSource.

Definition at line 690 of file Lights.cs.

7.5.3.2 GetObstruction()

```
double VectSharp.ThreeD.AreaLightSource.GetObstruction (
            Point3D point,
            IEnumerable< Triangle3DElement > shadowingTriangles )
```

Determines the amount of obstruction of the light that results at *point* due to the specified *shadowingTriangles* .

Parameters

point	The Point3D at which the obstruction should be computed.
shadowingTriangles	A collection of Triangle3DElement casting shadows.

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 763 of file Lights.cs.

7.5.4 Property Documentation

7.5.4.1 CastsShadow

```
bool VectSharp.ThreeD.AreaLightSource.CastsShadow = true  [get], [set]
```

Determines whether the light casts a shadow or not.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 582 of file Lights.cs.

7.5.4.2 Center

```
Point3D VectSharp.ThreeD.AreaLightSource.Center  [get]
```

The centre of the light-emitting area.

Definition at line 587 of file Lights.cs.

7.5.4.3 Direction

```
NormalizedVector3D VectSharp.ThreeD.AreaLightSource.Direction  [get]
```

The direction of the light's main axis, i.e. the normal to the plane containing the light-emitting area.

Definition at line 594 of file Lights.cs.

7.5.4.4 DistanceAttenuationExponent

```
double VectSharp.ThreeD.AreaLightSource.DistanceAttenuationExponent = 2  [get], [set]
```

An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.

Definition at line 619 of file Lights.cs.

7.5.4.5 Intensity

```
double VectSharp.ThreeD.AreaLightSource.Intensity  [get], [set]
```

The base intensity of the light.

Definition at line 609 of file Lights.cs.

7.5.4.6 PenumbraAttenuationExponent

```
double VectSharp.ThreeD.AreaLightSource.PenumbraAttenuationExponent = 1  [get], [set]
```

An exponent determining how fast the light attenuates between the light-emitting area radius and the penumbra radius.

Definition at line 624 of file Lights.cs.

7.5.4.7 PenumbraRadius

```
double VectSharp.ThreeD.AreaLightSource.PenumbraRadius [get]
```

The radius of the penumbra area.

Definition at line 604 of file Lights.cs.

7.5.4.8 Radius

```
double VectSharp.ThreeD.AreaLightSource.Radius [get]
```

The radius of the light emitting area.

Definition at line 599 of file Lights.cs.

7.5.4.9 ShadowSamplingPointCount

```
int VectSharp.ThreeD.AreaLightSource.ShadowSamplingPointCount [get]
```

The number of points to use when determining the amount of light that is obstructed at a certain point.

Definition at line 629 of file Lights.cs.

7.5.4.10 SourceDistance

```
double VectSharp.ThreeD.AreaLightSource.SourceDistance [get]
```

The distance between the focal point of the light and the light's Center.

Definition at line 614 of file Lights.cs.

The documentation for this class was generated from the following file:

- VectSharp.ThreeD/Lights.cs

7.6 VectSharp.Canvas.AvaloniaContextInterpreter Class Reference

Contains methods to render a Page to an Avalonia.Controls.Canvas.

Public Types

- enum TextOptions

  Defines whether text items should be converted into paths when drawing.

Static Public Member Functions

- static Avalonia.Controls.Canvas PaintToCanvas (this Page page, TextOptions textOption=TextOptions.↩ ConvertIfNecessary, FilterOption filterOption=default)

  Render a Page to an Avalonia.Controls.Canvas.
- static Avalonia.Controls.Canvas PaintToCanvas (this Page page, bool graphicsAsControls, TextOptions text↩ Option=TextOptions.ConvertIfNecessary, FilterOption filterOption=default)

  Render a Page to an Avalonia.Controls.Canvas.
- static Avalonia.Controls.Canvas PaintToCanvas (this Page page, bool graphicsAsControls, Dictionary< string, Delegate > taggedActions, bool removeTaggedActionsAfterExecution=true, TextOptions text↩ Option=TextOptions.ConvertIfNecessary, FilterOption filterOption=default)

  Render a Page to an Avalonia.Controls.Canvas.
- static Avalonia.Controls.Canvas PaintToCanvas (this Page page, Dictionary< string, Delegate > tagged↩ Actions, bool removeTaggedActionsAfterExecution=true, TextOptions textOption=TextOptions.ConvertIf↩ Necessary, FilterOption filterOption=default)

  Render a Page to an Avalonia.Controls.Canvas.
- static AnimatedCanvas PaintToAnimatedCanvas (this Animation animation, double frameRate=60, double durationScaling=1, TextOptions textOption=TextOptions.ConvertIfNecessary, FilterOption filter↩ Option=default)

  Render an Animation to an Avalonia.Controls.Canvas.

7.6.1 Detailed Description

Contains methods to render a Page to an Avalonia.Controls.Canvas.

Definition at line 2492 of file AvaloniaContext.cs.

7.6.2 Member Enumeration Documentation

7.6.2.1 TextOptions

```
enum VectSharp.Canvas.AvaloniaContextInterpreter.TextOptions
```

Defines whether text items should be converted into paths when drawing.

Definition at line 2497 of file AvaloniaContext.cs.

7.6.3 Member Function Documentation

7.6.3.1 PaintToAnimatedCanvas()

```
static AnimatedCanvas VectSharp.Canvas.AvaloniaContextInterpreter.PaintToAnimatedCanvas (
            this Animation animation,
            double frameRate = 60,
            double durationScaling = 1,
            TextOptions textOption = TextOptions.ConvertIfNecessary,
            FilterOption filterOption = default )  [static]
```

Render an Animation to an Avalonia.Controls.Canvas.

Parameters

animation	The Animation to render.
frameRate	The target frame rate of the animation, in frames-per-second (fps).
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered animation.

Definition at line 2617 of file AvaloniaContext.cs.

7.6.3.2 PaintToCanvas() [1/4]

```
static Avalonia.Controls.Canvas VectSharp.Canvas.AvaloniaContextInterpreter.PaintToCanvas (
            this Page page,
            bool graphicsAsControls,
            Dictionary< string, Delegate > taggedActions,
            bool removeTaggedActionsAfterExecution = true,
            TextOptions textOption = TextOptions.ConvertIfNecessary,
            FilterOption filterOption = default )  [static]
```

Render a Page to an Avalonia.Controls.Canvas.

Parameters

page	The Page to render.
graphicsAsControls	If this is true, each graphics object (e.g. paths, text...) is rendered as a separate Avalonia.Controls.Control. Otherwise, they are directly rendered onto the drawing context (which is faster, but does not allow interactivity).
taggedActions	A Dictionary<String, Delegate> containing the Actions that will be performed on items with the corresponding tag. If *graphicsAsControls* is true, the delegates should be voids that accept one parameter of type TextBlock or Path (depending on the tagged item), otherwise, they should accept one parameter of type RenderAction and return an IEnumerable<RenderAction> of the actions that will actually be performed.

Parameters

removeTaggedActionsAfterExecution	Whether the Actions should be removed from *taggedActions* after their execution. Set to false if the same Action should be performed on multiple items with the same tag.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered graphics objects.

Definition at line 2568 of file AvaloniaContext.cs.

7.6.3.3 PaintToCanvas() [2/4]

```
static Avalonia.Controls.Canvas VectSharp.Canvas.AvaloniaContextInterpreter.PaintToCanvas (
            this Page page,
            bool graphicsAsControls,
            TextOptions textOption = TextOptions.ConvertIfNecessary,
            FilterOption filterOption = default )  [static]
```

Render a Page to an Avalonia.Controls.Canvas.

Parameters

page	The Page to render.
graphicsAsControls	If this is true, each graphics object (e.g. paths, text...) is rendered as a separate Avalonia.Controls.Control. Otherwise, they are directly rendered onto the drawing context (which is faster, but does not allow interactivity).
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered graphics objects.

Definition at line 2543 of file AvaloniaContext.cs.

7.6.3.4 PaintToCanvas() [3/4]

```
static Avalonia.Controls.Canvas VectSharp.Canvas.AvaloniaContextInterpreter.PaintToCanvas (
            this Page page,
```

```
Dictionary< string, Delegate > taggedActions,
bool removeTaggedActionsAfterExecution = true,
TextOptions textOption = TextOptions.ConvertIfNecessary,
FilterOption filterOption = default )  [static]
```

Render a Page to an Avalonia.Controls.Canvas.

Parameters

page	The Page to render.
taggedActions	A Dictionary<String, Delegate> containing the Actions that will be performed on items with the corresponding tag. The delegates should accept one parameter of type TextBlock or Path (depending on the tagged item).
removeTaggedActionsAfterExecution	Whether the Actions should be removed from *taggedActions* after their execution. Set to false if the same Action should be performed on multiple items with the same tag.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered graphics objects.

Definition at line 2592 of file AvaloniaContext.cs.

7.6.3.5 PaintToCanvas() [4/4]

```
static Avalonia.Controls.Canvas VectSharp.Canvas.AvaloniaContextInterpreter.PaintToCanvas (
            this Page page,
            TextOptions textOption = TextOptions.ConvertIfNecessary,
            FilterOption filterOption = default )  [static]
```

Render a Page to an Avalonia.Controls.Canvas.

Parameters

page	The Page to render.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered graphics objects.

Definition at line 2522 of file AvaloniaContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.Canvas/AvaloniaContext.cs

7.7 VectSharp.TrueTypeFile.Bearings Struct Reference

Represents the left- and right-side bearings of a glyph.

Public Attributes

- int LeftSideBearing

  The left-side bearing of the glyph.
- int RightSideBearing

  The right-side bearing of the glyph.

7.7.1 Detailed Description

Represents the left- and right-side bearings of a glyph.

Definition at line 2185 of file TrueType.cs.

7.7.2 Member Data Documentation

7.7.2.1 LeftSideBearing

```
int VectSharp.TrueTypeFile.Bearings.LeftSideBearing
```

The left-side bearing of the glyph.

Definition at line 2190 of file TrueType.cs.

7.7.2.2 RightSideBearing

```
int VectSharp.TrueTypeFile.Bearings.RightSideBearing
```

The right-side bearing of the glyph.

Definition at line 2195 of file TrueType.cs.

The documentation for this struct was generated from the following file:

- VectSharp/TrueType.cs

7.8 VectSharp.Filters.BoxBlurFilter Class Reference

Represents a filter applying a box blur.

Inheritance diagram for VectSharp.Filters.BoxBlurFilter:

Public Member Functions

- BoxBlurFilter (double boxRadius)

  Creates a new BoxBlurFilter with the specified radius.
- RasterImage Filter (RasterImage image, double scale)

  Applies the filter to a RasterImage.

Parameters

image	*The RasterImage to which the filter will be applied.*
scale	*The scale of the image with respect to the filter.*

Returns

A new RasterImage containing the filtered image. The source image is left unaltered.

Properties

- double BoxRadius `[get]`

  The radius of the box blur (the actual size of the box is 2 ∗ BoxRadius + 1).
- Point TopLeftMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.
- Point BottomRightMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

7.8.1 Detailed Description

Represents a filter applying a box blur.

Definition at line 26 of file BoxBlurFilter.cs.

7.8.2 Constructor & Destructor Documentation

7.8.2.1 BoxBlurFilter()

```
VectSharp.Filters.BoxBlurFilter.BoxBlurFilter (
            double boxRadius )
```

Creates a new BoxBlurFilter with the specified radius.

Parameters

boxRadius	The radius of the box blur (the actual size of the box is 2 ∗ *boxRadius* + 1).

Definition at line 42 of file BoxBlurFilter.cs.

7.8.3 Member Function Documentation

7.8.3.1 Filter()

```
RasterImage VectSharp.Filters.BoxBlurFilter.Filter (
            RasterImage image,
            double scale )
```

Applies the filter to a RasterImage.

Parameters

image	The RasterImage to which the filter will be applied.
scale	The scale of the image with respect to the filter.

Returns

A new RasterImage containing the filtered image. The source *image* is left unaltered.

Implements VectSharp.Filters.ILocationInvariantFilter.

Definition at line 50 of file BoxBlurFilter.cs.

7.8.4 Property Documentation

7.8.4.1 BottomRightMargin

```
Point VectSharp.Filters.BoxBlurFilter.BottomRightMargin [get]
```

Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 36 of file BoxBlurFilter.cs.

7.8.4.2 BoxRadius

```
double VectSharp.Filters.BoxBlurFilter.BoxRadius [get]
```

The radius of the box blur (the actual size of the box is 2 ∗ BoxRadius + 1).

Definition at line 31 of file BoxBlurFilter.cs.

7.8.4.3 TopLeftMargin

```
Point VectSharp.Filters.BoxBlurFilter.TopLeftMargin [get]
```

Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 34 of file BoxBlurFilter.cs.

The documentation for this class was generated from the following file:

- VectSharp/Filters/BoxBlurFilter.cs

7.9 VectSharp.Brush Class Reference

Represents a brush used to fill or stroke graphics elements. This could be a solid colour, or a more complicated gradient or pattern.

Inheritance diagram for VectSharp.Brush:

Public Member Functions

- abstract Brush MultiplyOpacity (double opacity)

  Returns a brush corresponding the current instance, with the specified opacity multiplication applied.

Static Public Member Functions

- static implicit operator Brush (Colour colour)

  Implicitly converts a Colour into a SolidColourBrush.

7.9.1 Detailed Description

Represents a brush used to fill or stroke graphics elements. This could be a solid colour, or a more complicated gradient or pattern.

Definition at line 30 of file Brush.cs.

7.9.2 Member Function Documentation

7.9.2.1 MultiplyOpacity()

```
abstract Brush VectSharp.Brush.MultiplyOpacity (
            double opacity )  [pure virtual]
```

Returns a brush corresponding the current instance, with the specified *opacity* multiplication applied.

Parameters

opacity	The value that will be used to multiply the opacity of the brush.

Returns

A brush corresponding the current instance, with the specified *opacity* multiplication applied.

Implemented in VectSharp.SolidColourBrush, VectSharp.LinearGradientBrush, and VectSharp.RadialGradientBrush.

7.9.2.2 operator Brush()

```
static implicit VectSharp.Brush.operator Brush (
            Colour colour )  [static]
```

Implicitly converts a Colour into a SolidColourBrush.

Parameters

colour	The Colour to use for the brush.

Definition at line 45 of file Brush.cs.

The documentation for this class was generated from the following file:

- VectSharp/Brush.cs

7.10 VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord Struct Reference

Public Member Functions

- ClassRangeRecord (Stream fs)

Public Attributes

- ushort StartGlyphID
- ushort EndGlyphID
- ushort Class

7.10.1 Detailed Description

Definition at line 3743 of file TrueType.cs.

7.10.2 Constructor & Destructor Documentation

7.10.2.1 ClassRangeRecord()

```
VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord.ClassRangeRecord (
            Stream fs )
```

Definition at line 3749 of file TrueType.cs.

7.10.3 Member Data Documentation

7.10.3.1 Class

```
ushort VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord.Class
```

Definition at line 3747 of file TrueType.cs.

7.10.3.2 EndGlyphID

```
ushort VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord.EndGlyphID
```

Definition at line 3746 of file TrueType.cs.

7.10.3.3 StartGlyphID

```
ushort VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord.StartGlyphID
```

Definition at line 3745 of file TrueType.cs.

The documentation for this struct was generated from the following file:

- VectSharp/TrueType.cs

7.11 VectSharp.Colour Struct Reference

Represents an RGB colour.

Inheritance diagram for VectSharp.Colour:

Public Member Functions

- override bool Equals (object obj)
- bool Equals (Colour col)
- override int GetHashCode ()
- string ToCSSString (bool includeAlpha)

  Convert the Colour object into a hex string that is constituted by a "#" followed by two-digit hexadecimal representations of the red, green and blue components of the colour (in the range 0x00 - 0xFF). Optionally also includes opacity (alpha channel) data.
- Colour WithAlpha (double alpha)

  Create a new Colour with the same RGB components as the current Colour, but with the specified alpha .
- Colour WithAlpha (byte alpha)

  Create a new Colour with the same RGB components as the current Colour, but with the specified alpha .
- double double double Z ToXYZ ()
- double double double b ToLab ()
- double double double L ToHSL ()

Static Public Member Functions

- static Colour FromRgb (double r, double g, double b)

  Create a new colour from RGB (red, green and blue) values.
- static Colour FromRgb (byte r, byte g, byte b)

  Create a new colour from RGB (red, green and blue) values.
- static Colour FromRgb (int r, int g, int b)

  Create a new colour from RGB (red, green and blue) values.
- static Colour FromRgba (double r, double g, double b, double a)

  Create a new colour from RGBA (red, green, blue and alpha) values.
- static Colour FromRgba (byte r, byte g, byte b, byte a)

  Create a new colour from RGBA (red, green, blue and alpha) values.
- static Colour FromRgba (byte r, byte g, byte b, double a)

  Create a new colour from RGBA (red, green, blue and alpha) values.

- static Colour FromRgba (int r, int g, int b, int a)

  Create a new colour from RGBA (red, green, blue and alpha) values.
- static Colour FromRgba (int r, int g, int b, double a)

  Create a new colour from RGBA (red, green, blue and alpha) values.
- static Colour FromRgba ((int r, int g, int b, double a) colour)

  Create a new colour from RGBA (red, green, blue and alpha) values.
- static bool operator== (Colour col1, Colour col2)
- static bool operator!= (Colour col1, Colour col2)
- static ? Colour FromCSSString (string cssString)

  Convert a CSS colour string into a Colour object.
- static Colour WithAlpha (Colour original, double alpha)

  Create a new Colour with the same RGB components as the original Colour, but with the specified alpha .
- static Colour WithAlpha (Colour original, byte alpha)

  Create a new Colour with the same RGB components as the original Colour, but with the specified alpha .
- static Colour FromXYZ (double x, double y, double z)

  Creates a Colour from CIE XYZ coordinates.
- static Colour FromLab (double L, double a, double b)

  Creates a Colour from CIE Lab coordinates (under Illuminant D65).
- static Colour FromHSL (double h, double s, double l)

  Creates a Colour from HSL coordinates.

Public Attributes

- double R

  Red component of the colour. Range: [0, 1].
- double G

  Green component of the colour. Range: [0, 1].
- double B

  Blue component of the colour. Range: [0, 1].
- double A

  Alpha component of the colour. Range: [0, 1].
- double X

  Converts a Colour to the CIE XYZ colour space.
- double double Y
- double L

  Converts a Colour to the CIE Lab colour space (under Illuminant D65).
- double double a
- double H

  Converts a Colour to the HSL colour space.
- double double S

7.11.1 Detailed Description

Represents an RGB colour.

Definition at line 25 of file Colour.cs.

7.11.2 Member Function Documentation

7.11.2.1 Equals() [1/2]

```
bool VectSharp.Colour.Equals (
            Colour col )
```

Definition at line 179 of file Colour.cs.

7.11.2.2 Equals() [2/2]

```
override bool VectSharp.Colour.Equals (
            object obj )
```

Definition at line 166 of file Colour.cs.

7.11.2.3 FromCSSString()

```
static ? Colour VectSharp.Colour.FromCSSString (
            string cssString )  [static]
```

Convert a CSS colour string into a Colour object.

Parameters

cssString	The CSS colour string. In addition to 148 standard colour names (case-insensitive), #RGB, #RGBA, #RRGGBB and #RRGGBBAA hex strings and rgb(r, g, b) and rgba(r, g, b, a) functional colour notations are supported.

Returns

Definition at line 225 of file Colour.cs.

7.11.2.4 FromHSL()

```
static Colour VectSharp.Colour.FromHSL (
            double h,
            double s,
            double l )  [static]
```

Creates a Colour from HSL coordinates.

Parameters

h	The H component. Should be in range [0, 1].
s	The S component. Should be in range [0, 1].
l	The L component. Should be in range [0, 1].

Returns

A Colour created from the specified components.

Definition at line 582 of file Colour.cs.

7.11.2.5 FromLab()

```
static Colour VectSharp.Colour.FromLab (
            double L,
            double a,
            double b )  [static]
```

Creates a Colour from CIE Lab coordinates (under Illuminant D65).

Parameters

L	The L∗ component.
a	The a∗ component.
b	The b∗ component.

Returns

An sRGB Colour created from the specified components.

Definition at line 504 of file Colour.cs.

7.11.2.6 FromRgb() [1/3]

```
static Colour VectSharp.Colour.FromRgb (
            byte r,
            byte g,
            byte b )  [static]
```

Create a new colour from RGB (red, green and blue) values.

Parameters

r	The red component of the colour. Range: [0, 255].
g	The green component of the colour. Range: [0, 255].
b	The blue component of the colour. Range: [0, 255].

Returns

A Colour struct with the specified components and an alpha component of 1.

Definition at line 74 of file Colour.cs.

7.11.2.7 FromRgb() [2/3]

```
static Colour VectSharp.Colour.FromRgb (
            double r,
            double g,
            double b )  [static]
```

Create a new colour from RGB (red, green and blue) values.

Parameters

r	The red component of the colour. Range: [0, 1].
g	The green component of the colour. Range: [0, 1].
b	The blue component of the colour. Range: [0, 1].

Returns

A Colour struct with the specified components and an alpha component of 1.

Definition at line 62 of file Colour.cs.

7.11.2.8 FromRgb() [3/3]

```
static Colour VectSharp.Colour.FromRgb (
            int r,
            int g,
            int b )  [static]
```

Create a new colour from RGB (red, green and blue) values.

Parameters

r	The red component of the colour. Range: [0, 255].
g	The green component of the colour. Range: [0, 255].
b	The blue component of the colour. Range: [0, 255].

Returns

A Colour struct with the specified components and an alpha component of 1.

Definition at line 86 of file Colour.cs.

7.11.2.9 FromRgba() [1/6]

```
static Colour VectSharp.Colour.FromRgba (
            (int r, int g, int b, double a) colour )  [static]
```

Create a new colour from RGBA (red, green, blue and alpha) values.

Parameters

colour	A ValueTuple<Int32, Int32, Int32, Double> containing component information for the colour. For r, g, and b, range: [0, 255]; for a, range: [0, 1].

Returns

A Colour struct with the specified components.

Definition at line 160 of file Colour.cs.

7.11.2.10 FromRgba() [2/6]

```
static Colour VectSharp.Colour.FromRgba (
            byte r,
            byte g,
            byte b,
            byte a )  [static]
```

Create a new colour from RGBA (red, green, blue and alpha) values.

Parameters

r	The red component of the colour. Range: [0, 255].
g	The green component of the colour. Range: [0, 255].
b	The blue component of the colour. Range: [0, 255].
a	The alpha component of the colour. Range: [0, 255].

Returns

A ColourColour struct with the specified components.

Definition at line 112 of file Colour.cs.

7.11.2.11 FromRgba() [3/6]

```
static Colour VectSharp.Colour.FromRgba (
            byte r,
            byte g,
            byte b,
            double a )  [static]
```

Create a new colour from RGBA (red, green, blue and alpha) values.

Parameters

r	The red component of the colour. Range: [0, 255].
g	The green component of the colour. Range: [0, 255].
b	The blue component of the colour. Range: [0, 255].
a	The alpha component of the colour. Range: [0, 1].

Returns

A Colour struct with the specified components.

Definition at line 125 of file Colour.cs.

7.11.2.12 FromRgba() [4/6]

```
static Colour VectSharp.Colour.FromRgba (
            double r,
            double g,
            double b,
            double a )  [static]
```

Create a new colour from RGBA (red, green, blue and alpha) values.

Parameters

r	The red component of the colour. Range: [0, 1].
g	The green component of the colour. Range: [0, 1].
b	The blue component of the colour. Range: [0, 1].
a	The alpha component of the colour. Range: [0, 1].

Returns

A Colour struct with the specified components.

Definition at line 99 of file Colour.cs.

7.11.2.13 FromRgba() [5/6]

```
static Colour VectSharp.Colour.FromRgba (
            int r,
            int g,
            int b,
            double a )  [static]
```

Create a new colour from RGBA (red, green, blue and alpha) values.

Parameters

r	The red component of the colour. Range: [0, 255].
g	The green component of the colour. Range: [0, 255].
b	The blue component of the colour. Range: [0, 255].
a	The alpha component of the colour. Range: [0, 1].

Returns

A Colour struct with the specified components.

Definition at line 150 of file Colour.cs.

7.11.2.14 FromRgba() [6/6]

```
static Colour VectSharp.Colour.FromRgba (
            int r,
            int g,
            int b,
            int a )  [static]
```

Create a new colour from RGBA (red, green, blue and alpha) values.

Parameters

r	The red component of the colour. Range: [0, 255].
g	The green component of the colour. Range: [0, 255].
b	The blue component of the colour. Range: [0, 255].
a	The alpha component of the colour. Range: [0, 255].

Returns

A Colour struct with the specified components.

Definition at line 137 of file Colour.cs.

7.11.2.15 FromXYZ()

```
static Colour VectSharp.Colour.FromXYZ (
            double x,
            double y,
            double z )  [static]
```

Creates a Colour from CIE XYZ coordinates.

Parameters

x	The X coordinate.
y	The Y coordinate.
z	The Z coordinate.

Returns

An sRGB Colour created from the specified components.

Definition at line 422 of file Colour.cs.

7.11.2.16 GetHashCode()

```
override int VectSharp.Colour.GetHashCode ( )
```

Definition at line 197 of file Colour.cs.

7.11.2.17 operator"!=()

```
static bool VectSharp.Colour.operator!= (
            Colour col1,
            Colour col2 )  [static]
```

Definition at line 191 of file Colour.cs.

7.11.2.18 operator==()

```
static bool VectSharp.Colour.operator== (
            Colour col1,
            Colour col2 )  [static]
```

Definition at line 185 of file Colour.cs.

7.11.2.19 ToCSSString()

```
string VectSharp.Colour.ToCSSString (
            bool includeAlpha )
```

Convert the Colour object into a hex string that is constituted by a "#" followed by two-digit hexadecimal representations of the red, green and blue components of the colour (in the range 0x00 - 0xFF). Optionally also includes opacity (alpha channel) data.

Parameters

includeAlpha	Whether two additional hex digits representing the colour's opacity (alpha channel) should be included in the string.

Returns

A hex colour string.

Definition at line 208 of file Colour.cs.

7.11.2.20 ToHSL()

```
double double double L VectSharp.Colour.ToHSL ( )
```

Definition at line 535 of file Colour.cs.

7.11.2.21 ToLab()

```
double double double b VectSharp.Colour.ToLab ( )
```

Definition at line 466 of file Colour.cs.

7.11.2.22 ToXYZ()

```
double double double Z VectSharp.Colour.ToXYZ ( )
```

Definition at line 377 of file Colour.cs.

7.11.2.23 WithAlpha() [1/4]

```
Colour VectSharp.Colour.WithAlpha (
            byte alpha )
```

Create a new Colour with the same RGB components as the current Colour, but with the specified *alpha* .

Parameters

alpha	The alpha component of the new Colour.

Returns

A Colour struct with the same RGB components as the current Colour and the specified *alpha* .

Definition at line 368 of file Colour.cs.

7.11.2.24 WithAlpha() [2/4]

```
static Colour VectSharp.Colour.WithAlpha (
            Colour original,
            byte alpha )  [static]
```

Create a new Colour with the same RGB components as the *original* Colour, but with the specified *alpha* .

Parameters

original	The original Colour from which the RGB components will be taken.
alpha	The alpha component of the new Colour.

Returns

A Colour struct with the same RGB components as the *original* Colour and the specified *alpha* .

Definition at line 348 of file Colour.cs.

7.11.2.25 WithAlpha() [3/4]

```
static Colour VectSharp.Colour.WithAlpha (
            Colour original,
            double alpha )  [static]
```

Create a new Colour with the same RGB components as the *original* Colour, but with the specified *alpha* .

Parameters

original	The original Colour from which the RGB components will be taken.
alpha	The alpha component of the new Colour.

Returns

A Colour struct with the same RGB components as the *original* Colour and the specified *alpha* .

Definition at line 337 of file Colour.cs.

7.11.2.26 WithAlpha() [4/4]

```
Colour VectSharp.Colour.WithAlpha (
            double alpha )
```

Create a new Colour with the same RGB components as the current Colour, but with the specified *alpha* .

Parameters

alpha	The alpha component of the new Colour.

Returns

A Colour struct with the same RGB components as the current Colour and the specified *alpha* .

Definition at line 358 of file Colour.cs.

7.11.3 Member Data Documentation

7.11.3.1 A

```
double VectSharp.Colour.A
```

Alpha component of the colour. Range: [0, 1].

Definition at line 45 of file Colour.cs.

7.11.3.2 a

```
double double VectSharp.Colour.a
```

Definition at line 466 of file Colour.cs.

7.11.3.3 B

```
double VectSharp.Colour.B
```

Blue component of the colour. Range: [0, 1].

Definition at line 40 of file Colour.cs.

7.11.3.4 G

```
double VectSharp.Colour.G
```

Green component of the colour. Range: [0, 1].

Definition at line 35 of file Colour.cs.

7.11.3.5 H

```
double VectSharp.Colour.H
```

Converts a Colour to the HSL colour space.

Returns

A ValueType containing the H, S and L components of the Colour. Each component has range [0, 1].

Definition at line 535 of file Colour.cs.

7.11.3.6 L

```
double VectSharp.Colour.L
```

Converts a Colour to the CIE Lab colour space (under Illuminant D65).

Returns

A ValueType containing the L∗, a∗ and b∗ components of the Colour.

Definition at line 466 of file Colour.cs.

7.11.3.7 R

```
double VectSharp.Colour.R
```

Red component of the colour. Range: [0, 1].

Definition at line 30 of file Colour.cs.

7.11.3.8 S

```
double double VectSharp.Colour.S
```

Definition at line 535 of file Colour.cs.

7.11.3.9 X

```
double VectSharp.Colour.X
```

Converts a Colour to the CIE XYZ colour space.

Returns

A ValueTuple containing the X, Y and Z components of the Colour.

Definition at line 377 of file Colour.cs.

7.11.3.10 Y

```
double double VectSharp.Colour.Y
```

Definition at line 377 of file Colour.cs.

The documentation for this struct was generated from the following files:

- VectSharp/Colour.cs
- VectSharp/StandardColours.cs

7.12 VectSharp.ThreeD.ColourMaterial Class Reference

Represents a material that always has the same colour, regardless of light.

Inheritance diagram for VectSharp.ThreeD.ColourMaterial:

Public Member Functions

- ColourMaterial (Colour colour)

  Creates a new ColourMaterial instance.
- Colour GetColour (Point3D point, NormalizedVector3D surfaceNormal, Camera camera, IList< ILightSource > lights, IList< double > obstructions)

  Obtains the Colour at the specified point.

Parameters

point	*The point whose colour should be determined.*
surfaceNormal	*The normal to the surface at the specified point .*
camera	*The camera being used to render the scene.*
lights	*A list of light sources that are present in the scene.*
obstructions	*A list of values indicating how obstructed each light source is.*

Returns

The Colour of the specified point.

Properties

- Colour Colour `[get]`

  The colour of the material.

7.12.1 Detailed Description

Represents a material that always has the same colour, regardless of light.

Definition at line 48 of file Materials.cs.

7.12.2 Constructor & Destructor Documentation

7.12.2.1 ColourMaterial()

```
VectSharp.ThreeD.ColourMaterial.ColourMaterial (
            Colour colour )
```

Creates a new ColourMaterial instance.

Parameters

colour	The colour of the material.

Definition at line 59 of file Materials.cs.

7.12.3 Member Function Documentation

7.12.3.1 GetColour()

```
Colour VectSharp.ThreeD.ColourMaterial.GetColour (
            Point3D point,
            NormalizedVector3D surfaceNormal,
            Camera camera,
            IList< ILightSource > lights,
            IList< double > obstructions )
```

Obtains the Colour at the specified point.

Parameters

point	The point whose colour should be determined.
surfaceNormal	The normal to the surface at the specified *point* .
camera	The camera being used to render the scene.
lights	A list of light sources that are present in the scene.
obstructions	A list of values indicating how obstructed each light source is.

Returns

The Colour of the specified point.

Implements VectSharp.ThreeD.IMaterial.

Definition at line 65 of file Materials.cs.

7.12.4 Property Documentation

7.12.4.1 Colour

```
Colour VectSharp.ThreeD.ColourMaterial.Colour  [get]
```

The colour of the material.

Definition at line 53 of file Materials.cs.

The documentation for this class was generated from the following file:

- VectSharp.ThreeD/Materials.cs

7.13 VectSharp.Filters.ColourMatrix Class Reference

Represents a colour transformation matrix.

Public Member Functions

- ColourMatrix WithAlpha (double alpha)

  Creates a new ColourMatrix whose alpha coefficients are multiplied by the specified value.
- ColourMatrix (double[,] matrix)

  Creates a new ColourMatrix with the specified coefficients.
- Colour Apply (Colour colour)

  Applies the ColourMatrix to the specified Colour.
- void Apply (ref byte R, ref byte G, ref byte B, ref byte A)

  Applies the ColourMatrix to the specified colour, represented as four bytes, and stores the resulting colour in the same variables as the original RGBA values.
- void Apply (ref byte R, ref byte G, ref byte B)

  Applies the ColourMatrix to the specified colour, represented as three bytes, and stores the resulting colour in the same variables as the original RGB values.
- void Apply (byte R, byte G, byte B, byte A, out byte r, out byte g, out byte b, out byte a)

  Applies the ColourMatrix to the specified colour, represented as four bytes, and stores the resulting colour in the specified output bytes.
- void Apply (byte R, byte G, byte B, out byte r, out byte g, out byte b)

  Applies the ColourMatrix to the specified colour, represented as three bytes, and stores the resulting colour in the specified output bytes.

Static Public Member Functions

- static ColourMatrix ToColour (Colour colour, bool useAlpha=false)

  Creates a ColourMatrix that turns every colour to which it is applied into the specified colour .
- static ColourMatrix LuminanceToColour (Colour colour, bool useAlpha=false)

  Creates a ColourMatrix that turns every colour to which it is applied into a shade of the specified colour . The brightness of the output colour depends on the luminance of the input colour.
- static ColourMatrix LuminanceToAlpha (bool preserveColour=false)

  Creates a ColourMatrix that transforms the alpha value of the colour it is applied to into a value depending on the luminance of the input colour.
- static ColourMatrix operator∗ (ColourMatrix matrix1, ColourMatrix matrix2)

  Concatenates two matrices. The resulting ColourMatrix is equivalent to first applying matrix2 , and then matrix1 .

Static Public Attributes

- static ColourMatrix Identity = new ColourMatrix(new double[,] { { 1, 0, 0, 0, 0 }, { 0, 1, 0, 0, 0 }, { 0, 0, 1, 0, 0 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } })

  A ColourMatrix that whose output colour is always the same as the input colour.
- static ColourMatrix GreyScale = new ColourMatrix(new double[,] { { 0.2126, 0.7152, 0.0722, 0, 0 }, { 0.2126, 0.7152, 0.0722, 0, 0 }, { 0.2126, 0.7152, 0.0722, 0, 0 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } })

  A ColourMatrix that transforms every colour in a shade of grey with approximately the same luminance.
- static ColourMatrix Pastel = new ColourMatrix(new double[,] { { 0.75, 0.25, 0.25, 0, 0 }, { 0.25, 0.75, 0.25, 0, 0 }, { 0.25, 0.25, 0.75, 0, 0 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } })

  A ColourMatrix producing a "pastel" (desaturation) effect.
- static ColourMatrix Inversion = new ColourMatrix(new double[,] { { -1, 0, 0, 0, 1 }, { 0, -1, 0, 0, 1 }, { 0, 0, -1, 0, 1 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } })

  A ColourMatrix that inverts every colour, leaving the alpha component intact.
- static ColourMatrix AlphaInversion = new ColourMatrix(new double[,] { { 1, 0, 0, 0, 0 }, { 0, 1, 0, 0, 0 }, { 0, 0, 1, 0, 0 }, { 0, 0, 0, -1, 1 }, { 0, 0, 0, 0, 1 } })

  A ColourMatrix that inverts the alpha component, leaving the other components intact.
- static ColourMatrix InvertedAlphaShift = new ColourMatrix(new double[,] { { 1, 0, 0, -1, 1 }, { 0, 1, 0, -1, 1 }, { 0, 0, 1, -1, 1 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } })

  A ColourMatrix that shifts every colour component by an amount corresponding to the inverted alpha value. The alpha component is left intact.

Properties

- double R1 [get, set]

  The coefficient relating the R component of the output colour to the R component of the input colour.
- double R2 [get, set]

  The coefficient relating the R component of the output colour to the G component of the input colour.
- double R3 [get, set]

  The coefficient relating the R component of the output colour to the B component of the input colour.
- double R4 [get, set]

  The coefficient relating the R component of the output colour to the A component of the input colour.
- double R5 [get, set]

  The bias (translation) applied to the R component of the output colour.
- double G1 [get, set]

  The coefficient relating the G component of the output colour to the R component of the input colour.
- double G2 [get, set]

  The coefficient relating the G component of the output colour to the G component of the input colour.
- double G3 [get, set]

  The coefficient relating the G component of the output colour to the B component of the input colour.
- double G4 [get, set]

  The coefficient relating the G component of the output colour to the A component of the input colour.
- double G5 [get, set]

  The bias (translation) applied to the R component of the output colour.
- double B1 [get, set]

  The coefficient relating the B component of the output colour to the R component of the input colour.
- double B2 [get, set]

  The coefficient relating the B component of the output colour to the G component of the input colour.
- double B3 [get, set]

  The coefficient relating the B component of the output colour to the B component of the input colour.
- double B4 [get, set]

  The coefficient relating the B component of the output colour to the A component of the input colour.
- double B5 [get, set]

  The bias (translation) applied to the B component of the output colour.
- double A1 [get, set]

  The coefficient relating the A component of the output colour to the R component of the input colour.
- double A2 [get, set]

  The coefficient relating the A component of the output colour to the G component of the input colour.
- double A3 [get, set]

  The coefficient relating the A component of the output colour to the B component of the input colour.
- double A4 [get, set]

  The coefficient relating the A component of the output colour to the A component of the input colour.
- double A5 [get, set]

  The bias (translation) applied to the A component of the output colour.
- double this[int y, int x] [get]

  Gets or sets the requested element of the matrix. Elements of the last row of the matrix can be read, but not set.

7.13.1 Detailed Description

Represents a colour transformation matrix.

Definition at line 26 of file ColourMatrixFilter.cs.

7.13.2 Constructor & Destructor Documentation

7.13.2.1 ColourMatrix()

```
VectSharp.Filters.ColourMatrix.ColourMatrix (
            double matrix[,] )
```

Creates a new ColourMatrix with the specified coefficients.

Parameters

matrix	The coefficients of the ColourMatrix.

Definition at line 454 of file ColourMatrixFilter.cs.

7.13.3 Member Function Documentation

7.13.3.1 Apply() [1/5]

```
void VectSharp.Filters.ColourMatrix.Apply (
            byte R,
            byte G,
            byte B,
            byte A,
            out byte r,
            out byte g,
            out byte b,
            out byte a )
```

Applies the ColourMatrix to the specified colour, represented as four bytes, and stores the resulting colour in the specified output bytes.

Parameters

R	The R component of the input colour.
G	The G component of the input colour.
B	The B component of the input colour.
A	The A component of the input colour.
r	After this method returns, this will contain the R component of the output colour.
g	After this method returns, this will contain the G component of the output colour.
b	After this method returns, this will contain the B component of the output colour.
a	After this method returns, this will contain the A component of the output colour.

Definition at line 538 of file ColourMatrixFilter.cs.

7.13.3.2 Apply() [2/5]

```
void VectSharp.Filters.ColourMatrix.Apply (
            byte R,
            byte G,
            byte B,
            out byte r,
            out byte g,
            out byte b )
```

Applies the ColourMatrix to the specified colour, represented as three bytes, and stores the resulting colour in the specified output bytes.

Parameters

R	The R component of the input colour.
G	The G component of the input colour.
B	The B component of the input colour.
r	After this method returns, this will contain the R component of the output colour.
g	After this method returns, this will contain the G component of the output colour.
b	After this method returns, this will contain the B component of the output colour.

Definition at line 556 of file ColourMatrixFilter.cs.

7.13.3.3 Apply() [3/5]

```
Colour VectSharp.Filters.ColourMatrix.Apply (
            Colour colour )
```

Applies the ColourMatrix to the specified Colour.

Parameters

colour	The Colour to which the ColourMatrix should be applied.

Returns

The result of applying the ColourMatrix to the specified colour.

Definition at line 470 of file ColourMatrixFilter.cs.

7.13.3.4 Apply() [4/5]

```
void VectSharp.Filters.ColourMatrix.Apply (
            ref byte R,
            ref byte G,
            ref byte B )
```

Applies the ColourMatrix to the specified colour, represented as three bytes, and stores the resulting colour in the same variables as the original RGB values.

Parameters

R	The R component of the input colour. After this method returns, this will contain the R component of the output colour.
G	The G component of the input colour. After this method returns, this will contain the G component of the output colour.
B	The B component of the input colour. After this method returns, this will contain the B component of the output colour.

Definition at line 515 of file ColourMatrixFilter.cs.

7.13.3.5 Apply() [5/5]

```
void VectSharp.Filters.ColourMatrix.Apply (
            ref byte R,
            ref byte G,
            ref byte B,
            ref byte A )
```

Applies the ColourMatrix to the specified colour, represented as four bytes, and stores the resulting colour in the same variables as the original RGBA values.

Parameters

R	The R component of the input colour. After this method returns, this will contain the R component of the output colour.
G	The G component of the input colour. After this method returns, this will contain the G component of the output colour.
B	The B component of the input colour. After this method returns, this will contain the B component of the output colour.
A	The A component of the input colour. After this method returns, this will contain the A component of the output colour.

Definition at line 495 of file ColourMatrixFilter.cs.

7.13.3.6 LuminanceToAlpha()

```
static ColourMatrix VectSharp.Filters.ColourMatrix.LuminanceToAlpha (
            bool preserveColour = false )  [static]
```

Creates a ColourMatrix that transforms the alpha value of the colour it is applied to into a value depending on the luminance of the input colour.

Parameters

preserveColour	If this is `true`, the values of the red, green and blue components of the input colour are preserved in the output colour. If this is false, the output colour will always be black.

Returns

A ColourMatrix that transforms the alpha value of the colour it is applied to into a value depending on the luminance of the input colour.

Definition at line 404 of file ColourMatrixFilter.cs.

7.13.3.7 LuminanceToColour()

```
static ColourMatrix VectSharp.Filters.ColourMatrix.LuminanceToColour (
            Colour colour,
            bool useAlpha = false )  [static]
```

Creates a ColourMatrix that turns every colour to which it is applied into a shade of the specified *colour* . The brightness of the output colour depends on the luminance of the input colour.

Parameters

colour	The colour whose shades will be produced by the ColourMatrix.
useAlpha	If this is `true`, the transformation will also be applied to the alpha channel. If this is false, the alpha value of the input pixels is preserved.

Returns

A ColourMatrix that turns every colour to which it is applied into a shade of the specified *colour* .

Definition at line 387 of file ColourMatrixFilter.cs.

7.13.3.8 operator∗()

```
static ColourMatrix VectSharp.Filters.ColourMatrix.operator* (
            ColourMatrix matrix1,
            ColourMatrix matrix2 )  [static]
```

Concatenates two matrices. The resulting ColourMatrix is equivalent to first applying *matrix2* , and then *matrix1* .

Parameters

matrix1	The matrix that acts second.
matrix2	The matrix that acts first.

Returns

A ColourMatrix equivalent to first applying *matrix2* , and then *matrix1* .

Definition at line 432 of file ColourMatrixFilter.cs.

7.13.3.9 ToColour()

```
static ColourMatrix VectSharp.Filters.ColourMatrix.ToColour (
            Colour colour,
            bool useAlpha = false )  [static]
```

Creates a ColourMatrix that turns every colour to which it is applied into the specified *colour* .

Parameters

colour	The colour that will be produced by the ColourMatrix.
useAlpha	If this is `true`, all output pixels will have the same alpha value as the supplied *colour* . If this is false, the alpha value of the input pixels is preserved.

Returns

A ColourMatrix that turns every colour to which it is applied into the specified *colour* .

Definition at line 369 of file ColourMatrixFilter.cs.

7.13.3.10 WithAlpha()

```
ColourMatrix VectSharp.Filters.ColourMatrix.WithAlpha (
            double alpha )
```

Creates a new ColourMatrix whose alpha coefficients are multiplied by the specified value.

Parameters

alpha	The value that will be used to multiply all the alpha coefficients of the ColourMatrix.

Returns

A new ColourMatrix whose alpha coefficients have been multiplied by the specified value.

Definition at line 421 of file ColourMatrixFilter.cs.

7.13.4 Member Data Documentation

7.13.4.1 AlphaInversion

```
ColourMatrix VectSharp.Filters.ColourMatrix.AlphaInversion = new ColourMatrix(new double[,] {
{ 1, 0, 0, 0, 0 }, { 0, 1, 0, 0, 0 }, { 0, 0, 1, 0, 0 }, { 0, 0, 0, -1, 1 }, { 0, 0, 0, 0, 1 }
})  [static]
```

A ColourMatrix that inverts the alpha component, leaving the other components intact.

Definition at line 356 of file ColourMatrixFilter.cs.

7.13.4.2 GreyScale

```
ColourMatrix VectSharp.Filters.ColourMatrix.GreyScale = new ColourMatrix(new double[,] { {
0.2126, 0.7152, 0.0722, 0, 0 }, { 0.2126, 0.7152, 0.0722, 0, 0 }, { 0.2126, 0.7152, 0.0722, 0,
0 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } })  [static]
```

A ColourMatrix that transforms every colour in a shade of grey with approximately the same luminance.

Definition at line 341 of file ColourMatrixFilter.cs.

7.13.4.3 Identity

```
ColourMatrix VectSharp.Filters.ColourMatrix.Identity = new ColourMatrix(new double[,] { { 1,
0, 0, 0, 0 }, { 0, 1, 0, 0, 0 }, { 0, 0, 1, 0, 0 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } })
[static]
```

A ColourMatrix that whose output colour is always the same as the input colour.

Definition at line 336 of file ColourMatrixFilter.cs.

7.13.4.4 Inversion

```
ColourMatrix VectSharp.Filters.ColourMatrix.Inversion = new ColourMatrix(new double[,] { { -1,
0, 0, 0, 1 }, { 0, -1, 0, 0, 1 }, { 0, 0, -1, 0, 1 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } })
[static]
```

A ColourMatrix that inverts every colour, leaving the alpha component intact.

Definition at line 351 of file ColourMatrixFilter.cs.

7.13.4.5 InvertedAlphaShift

```
ColourMatrix VectSharp.Filters.ColourMatrix.InvertedAlphaShift = new ColourMatrix(new double[,]
{ { 1, 0, 0, -1, 1 }, { 0, 1, 0, -1, 1 }, { 0, 0, 1, -1, 1 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0,
1 } })  [static]
```

A ColourMatrix that shifts every colour component by an amount corresponding to the inverted alpha value. The alpha component is left intact.

Definition at line 361 of file ColourMatrixFilter.cs.

7.13.4.6 Pastel

```
ColourMatrix VectSharp.Filters.ColourMatrix.Pastel = new ColourMatrix(new double[,] { { 0.75,
0.25, 0.25, 0, 0 }, { 0.25, 0.75, 0.25, 0, 0 }, { 0.25, 0.25, 0.75, 0, 0 }, { 0, 0, 0, 1, 0 },
{ 0, 0, 0, 0, 1 } })  [static]
```

A ColourMatrix producing a "pastel" (desaturation) effect.

Definition at line 346 of file ColourMatrixFilter.cs.

7.13.5 Property Documentation

7.13.5.1 A1

```
double VectSharp.Filters.ColourMatrix.A1  [get], [set]
```

The coefficient relating the A component of the output colour to the R component of the input colour.

Definition at line 106 of file ColourMatrixFilter.cs.

7.13.5.2 A2

```
double VectSharp.Filters.ColourMatrix.A2  [get], [set]
```

The coefficient relating the A component of the output colour to the G component of the input colour.

Definition at line 111 of file ColourMatrixFilter.cs.

7.13.5.3 A3

```
double VectSharp.Filters.ColourMatrix.A3  [get], [set]
```

The coefficient relating the A component of the output colour to the B component of the input colour.

Definition at line 116 of file ColourMatrixFilter.cs.

7.13.5.4 A4

```
double VectSharp.Filters.ColourMatrix.A4  [get], [set]
```

The coefficient relating the A component of the output colour to the A component of the input colour.

Definition at line 121 of file ColourMatrixFilter.cs.

7.13.5.5 A5

```
double VectSharp.Filters.ColourMatrix.A5  [get], [set]
```

The bias (translation) applied to the A component of the output colour.

Definition at line 126 of file ColourMatrixFilter.cs.

7.13.5.6 B1

```
double VectSharp.Filters.ColourMatrix.B1  [get], [set]
```

The coefficient relating the B component of the output colour to the R component of the input colour.

Definition at line 81 of file ColourMatrixFilter.cs.

7.13.5.7 B2

```
double VectSharp.Filters.ColourMatrix.B2  [get], [set]
```

The coefficient relating the B component of the output colour to the G component of the input colour.

Definition at line 86 of file ColourMatrixFilter.cs.

7.13.5.8 B3

```
double VectSharp.Filters.ColourMatrix.B3  [get], [set]
```

The coefficient relating the B component of the output colour to the B component of the input colour.

Definition at line 91 of file ColourMatrixFilter.cs.

7.13.5.9 B4

```
double VectSharp.Filters.ColourMatrix.B4  [get], [set]
```

The coefficient relating the B component of the output colour to the A component of the input colour.

Definition at line 96 of file ColourMatrixFilter.cs.

7.13.5.10 B5

```
double VectSharp.Filters.ColourMatrix.B5  [get], [set]
```

The bias (translation) applied to the B component of the output colour.

Definition at line 101 of file ColourMatrixFilter.cs.

7.13.5.11 G1

```
double VectSharp.Filters.ColourMatrix.G1  [get], [set]
```

The coefficient relating the G component of the output colour to the R component of the input colour.

Definition at line 56 of file ColourMatrixFilter.cs.

7.13.5.12 G2

```
double VectSharp.Filters.ColourMatrix.G2  [get], [set]
```

The coefficient relating the G component of the output colour to the G component of the input colour.

Definition at line 61 of file ColourMatrixFilter.cs.

7.13.5.13 G3

```
double VectSharp.Filters.ColourMatrix.G3  [get], [set]
```

The coefficient relating the G component of the output colour to the B component of the input colour.

Definition at line 66 of file ColourMatrixFilter.cs.

7.13.5.14 G4

```
double VectSharp.Filters.ColourMatrix.G4  [get], [set]
```

The coefficient relating the G component of the output colour to the A component of the input colour.

Definition at line 71 of file ColourMatrixFilter.cs.

7.13.5.15 G5

```
double VectSharp.Filters.ColourMatrix.G5  [get], [set]
```

The bias (translation) applied to the R component of the output colour.

Definition at line 76 of file ColourMatrixFilter.cs.

7.13.5.16 R1

```
double VectSharp.Filters.ColourMatrix.R1  [get], [set]
```

The coefficient relating the R component of the output colour to the R component of the input colour.

Definition at line 31 of file ColourMatrixFilter.cs.

7.13.5.17 R2

```
double VectSharp.Filters.ColourMatrix.R2  [get], [set]
```

The coefficient relating the R component of the output colour to the G component of the input colour.

Definition at line 36 of file ColourMatrixFilter.cs.

7.13.5.18 R3

```
double VectSharp.Filters.ColourMatrix.R3  [get], [set]
```

The coefficient relating the R component of the output colour to the B component of the input colour.

Definition at line 41 of file ColourMatrixFilter.cs.

7.13.5.19 R4

```
double VectSharp.Filters.ColourMatrix.R4  [get], [set]
```

The coefficient relating the R component of the output colour to the A component of the input colour.

Definition at line 46 of file ColourMatrixFilter.cs.

7.13.5.20 R5

```
double VectSharp.Filters.ColourMatrix.R5  [get], [set]
```

The bias (translation) applied to the R component of the output colour.

Definition at line 51 of file ColourMatrixFilter.cs.

7.13.5.21 this[int y, int x]

```
double VectSharp.Filters.ColourMatrix.this[int y, int x]  [get]
```

Gets or sets the requested element of the matrix. Elements of the last row of the matrix can be read, but not set.

Parameters

y	The row of the matrix.
x	The column of the matrix.

Returns

The requested element of the matrix.

Exceptions

ArgumentOutOfRangeException	An attempt has been made to access an element out of the bounds of the matrix.

Definition at line 135 of file ColourMatrixFilter.cs.

The documentation for this class was generated from the following file:

- VectSharp/Filters/ColourMatrixFilter.cs

7.14 VectSharp.Filters.ColourMatrixFilter Class Reference

Represents a filter that applies a Filters.ColourMatrix to the colours of the image.

Inheritance diagram for VectSharp.Filters.ColourMatrixFilter:

Public Member Functions

- ColourMatrixFilter (ColourMatrix colorMatrix)

  Creates a new ColourMatrixFilter with the specified Filters.ColourMatrix.
- RasterImage Filter (RasterImage image, double scale)

  Applies the filter to a RasterImage.

Parameters

image	*The RasterImage to which the filter will be applied.*
scale	*The scale of the image with respect to the filter.*

Returns

A new RasterImage containing the filtered image. The source image is left unaltered.

Properties

- ColourMatrix ColourMatrix `[get]`

  The Filters.ColourMatrix that is applied by this filter.
- Point TopLeftMargin = new Point() `[get]`

  Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.
- Point BottomRightMargin = new Point() `[get]`

  Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

7.14.1 Detailed Description

Represents a filter that applies a Filters.ColourMatrix to the colours of the image.

Definition at line 567 of file ColourMatrixFilter.cs.

7.14.2 Constructor & Destructor Documentation

7.14.2.1 ColourMatrixFilter()

```
VectSharp.Filters.ColourMatrixFilter.ColourMatrixFilter (
            ColourMatrix colorMatrix )
```

Creates a new ColourMatrixFilter with the specified Filters.ColourMatrix.

Parameters

colorMatrix	The Filters.ColourMatrix that will be applied by the filter.

Definition at line 584 of file ColourMatrixFilter.cs.

7.14.3 Member Function Documentation

7.14.3.1 Filter()

```
RasterImage VectSharp.Filters.ColourMatrixFilter.Filter (
            RasterImage image,
            double scale )
```

Applies the filter to a RasterImage.

Parameters

image	The RasterImage to which the filter will be applied.
scale	The scale of the image with respect to the filter.

Returns

A new RasterImage containing the filtered image. The source *image* is left unaltered.

Implements VectSharp.Filters.ILocationInvariantFilter.

Definition at line 590 of file ColourMatrixFilter.cs.

7.14.4 Property Documentation

7.14.4.1 BottomRightMargin

```
Point VectSharp.Filters.ColourMatrixFilter.BottomRightMargin = new Point()  [get]
```

Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 578 of file ColourMatrixFilter.cs.

7.14.4.2 ColourMatrix

```
ColourMatrix VectSharp.Filters.ColourMatrixFilter.ColourMatrix  [get]
```

The Filters.ColourMatrix that is applied by this filter.

Definition at line 572 of file ColourMatrixFilter.cs.

7.14.4.3 TopLeftMargin

```
Point VectSharp.Filters.ColourMatrixFilter.TopLeftMargin = new Point()  [get]
```

Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 575 of file ColourMatrixFilter.cs.

The documentation for this class was generated from the following file:

- VectSharp/Filters/ColourMatrixFilter.cs

7.15 VectSharp.Colours Class Reference

Standard colours.

Static Public Attributes

- static Colour Black = Colour.FromRgb(0, 0, 0)

  Black #000000
- static Colour Navy = Colour.FromRgb(0, 0, 128)

  Navy #000080
- static Colour DarkBlue = Colour.FromRgb(0, 0, 139)

  DarkBlue #00008B
- static Colour MediumBlue = Colour.FromRgb(0, 0, 205)

  MediumBlue #0000CD
- static Colour Blue = Colour.FromRgb(0, 0, 255)

  Blue #0000FF
- static Colour DarkGreen = Colour.FromRgb(0, 100, 0)

  DarkGreen #006400
- static Colour Green = Colour.FromRgb(0, 128, 0)

  Green #008000
- static Colour Teal = Colour.FromRgb(0, 128, 128)

  Teal #008080
- static Colour DarkCyan = Colour.FromRgb(0, 139, 139)

  DarkCyan #008B8B
- static Colour DeepSkyBlue = Colour.FromRgb(0, 191, 255)

  DeepSkyBlue #00BFFF
- static Colour DarkTurquoise = Colour.FromRgb(0, 206, 209)

  DarkTurquoise #00CED1
- static Colour MediumSpringGreen = Colour.FromRgb(0, 250, 154)

  MediumSpringGreen #00FA9A
- static Colour Lime = Colour.FromRgb(0, 255, 0)

  Lime #00FF00
- static Colour SpringGreen = Colour.FromRgb(0, 255, 127)

  SpringGreen #00FF7F
- static Colour Aqua = Colour.FromRgb(0, 255, 255)

  Aqua #00FFFF
- static Colour Cyan = Colour.FromRgb(0, 255, 255)

  Cyan #00FFFF
- static Colour MidnightBlue = Colour.FromRgb(25, 25, 112)

  MidnightBlue #191970
- static Colour DodgerBlue = Colour.FromRgb(30, 144, 255)

  DodgerBlue #1E90FF
- static Colour LightSeaGreen = Colour.FromRgb(32, 178, 170)

  LightSeaGreen #20B2AA
- static Colour ForestGreen = Colour.FromRgb(34, 139, 34)

  ForestGreen #228B22
- static Colour SeaGreen = Colour.FromRgb(46, 139, 87)

  SeaGreen #2E8B57
- static Colour DarkSlateGray = Colour.FromRgb(47, 79, 79)

DarkSlateGray #2F4F4F

- static Colour DarkSlateGrey = Colour.FromRgb(47, 79, 79)

  DarkSlateGrey #2F4F4F
- static Colour LimeGreen = Colour.FromRgb(50, 205, 50)

  LimeGreen #32CD32
- static Colour MediumSeaGreen = Colour.FromRgb(60, 179, 113)

  MediumSeaGreen #3CB371
- static Colour Turquoise = Colour.FromRgb(64, 224, 208)

  Turquoise #40E0D0
- static Colour RoyalBlue = Colour.FromRgb(65, 105, 225)

  RoyalBlue #4169E1
- static Colour SteelBlue = Colour.FromRgb(70, 130, 180)

  SteelBlue #4682B4
- static Colour DarkSlateBlue = Colour.FromRgb(72, 61, 139)

  DarkSlateBlue #483D8B
- static Colour MediumTurquoise = Colour.FromRgb(72, 209, 204)

  MediumTurquoise #48D1CC
- static Colour Indigo = Colour.FromRgb(75, 0, 130)

  Indigo #4B0082
- static Colour DarkOliveGreen = Colour.FromRgb(85, 107, 47)

  DarkOliveGreen #556B2F
- static Colour CadetBlue = Colour.FromRgb(95, 158, 160)

  CadetBlue #5F9EA0
- static Colour CornflowerBlue = Colour.FromRgb(100, 149, 237)

  CornflowerBlue #6495ED
- static Colour RebeccaPurple = Colour.FromRgb(102, 51, 153)

  RebeccaPurple #663399
- static Colour MediumAquaMarine = Colour.FromRgb(102, 205, 170)

  MediumAquaMarine #66CDAA
- static Colour DimGray = Colour.FromRgb(105, 105, 105)

  DimGray #696969
- static Colour DimGrey = Colour.FromRgb(105, 105, 105)

  DimGrey #696969
- static Colour SlateBlue = Colour.FromRgb(106, 90, 205)

  SlateBlue #6A5ACD
- static Colour OliveDrab = Colour.FromRgb(107, 142, 35)

  OliveDrab #6B8E23
- static Colour SlateGray = Colour.FromRgb(112, 128, 144)

  SlateGray #708090
- static Colour SlateGrey = Colour.FromRgb(112, 128, 144)

  SlateGrey #708090
- static Colour LightSlateGray = Colour.FromRgb(119, 136, 153)

  LightSlateGray #778899
- static Colour LightSlateGrey = Colour.FromRgb(119, 136, 153)

  LightSlateGrey #778899
- static Colour MediumSlateBlue = Colour.FromRgb(123, 104, 238)

  MediumSlateBlue #7B68EE
- static Colour LawnGreen = Colour.FromRgb(124, 252, 0)

  LawnGreen #7CFC00
- static Colour Chartreuse = Colour.FromRgb(127, 255, 0)

  Chartreuse #7FFF00

- static Colour Aquamarine = Colour.FromRgb(127, 255, 212)

  Aquamarine #7FFFD4
- static Colour Maroon = Colour.FromRgb(128, 0, 0)

  Maroon #800000
- static Colour Purple = Colour.FromRgb(128, 0, 128)

  Purple #800080
- static Colour Olive = Colour.FromRgb(128, 128, 0)

  Olive #808000
- static Colour Gray = Colour.FromRgb(128, 128, 128)

  Gray #808080
- static Colour Grey = Colour.FromRgb(128, 128, 128)

  Grey #808080
- static Colour SkyBlue = Colour.FromRgb(135, 206, 235)

  SkyBlue #87CEEB
- static Colour LightSkyBlue = Colour.FromRgb(135, 206, 250)

  LightSkyBlue #87CEFA
- static Colour BlueViolet = Colour.FromRgb(138, 43, 226)

  BlueViolet #8A2BE2
- static Colour DarkRed = Colour.FromRgb(139, 0, 0)

  DarkRed #8B0000
- static Colour DarkMagenta = Colour.FromRgb(139, 0, 139)

  DarkMagenta #8B008B
- static Colour SaddleBrown = Colour.FromRgb(139, 69, 19)

  SaddleBrown #8B4513
- static Colour DarkSeaGreen = Colour.FromRgb(143, 188, 143)

  DarkSeaGreen #8FBC8F
- static Colour LightGreen = Colour.FromRgb(144, 238, 144)

  LightGreen #90EE90
- static Colour MediumPurple = Colour.FromRgb(147, 112, 219)

  MediumPurple #9370DB
- static Colour DarkViolet = Colour.FromRgb(148, 0, 211)

  DarkViolet #9400D3
- static Colour PaleGreen = Colour.FromRgb(152, 251, 152)

  PaleGreen #98FB98
- static Colour DarkOrchid = Colour.FromRgb(153, 50, 204)

  DarkOrchid #9932CC
- static Colour YellowGreen = Colour.FromRgb(154, 205, 50)

  YellowGreen #9ACD32
- static Colour Sienna = Colour.FromRgb(160, 82, 45)

  Sienna #A0522D
- static Colour Brown = Colour.FromRgb(165, 42, 42)

  Brown #A52A2A
- static Colour DarkGray = Colour.FromRgb(169, 169, 169)

  DarkGray #A9A9A9
- static Colour DarkGrey = Colour.FromRgb(169, 169, 169)

  DarkGrey #A9A9A9
- static Colour LightBlue = Colour.FromRgb(173, 216, 230)

  LightBlue #ADD8E6
- static Colour GreenYellow = Colour.FromRgb(173, 255, 47)

  GreenYellow #ADFF2F
- static Colour PaleTurquoise = Colour.FromRgb(175, 238, 238)

PaleTurquoise #AFEEEE

- static Colour LightSteelBlue = Colour.FromRgb(176, 196, 222)

  LightSteelBlue #B0C4DE
- static Colour PowderBlue = Colour.FromRgb(176, 224, 230)

  PowderBlue #B0E0E6
- static Colour FireBrick = Colour.FromRgb(178, 34, 34)

  FireBrick #B22222
- static Colour DarkGoldenRod = Colour.FromRgb(184, 134, 11)

  DarkGoldenRod #B8860B
- static Colour MediumOrchid = Colour.FromRgb(186, 85, 211)

  MediumOrchid #BA55D3
- static Colour RosyBrown = Colour.FromRgb(188, 143, 143)

  RosyBrown #BC8F8F
- static Colour DarkKhaki = Colour.FromRgb(189, 183, 107)

  DarkKhaki #BDB76B
- static Colour Silver = Colour.FromRgb(192, 192, 192)

  Silver #C0C0C0
- static Colour MediumVioletRed = Colour.FromRgb(199, 21, 133)

  MediumVioletRed #C71585
- static Colour IndianRed = Colour.FromRgb(205, 92, 92)

  IndianRed #CD5C5C
- static Colour Peru = Colour.FromRgb(205, 133, 63)

  Peru #CD853F
- static Colour Chocolate = Colour.FromRgb(210, 105, 30)

  Chocolate #D2691E
- static Colour Tan = Colour.FromRgb(210, 180, 140)

  Tan #D2B48C
- static Colour LightGray = Colour.FromRgb(211, 211, 211)

  LightGray #D3D3D3
- static Colour LightGrey = Colour.FromRgb(211, 211, 211)

  LightGrey #D3D3D3
- static Colour Thistle = Colour.FromRgb(216, 191, 216)

  Thistle #D8BFD8
- static Colour Orchid = Colour.FromRgb(218, 112, 214)

  Orchid #DA70D6
- static Colour GoldenRod = Colour.FromRgb(218, 165, 32)

  GoldenRod #DAA520
- static Colour PaleVioletRed = Colour.FromRgb(219, 112, 147)

  PaleVioletRed #DB7093
- static Colour Crimson = Colour.FromRgb(220, 20, 60)

  Crimson #DC143C
- static Colour Gainsboro = Colour.FromRgb(220, 220, 220)

  Gainsboro #DCDCDC
- static Colour Plum = Colour.FromRgb(221, 160, 221)

  Plum #DDA0DD
- static Colour BurlyWood = Colour.FromRgb(222, 184, 135)

  BurlyWood #DEB887
- static Colour LightCyan = Colour.FromRgb(224, 255, 255)

  LightCyan #E0FFFF
- static Colour Lavender = Colour.FromRgb(230, 230, 250)

  Lavender #E6E6FA

- static Colour DarkSalmon = Colour.FromRgb(233, 150, 122)

  DarkSalmon #E9967A
- static Colour Violet = Colour.FromRgb(238, 130, 238)

  Violet #EE82EE
- static Colour PaleGoldenRod = Colour.FromRgb(238, 232, 170)

  PaleGoldenRod #EEE8AA
- static Colour LightCoral = Colour.FromRgb(240, 128, 128)

  LightCoral #F08080
- static Colour Khaki = Colour.FromRgb(240, 230, 140)

  Khaki #F0E68C
- static Colour AliceBlue = Colour.FromRgb(240, 248, 255)

  AliceBlue #F0F8FF
- static Colour HoneyDew = Colour.FromRgb(240, 255, 240)

  HoneyDew #F0FFF0
- static Colour Azure = Colour.FromRgb(240, 255, 255)

  Azure #F0FFFF
- static Colour SandyBrown = Colour.FromRgb(244, 164, 96)

  SandyBrown #F4A460
- static Colour Wheat = Colour.FromRgb(245, 222, 179)

  Wheat #F5DEB3
- static Colour Beige = Colour.FromRgb(245, 245, 220)

  Beige #F5F5DC
- static Colour WhiteSmoke = Colour.FromRgb(245, 245, 245)

  WhiteSmoke #F5F5F5
- static Colour MintCream = Colour.FromRgb(245, 255, 250)

  MintCream #F5FFFA
- static Colour GhostWhite = Colour.FromRgb(248, 248, 255)

  GhostWhite #F8F8FF
- static Colour Salmon = Colour.FromRgb(250, 128, 114)

  Salmon #FA8072
- static Colour AntiqueWhite = Colour.FromRgb(250, 235, 215)

  AntiqueWhite #FAEBD7
- static Colour Linen = Colour.FromRgb(250, 240, 230)

  Linen #FAF0E6
- static Colour LightGoldenRodYellow = Colour.FromRgb(250, 250, 210)

  LightGoldenRodYellow #FAFAD2
- static Colour OldLace = Colour.FromRgb(253, 245, 230)

  OldLace #FDF5E6
- static Colour Red = Colour.FromRgb(255, 0, 0)

  Red #FF0000
- static Colour Fuchsia = Colour.FromRgb(255, 0, 255)

  Fuchsia #FF00FF
- static Colour Magenta = Colour.FromRgb(255, 0, 255)

  Magenta #FF00FF
- static Colour DeepPink = Colour.FromRgb(255, 20, 147)

  DeepPink #FF1493
- static Colour OrangeRed = Colour.FromRgb(255, 69, 0)

  OrangeRed #FF4500
- static Colour Tomato = Colour.FromRgb(255, 99, 71)

  Tomato #FF6347
- static Colour HotPink = Colour.FromRgb(255, 105, 180)

HotPink #FF69B4

- static Colour Coral = Colour.FromRgb(255, 127, 80)

  Coral #FF7F50
- static Colour DarkOrange = Colour.FromRgb(255, 140, 0)

  DarkOrange #FF8C00
- static Colour LightSalmon = Colour.FromRgb(255, 160, 122)

  LightSalmon #FFA07A
- static Colour Orange = Colour.FromRgb(255, 165, 0)

  Orange #FFA500
- static Colour LightPink = Colour.FromRgb(255, 182, 193)

  LightPink #FFB6C1
- static Colour Pink = Colour.FromRgb(255, 192, 203)

  Pink #FFC0CB
- static Colour Gold = Colour.FromRgb(255, 215, 0)

  Gold #FFD700
- static Colour PeachPuff = Colour.FromRgb(255, 218, 185)

  PeachPuff #FFDAB9
- static Colour NavajoWhite = Colour.FromRgb(255, 222, 173)

  NavajoWhite #FFDEAD
- static Colour Moccasin = Colour.FromRgb(255, 228, 181)

  Moccasin #FFE4B5
- static Colour Bisque = Colour.FromRgb(255, 228, 196)

  Bisque #FFE4C4
- static Colour MistyRose = Colour.FromRgb(255, 228, 225)

  MistyRose #FFE4E1
- static Colour BlanchedAlmond = Colour.FromRgb(255, 235, 205)

  BlanchedAlmond #FFEBCD
- static Colour PapayaWhip = Colour.FromRgb(255, 239, 213)

  PapayaWhip #FFEFD5
- static Colour LavenderBlush = Colour.FromRgb(255, 240, 245)

  LavenderBlush #FFF0F5
- static Colour SeaShell = Colour.FromRgb(255, 245, 238)

  SeaShell #FFF5EE
- static Colour Cornsilk = Colour.FromRgb(255, 248, 220)

  Cornsilk #FFF8DC
- static Colour LemonChiffon = Colour.FromRgb(255, 250, 205)

  LemonChiffon #FFFACD
- static Colour FloralWhite = Colour.FromRgb(255, 250, 240)

  FloralWhite #FFFAF0
- static Colour Snow = Colour.FromRgb(255, 250, 250)

  Snow #FFFAFA
- static Colour Yellow = Colour.FromRgb(255, 255, 0)

  Yellow #FFFF00
- static Colour LightYellow = Colour.FromRgb(255, 255, 224)

  LightYellow #FFFFE0
- static Colour Ivory = Colour.FromRgb(255, 255, 240)

  Ivory #FFFFF0
- static Colour White = Colour.FromRgb(255, 255, 255)

  White #FFFFFF

7.15.1 Detailed Description

Standard colours.

Definition at line 182 of file StandardColours.cs.

7.15.2 Member Data Documentation

7.15.2.1 AliceBlue

```
Colour VectSharp.Colours.AliceBlue = Colour.FromRgb(240, 248, 255)  [static]
```

AliceBlue #F0F8FF

Definition at line 599 of file StandardColours.cs.

7.15.2.2 AntiqueWhite

```
Colour VectSharp.Colours.AntiqueWhite = Colour.FromRgb(250, 235, 215)  [static]
```

AntiqueWhite #FAEBD7

Definition at line 639 of file StandardColours.cs.

7.15.2.3 Aqua

```
Colour VectSharp.Colours.Aqua = Colour.FromRgb(0, 255, 255)  [static]
```

Aqua #00FFFF

Definition at line 243 of file StandardColours.cs.

7.15.2.4 Aquamarine

```
Colour VectSharp.Colours.Aquamarine = Colour.FromRgb(127, 255, 212)  [static]
```

Aquamarine #7FFFD4

Definition at line 375 of file StandardColours.cs.

7.15.2.5 Azure

```
Colour VectSharp.Colours.Azure = Colour.FromRgb(240, 255, 255)  [static]
```

Azure #F0FFFF

Definition at line 607 of file StandardColours.cs.

7.15.2.6 Beige

```
Colour VectSharp.Colours.Beige = Colour.FromRgb(245, 245, 220)  [static]
```

Beige #F5F5DC

Definition at line 619 of file StandardColours.cs.

7.15.2.7 Bisque

```
Colour VectSharp.Colours.Bisque = Colour.FromRgb(255, 228, 196)  [static]
```

Bisque #FFE4C4

Definition at line 723 of file StandardColours.cs.

7.15.2.8 Black

```
Colour VectSharp.Colours.Black = Colour.FromRgb(0, 0, 0)  [static]
```

Black #000000

Definition at line 187 of file StandardColours.cs.

7.15.2.9 BlanchedAlmond

```
Colour VectSharp.Colours.BlanchedAlmond = Colour.FromRgb(255, 235, 205)  [static]
```

BlanchedAlmond #FFEBCD

Definition at line 731 of file StandardColours.cs.

7.15.2.10 Blue

```
Colour VectSharp.Colours.Blue = Colour.FromRgb(0, 0, 255)  [static]
```

Blue #0000FF

Definition at line 203 of file StandardColours.cs.

7.15.2.11 BlueViolet

```
Colour VectSharp.Colours.BlueViolet = Colour.FromRgb(138, 43, 226)  [static]
```

BlueViolet #8A2BE2

Definition at line 407 of file StandardColours.cs.

7.15.2.12 Brown

```
Colour VectSharp.Colours.Brown = Colour.FromRgb(165, 42, 42)  [static]
```

Brown #A52A2A

Definition at line 455 of file StandardColours.cs.

7.15.2.13 BurlyWood

```
Colour VectSharp.Colours.BurlyWood = Colour.FromRgb(222, 184, 135)  [static]
```

BurlyWood #DEB887

Definition at line 567 of file StandardColours.cs.

7.15.2.14 CadetBlue

```
Colour VectSharp.Colours.CadetBlue = Colour.FromRgb(95, 158, 160)  [static]
```

CadetBlue #5F9EA0

Definition at line 315 of file StandardColours.cs.

7.15.2.15 Chartreuse

```
Colour VectSharp.Colours.Chartreuse = Colour.FromRgb(127, 255, 0)  [static]
```

Chartreuse #7FFF00

Definition at line 371 of file StandardColours.cs.

7.15.2.16 Chocolate

```
Colour VectSharp.Colours.Chocolate = Colour.FromRgb(210, 105, 30)  [static]
```

Chocolate #D2691E

Definition at line 523 of file StandardColours.cs.

7.15.2.17 Coral

```
Colour VectSharp.Colours.Coral = Colour.FromRgb(255, 127, 80)  [static]
```

Coral #FF7F50

Definition at line 683 of file StandardColours.cs.

7.15.2.18 CornflowerBlue

```
Colour VectSharp.Colours.CornflowerBlue = Colour.FromRgb(100, 149, 237)  [static]
```

CornflowerBlue #6495ED

Definition at line 319 of file StandardColours.cs.

7.15.2.19 Cornsilk

```
Colour VectSharp.Colours.Cornsilk = Colour.FromRgb(255, 248, 220)  [static]
```

Cornsilk #FFF8DC

Definition at line 747 of file StandardColours.cs.

7.15.2.20 Crimson

```
Colour VectSharp.Colours.Crimson = Colour.FromRgb(220, 20, 60)  [static]
```

Crimson #DC143C

Definition at line 555 of file StandardColours.cs.

7.15.2.21 Cyan

```
Colour VectSharp.Colours.Cyan = Colour.FromRgb(0, 255, 255)  [static]
```

Cyan #00FFFF

Definition at line 247 of file StandardColours.cs.

7.15.2.22 DarkBlue

```
Colour VectSharp.Colours.DarkBlue = Colour.FromRgb(0, 0, 139)  [static]
```

DarkBlue #00008B

Definition at line 195 of file StandardColours.cs.

7.15.2.23 DarkCyan

```
Colour VectSharp.Colours.DarkCyan = Colour.FromRgb(0, 139, 139)  [static]
```

DarkCyan #008B8B

Definition at line 219 of file StandardColours.cs.

7.15.2.24 DarkGoldenRod

```
Colour VectSharp.Colours.DarkGoldenRod = Colour.FromRgb(184, 134, 11)  [static]
```

DarkGoldenRod #B8860B

Definition at line 491 of file StandardColours.cs.

7.15.2.25 DarkGray

```
Colour VectSharp.Colours.DarkGray = Colour.FromRgb(169, 169, 169)  [static]
```

DarkGray #A9A9A9

Definition at line 459 of file StandardColours.cs.

7.15.2.26 DarkGreen

```
Colour VectSharp.Colours.DarkGreen = Colour.FromRgb(0, 100, 0)  [static]
```

DarkGreen #006400

Definition at line 207 of file StandardColours.cs.

7.15.2.27 DarkGrey

```
Colour VectSharp.Colours.DarkGrey = Colour.FromRgb(169, 169, 169)  [static]
```

DarkGrey #A9A9A9

Definition at line 463 of file StandardColours.cs.

7.15.2.28 DarkKhaki

```
Colour VectSharp.Colours.DarkKhaki = Colour.FromRgb(189, 183, 107)  [static]
```

DarkKhaki #BDB76B

Definition at line 503 of file StandardColours.cs.

7.15.2.29 DarkMagenta

```
Colour VectSharp.Colours.DarkMagenta = Colour.FromRgb(139, 0, 139)  [static]
```

DarkMagenta #8B008B

Definition at line 415 of file StandardColours.cs.

7.15.2.30 DarkOliveGreen

```
Colour VectSharp.Colours.DarkOliveGreen = Colour.FromRgb(85, 107, 47)  [static]
```

DarkOliveGreen #556B2F

Definition at line 311 of file StandardColours.cs.

7.15.2.31 DarkOrange

```
Colour VectSharp.Colours.DarkOrange = Colour.FromRgb(255, 140, 0)  [static]
```

DarkOrange #FF8C00

Definition at line 687 of file StandardColours.cs.

7.15.2.32 DarkOrchid

```
Colour VectSharp.Colours.DarkOrchid = Colour.FromRgb(153, 50, 204)  [static]
```

DarkOrchid #9932CC

Definition at line 443 of file StandardColours.cs.

7.15.2.33 DarkRed

```
Colour VectSharp.Colours.DarkRed = Colour.FromRgb(139, 0, 0)  [static]
```

DarkRed #8B0000

Definition at line 411 of file StandardColours.cs.

7.15.2.34 DarkSalmon

```
Colour VectSharp.Colours.DarkSalmon = Colour.FromRgb(233, 150, 122)  [static]
```

DarkSalmon #E9967A

Definition at line 579 of file StandardColours.cs.

7.15.2.35 DarkSeaGreen

```
Colour VectSharp.Colours.DarkSeaGreen = Colour.FromRgb(143, 188, 143)  [static]
```

DarkSeaGreen #8FBC8F

Definition at line 423 of file StandardColours.cs.

7.15.2.36 DarkSlateBlue

```
Colour VectSharp.Colours.DarkSlateBlue = Colour.FromRgb(72, 61, 139)  [static]
```

DarkSlateBlue #483D8B

Definition at line 299 of file StandardColours.cs.

7.15.2.37 DarkSlateGray

```
Colour VectSharp.Colours.DarkSlateGray = Colour.FromRgb(47, 79, 79)  [static]
```

DarkSlateGray #2F4F4F

Definition at line 271 of file StandardColours.cs.

7.15.2.38 DarkSlateGrey

```
Colour VectSharp.Colours.DarkSlateGrey = Colour.FromRgb(47, 79, 79)  [static]
```

DarkSlateGrey #2F4F4F

Definition at line 275 of file StandardColours.cs.

7.15.2.39 DarkTurquoise

```
Colour VectSharp.Colours.DarkTurquoise = Colour.FromRgb(0, 206, 209)  [static]
```

DarkTurquoise #00CED1

Definition at line 227 of file StandardColours.cs.

7.15.2.40 DarkViolet

```
Colour VectSharp.Colours.DarkViolet = Colour.FromRgb(148, 0, 211)  [static]
```

DarkViolet #9400D3

Definition at line 435 of file StandardColours.cs.

7.15.2.41 DeepPink

```
Colour VectSharp.Colours.DeepPink = Colour.FromRgb(255, 20, 147)  [static]
```

DeepPink #FF1493

Definition at line 667 of file StandardColours.cs.

7.15.2.42 DeepSkyBlue

```
Colour VectSharp.Colours.DeepSkyBlue = Colour.FromRgb(0, 191, 255)  [static]
```

DeepSkyBlue #00BFFF

Definition at line 223 of file StandardColours.cs.

7.15.2.43 DimGray

```
Colour VectSharp.Colours.DimGray = Colour.FromRgb(105, 105, 105)  [static]
```

DimGray #696969

Definition at line 331 of file StandardColours.cs.

7.15.2.44 DimGrey

```
Colour VectSharp.Colours.DimGrey = Colour.FromRgb(105, 105, 105)  [static]
```

DimGrey #696969

Definition at line 335 of file StandardColours.cs.

7.15.2.45 DodgerBlue

```
Colour VectSharp.Colours.DodgerBlue = Colour.FromRgb(30, 144, 255)  [static]
```

DodgerBlue #1E90FF

Definition at line 255 of file StandardColours.cs.

7.15.2.46 FireBrick

```
Colour VectSharp.Colours.FireBrick = Colour.FromRgb(178, 34, 34)  [static]
```

FireBrick #B22222

Definition at line 487 of file StandardColours.cs.

7.15.2.47 FloralWhite

```
Colour VectSharp.Colours.FloralWhite = Colour.FromRgb(255, 250, 240)  [static]
```

FloralWhite #FFFAF0

Definition at line 755 of file StandardColours.cs.

7.15.2.48 ForestGreen

```
Colour VectSharp.Colours.ForestGreen = Colour.FromRgb(34, 139, 34)  [static]
```

ForestGreen #228B22

Definition at line 263 of file StandardColours.cs.

7.15.2.49 Fuchsia

```
Colour VectSharp.Colours.Fuchsia = Colour.FromRgb(255, 0, 255)  [static]
```

Fuchsia #FF00FF

Definition at line 659 of file StandardColours.cs.

7.15.2.50 Gainsboro

```
Colour VectSharp.Colours.Gainsboro = Colour.FromRgb(220, 220, 220)  [static]
```

Gainsboro #DCDCDC

Definition at line 559 of file StandardColours.cs.

7.15.2.51 GhostWhite

```
Colour VectSharp.Colours.GhostWhite = Colour.FromRgb(248, 248, 255)  [static]
```

GhostWhite #F8F8FF

Definition at line 631 of file StandardColours.cs.

7.15.2.52 Gold

```
Colour VectSharp.Colours.Gold = Colour.FromRgb(255, 215, 0)  [static]
```

Gold #FFD700

Definition at line 707 of file StandardColours.cs.

7.15.2.53 GoldenRod

```
Colour VectSharp.Colours.GoldenRod = Colour.FromRgb(218, 165, 32)  [static]
```

GoldenRod #DAA520

Definition at line 547 of file StandardColours.cs.

7.15.2.54 Gray

```
Colour VectSharp.Colours.Gray = Colour.FromRgb(128, 128, 128)  [static]
```

Gray #808080

Definition at line 391 of file StandardColours.cs.

7.15.2.55 Green

```
Colour VectSharp.Colours.Green = Colour.FromRgb(0, 128, 0)  [static]
```

Green #008000

Definition at line 211 of file StandardColours.cs.

7.15.2.56 GreenYellow

```
Colour VectSharp.Colours.GreenYellow = Colour.FromRgb(173, 255, 47)  [static]
```

GreenYellow #ADFF2F

Definition at line 471 of file StandardColours.cs.

7.15.2.57 Grey

```
Colour VectSharp.Colours.Grey = Colour.FromRgb(128, 128, 128)  [static]
```

Grey #808080

Definition at line 395 of file StandardColours.cs.

7.15.2.58 HoneyDew

```
Colour VectSharp.Colours.HoneyDew = Colour.FromRgb(240, 255, 240)  [static]
```

HoneyDew #F0FFF0

Definition at line 603 of file StandardColours.cs.

7.15.2.59 HotPink

```
Colour VectSharp.Colours.HotPink = Colour.FromRgb(255, 105, 180)  [static]
```

HotPink #FF69B4

Definition at line 679 of file StandardColours.cs.

7.15.2.60 IndianRed

```
Colour VectSharp.Colours.IndianRed = Colour.FromRgb(205, 92, 92)  [static]
```

IndianRed #CD5C5C

Definition at line 515 of file StandardColours.cs.

7.15.2.61 Indigo

```
Colour VectSharp.Colours.Indigo = Colour.FromRgb(75, 0, 130)  [static]
```

Indigo #4B0082

Definition at line 307 of file StandardColours.cs.

7.15.2.62 Ivory

```
Colour VectSharp.Colours.Ivory = Colour.FromRgb(255, 255, 240)  [static]
```

Ivory #FFFFF0

Definition at line 771 of file StandardColours.cs.

7.15.2.63 Khaki

```
Colour VectSharp.Colours.Khaki = Colour.FromRgb(240, 230, 140)  [static]
```

Khaki #F0E68C

Definition at line 595 of file StandardColours.cs.

7.15.2.64 Lavender

```
Colour VectSharp.Colours.Lavender = Colour.FromRgb(230, 230, 250)  [static]
```

Lavender #E6E6FA

Definition at line 575 of file StandardColours.cs.

7.15.2.65 LavenderBlush

```
Colour VectSharp.Colours.LavenderBlush = Colour.FromRgb(255, 240, 245)  [static]
```

LavenderBlush #FFF0F5

Definition at line 739 of file StandardColours.cs.

7.15.2.66 LawnGreen

```
Colour VectSharp.Colours.LawnGreen = Colour.FromRgb(124, 252, 0)  [static]
```

LawnGreen #7CFC00

Definition at line 367 of file StandardColours.cs.

7.15.2.67 LemonChiffon

```
Colour VectSharp.Colours.LemonChiffon = Colour.FromRgb(255, 250, 205)  [static]
```

LemonChiffon #FFFACD

Definition at line 751 of file StandardColours.cs.

7.15.2.68 LightBlue

```
Colour VectSharp.Colours.LightBlue = Colour.FromRgb(173, 216, 230)  [static]
```

LightBlue #ADD8E6

Definition at line 467 of file StandardColours.cs.

7.15.2.69 LightCoral

```
Colour VectSharp.Colours.LightCoral = Colour.FromRgb(240, 128, 128)  [static]
```

LightCoral #F08080

Definition at line 591 of file StandardColours.cs.

7.15.2.70 LightCyan

```
Colour VectSharp.Colours.LightCyan = Colour.FromRgb(224, 255, 255)  [static]
```

LightCyan #E0FFFF

Definition at line 571 of file StandardColours.cs.

7.15.2.71 LightGoldenRodYellow

```
Colour VectSharp.Colours.LightGoldenRodYellow = Colour.FromRgb(250, 250, 210)  [static]
```

LightGoldenRodYellow #FAFAD2

Definition at line 647 of file StandardColours.cs.

7.15.2.72 LightGray

```
Colour VectSharp.Colours.LightGray = Colour.FromRgb(211, 211, 211)  [static]
```

LightGray #D3D3D3

Definition at line 531 of file StandardColours.cs.

7.15.2.73 LightGreen

```
Colour VectSharp.Colours.LightGreen = Colour.FromRgb(144, 238, 144)  [static]
```

LightGreen #90EE90

Definition at line 427 of file StandardColours.cs.

7.15.2.74 LightGrey

```
Colour VectSharp.Colours.LightGrey = Colour.FromRgb(211, 211, 211)  [static]
```

LightGrey #D3D3D3

Definition at line 535 of file StandardColours.cs.

7.15.2.75 LightPink

```
Colour VectSharp.Colours.LightPink = Colour.FromRgb(255, 182, 193)  [static]
```

LightPink #FFB6C1

Definition at line 699 of file StandardColours.cs.

7.15.2.76 LightSalmon

```
Colour VectSharp.Colours.LightSalmon = Colour.FromRgb(255, 160, 122)  [static]
```

LightSalmon #FFA07A

Definition at line 691 of file StandardColours.cs.

7.15.2.77 LightSeaGreen

```
Colour VectSharp.Colours.LightSeaGreen = Colour.FromRgb(32, 178, 170)  [static]
```

LightSeaGreen #20B2AA

Definition at line 259 of file StandardColours.cs.

7.15.2.78 LightSkyBlue

```
Colour VectSharp.Colours.LightSkyBlue = Colour.FromRgb(135, 206, 250)  [static]
```

LightSkyBlue #87CEFA

Definition at line 403 of file StandardColours.cs.

7.15.2.79 LightSlateGray

```
Colour VectSharp.Colours.LightSlateGray = Colour.FromRgb(119, 136, 153)  [static]
```

LightSlateGray #778899

Definition at line 355 of file StandardColours.cs.

7.15.2.80 LightSlateGrey

```
Colour VectSharp.Colours.LightSlateGrey = Colour.FromRgb(119, 136, 153)  [static]
```

LightSlateGrey #778899

Definition at line 359 of file StandardColours.cs.

7.15.2.81 LightSteelBlue

```
Colour VectSharp.Colours.LightSteelBlue = Colour.FromRgb(176, 196, 222)  [static]
```

LightSteelBlue #B0C4DE

Definition at line 479 of file StandardColours.cs.

7.15.2.82 LightYellow

```
Colour VectSharp.Colours.LightYellow = Colour.FromRgb(255, 255, 224)  [static]
```

LightYellow #FFFFE0

Definition at line 767 of file StandardColours.cs.

7.15.2.83 Lime

```
Colour VectSharp.Colours.Lime = Colour.FromRgb(0, 255, 0)  [static]
```

Lime #00FF00

Definition at line 235 of file StandardColours.cs.

7.15.2.84 LimeGreen

```
Colour VectSharp.Colours.LimeGreen = Colour.FromRgb(50, 205, 50)  [static]
```

LimeGreen #32CD32

Definition at line 279 of file StandardColours.cs.

7.15.2.85 Linen

```
Colour VectSharp.Colours.Linen = Colour.FromRgb(250, 240, 230)  [static]
```

Linen #FAF0E6

Definition at line 643 of file StandardColours.cs.

7.15.2.86 Magenta

```
Colour VectSharp.Colours.Magenta = Colour.FromRgb(255, 0, 255)  [static]
```

Magenta #FF00FF

Definition at line 663 of file StandardColours.cs.

7.15.2.87 Maroon

```
Colour VectSharp.Colours.Maroon = Colour.FromRgb(128, 0, 0)  [static]
```

Maroon #800000

Definition at line 379 of file StandardColours.cs.

7.15.2.88 MediumAquaMarine

```
Colour VectSharp.Colours.MediumAquaMarine = Colour.FromRgb(102, 205, 170)  [static]
```

MediumAquaMarine #66CDAA

Definition at line 327 of file StandardColours.cs.

7.15.2.89 MediumBlue

```
Colour VectSharp.Colours.MediumBlue = Colour.FromRgb(0, 0, 205)  [static]
```

MediumBlue #0000CD

Definition at line 199 of file StandardColours.cs.

7.15.2.90 MediumOrchid

```
Colour VectSharp.Colours.MediumOrchid = Colour.FromRgb(186, 85, 211)  [static]
```

MediumOrchid #BA55D3

Definition at line 495 of file StandardColours.cs.

7.15.2.91 MediumPurple

```
Colour VectSharp.Colours.MediumPurple = Colour.FromRgb(147, 112, 219)  [static]
```

MediumPurple #9370DB

Definition at line 431 of file StandardColours.cs.

7.15.2.92 MediumSeaGreen

```
Colour VectSharp.Colours.MediumSeaGreen = Colour.FromRgb(60, 179, 113)  [static]
```

MediumSeaGreen #3CB371

Definition at line 283 of file StandardColours.cs.

7.15.2.93 MediumSlateBlue

```
Colour VectSharp.Colours.MediumSlateBlue = Colour.FromRgb(123, 104, 238)  [static]
```

MediumSlateBlue #7B68EE

Definition at line 363 of file StandardColours.cs.

7.15.2.94 MediumSpringGreen

```
Colour VectSharp.Colours.MediumSpringGreen = Colour.FromRgb(0, 250, 154)  [static]
```

MediumSpringGreen #00FA9A

Definition at line 231 of file StandardColours.cs.

7.15.2.95 MediumTurquoise

```
Colour VectSharp.Colours.MediumTurquoise = Colour.FromRgb(72, 209, 204)  [static]
```

MediumTurquoise #48D1CC

Definition at line 303 of file StandardColours.cs.

7.15.2.96 MediumVioletRed

```
Colour VectSharp.Colours.MediumVioletRed = Colour.FromRgb(199, 21, 133)  [static]
```

MediumVioletRed #C71585

Definition at line 511 of file StandardColours.cs.

7.15.2.97 MidnightBlue

```
Colour VectSharp.Colours.MidnightBlue = Colour.FromRgb(25, 25, 112)  [static]
```

MidnightBlue #191970

Definition at line 251 of file StandardColours.cs.

7.15.2.98 MintCream

```
Colour VectSharp.Colours.MintCream = Colour.FromRgb(245, 255, 250)  [static]
```

MintCream #F5FFFA

Definition at line 627 of file StandardColours.cs.

7.15.2.99 MistyRose

```
Colour VectSharp.Colours.MistyRose = Colour.FromRgb(255, 228, 225)  [static]
```

MistyRose #FFE4E1

Definition at line 727 of file StandardColours.cs.

7.15.2.100 Moccasin

```
Colour VectSharp.Colours.Moccasin = Colour.FromRgb(255, 228, 181)  [static]
```

Moccasin #FFE4B5

Definition at line 719 of file StandardColours.cs.

7.15.2.101 NavajoWhite

```
Colour VectSharp.Colours.NavajoWhite = Colour.FromRgb(255, 222, 173)  [static]
```

NavajoWhite #FFDEAD

Definition at line 715 of file StandardColours.cs.

7.15.2.102 Navy

```
Colour VectSharp.Colours.Navy = Colour.FromRgb(0, 0, 128)  [static]
```

Navy #000080

Definition at line 191 of file StandardColours.cs.

7.15.2.103 OldLace

```
Colour VectSharp.Colours.OldLace = Colour.FromRgb(253, 245, 230)  [static]
```

OldLace #FDF5E6

Definition at line 651 of file StandardColours.cs.

7.15.2.104 Olive

```
Colour VectSharp.Colours.Olive = Colour.FromRgb(128, 128, 0)  [static]
```

Olive #808000

Definition at line 387 of file StandardColours.cs.

7.15.2.105 OliveDrab

```
Colour VectSharp.Colours.OliveDrab = Colour.FromRgb(107, 142, 35)  [static]
```

OliveDrab #6B8E23

Definition at line 343 of file StandardColours.cs.

7.15.2.106 Orange

```
Colour VectSharp.Colours.Orange = Colour.FromRgb(255, 165, 0)  [static]
```

Orange #FFA500

Definition at line 695 of file StandardColours.cs.

7.15.2.107 OrangeRed

```
Colour VectSharp.Colours.OrangeRed = Colour.FromRgb(255, 69, 0)  [static]
```

OrangeRed #FF4500

Definition at line 671 of file StandardColours.cs.

7.15.2.108 Orchid

```
Colour VectSharp.Colours.Orchid = Colour.FromRgb(218, 112, 214)  [static]
```

Orchid #DA70D6

Definition at line 543 of file StandardColours.cs.

7.15.2.109 PaleGoldenRod

```
Colour VectSharp.Colours.PaleGoldenRod = Colour.FromRgb(238, 232, 170)  [static]
```

PaleGoldenRod #EEE8AA

Definition at line 587 of file StandardColours.cs.

7.15.2.110 PaleGreen

```
Colour VectSharp.Colours.PaleGreen = Colour.FromRgb(152, 251, 152)  [static]
```

PaleGreen #98FB98

Definition at line 439 of file StandardColours.cs.

7.15.2.111 PaleTurquoise

```
Colour VectSharp.Colours.PaleTurquoise = Colour.FromRgb(175, 238, 238)  [static]
```

PaleTurquoise #AFEEEE

Definition at line 475 of file StandardColours.cs.

7.15.2.112 PaleVioletRed

```
Colour VectSharp.Colours.PaleVioletRed = Colour.FromRgb(219, 112, 147)  [static]
```

PaleVioletRed #DB7093

Definition at line 551 of file StandardColours.cs.

7.15.2.113 PapayaWhip

```
Colour VectSharp.Colours.PapayaWhip = Colour.FromRgb(255, 239, 213)  [static]
```

PapayaWhip #FFEFD5

Definition at line 735 of file StandardColours.cs.

7.15.2.114 PeachPuff

```
Colour VectSharp.Colours.PeachPuff = Colour.FromRgb(255, 218, 185)  [static]
```

PeachPuff #FFDAB9

Definition at line 711 of file StandardColours.cs.

7.15.2.115 Peru

```
Colour VectSharp.Colours.Peru = Colour.FromRgb(205, 133, 63)  [static]
```

Peru #CD853F

Definition at line 519 of file StandardColours.cs.

7.15.2.116 Pink

```
Colour VectSharp.Colours.Pink = Colour.FromRgb(255, 192, 203)  [static]
```

Pink #FFC0CB

Definition at line 703 of file StandardColours.cs.

7.15.2.117 Plum

```
Colour VectSharp.Colours.Plum = Colour.FromRgb(221, 160, 221)  [static]
```

Plum #DDA0DD

Definition at line 563 of file StandardColours.cs.

7.15.2.118 PowderBlue

```
Colour VectSharp.Colours.PowderBlue = Colour.FromRgb(176, 224, 230)  [static]
```

PowderBlue #B0E0E6

Definition at line 483 of file StandardColours.cs.

7.15.2.119 Purple

```
Colour VectSharp.Colours.Purple = Colour.FromRgb(128, 0, 128)  [static]
```

Purple #800080

Definition at line 383 of file StandardColours.cs.

7.15.2.120 RebeccaPurple

```
Colour VectSharp.Colours.RebeccaPurple = Colour.FromRgb(102, 51, 153)  [static]
```

RebeccaPurple #663399

Definition at line 323 of file StandardColours.cs.

7.15.2.121 Red

```
Colour VectSharp.Colours.Red = Colour.FromRgb(255, 0, 0)  [static]
```

Red #FF0000

Definition at line 655 of file StandardColours.cs.

7.15.2.122 RosyBrown

```
Colour VectSharp.Colours.RosyBrown = Colour.FromRgb(188, 143, 143)  [static]
```

RosyBrown #BC8F8F

Definition at line 499 of file StandardColours.cs.

7.15.2.123 RoyalBlue

```
Colour VectSharp.Colours.RoyalBlue = Colour.FromRgb(65, 105, 225)  [static]
```

RoyalBlue #4169E1

Definition at line 291 of file StandardColours.cs.

7.15.2.124 SaddleBrown

```
Colour VectSharp.Colours.SaddleBrown = Colour.FromRgb(139, 69, 19)  [static]
```

SaddleBrown #8B4513

Definition at line 419 of file StandardColours.cs.

7.15.2.125 Salmon

```
Colour VectSharp.Colours.Salmon = Colour.FromRgb(250, 128, 114)  [static]
```

Salmon #FA8072

Definition at line 635 of file StandardColours.cs.

7.15.2.126 SandyBrown

```
Colour VectSharp.Colours.SandyBrown = Colour.FromRgb(244, 164, 96)  [static]
```

SandyBrown #F4A460

Definition at line 611 of file StandardColours.cs.

7.15.2.127 SeaGreen

```
Colour VectSharp.Colours.SeaGreen = Colour.FromRgb(46, 139, 87)  [static]
```

SeaGreen #2E8B57

Definition at line 267 of file StandardColours.cs.

7.15.2.128 SeaShell

```
Colour VectSharp.Colours.SeaShell = Colour.FromRgb(255, 245, 238)  [static]
```

SeaShell #FFF5EE

Definition at line 743 of file StandardColours.cs.

7.15.2.129 Sienna

```
Colour VectSharp.Colours.Sienna = Colour.FromRgb(160, 82, 45)  [static]
```

Sienna #A0522D

Definition at line 451 of file StandardColours.cs.

7.15.2.130 Silver

```
Colour VectSharp.Colours.Silver = Colour.FromRgb(192, 192, 192)  [static]
```

Silver #C0C0C0

Definition at line 507 of file StandardColours.cs.

7.15.2.131 SkyBlue

```
Colour VectSharp.Colours.SkyBlue = Colour.FromRgb(135, 206, 235)  [static]
```

SkyBlue #87CEEB

Definition at line 399 of file StandardColours.cs.

7.15.2.132 SlateBlue

```
Colour VectSharp.Colours.SlateBlue = Colour.FromRgb(106, 90, 205)  [static]
```

SlateBlue #6A5ACD

Definition at line 339 of file StandardColours.cs.

7.15.2.133 SlateGray

```
Colour VectSharp.Colours.SlateGray = Colour.FromRgb(112, 128, 144)  [static]
```

SlateGray #708090

Definition at line 347 of file StandardColours.cs.

7.15.2.134 SlateGrey

```
Colour VectSharp.Colours.SlateGrey = Colour.FromRgb(112, 128, 144)  [static]
```

SlateGrey #708090

Definition at line 351 of file StandardColours.cs.

7.15.2.135 Snow

```
Colour VectSharp.Colours.Snow = Colour.FromRgb(255, 250, 250)  [static]
```

Snow #FFFAFA

Definition at line 759 of file StandardColours.cs.

7.15.2.136 SpringGreen

```
Colour VectSharp.Colours.SpringGreen = Colour.FromRgb(0, 255, 127)  [static]
```

SpringGreen #00FF7F

Definition at line 239 of file StandardColours.cs.

7.15.2.137 SteelBlue

```
Colour VectSharp.Colours.SteelBlue = Colour.FromRgb(70, 130, 180)  [static]
```

SteelBlue #4682B4

Definition at line 295 of file StandardColours.cs.

7.15.2.138 Tan

```
Colour VectSharp.Colours.Tan = Colour.FromRgb(210, 180, 140)  [static]
```

Tan #D2B48C

Definition at line 527 of file StandardColours.cs.

7.15.2.139 Teal

```
Colour VectSharp.Colours.Teal = Colour.FromRgb(0, 128, 128)  [static]
```

Teal #008080

Definition at line 215 of file StandardColours.cs.

7.15.2.140 Thistle

```
Colour VectSharp.Colours.Thistle = Colour.FromRgb(216, 191, 216)  [static]
```

Thistle #D8BFD8

Definition at line 539 of file StandardColours.cs.

7.15.2.141 Tomato

```
Colour VectSharp.Colours.Tomato = Colour.FromRgb(255, 99, 71)  [static]
```

Tomato #FF6347

Definition at line 675 of file StandardColours.cs.

7.15.2.142 Turquoise

```
Colour VectSharp.Colours.Turquoise = Colour.FromRgb(64, 224, 208)  [static]
```

Turquoise #40E0D0

Definition at line 287 of file StandardColours.cs.

7.15.2.143 Violet

```
Colour VectSharp.Colours.Violet = Colour.FromRgb(238, 130, 238)  [static]
```

Violet #EE82EE

Definition at line 583 of file StandardColours.cs.

7.15.2.144 Wheat

```
Colour VectSharp.Colours.Wheat = Colour.FromRgb(245, 222, 179)  [static]
```

Wheat #F5DEB3

Definition at line 615 of file StandardColours.cs.

7.15.2.145 White

```
Colour VectSharp.Colours.White = Colour.FromRgb(255, 255, 255)  [static]
```

White #FFFFFF

Definition at line 775 of file StandardColours.cs.

7.15.2.146 WhiteSmoke

```
Colour VectSharp.Colours.WhiteSmoke = Colour.FromRgb(245, 245, 245)  [static]
```

WhiteSmoke #F5F5F5

Definition at line 623 of file StandardColours.cs.

7.15.2.147 Yellow

```
Colour VectSharp.Colours.Yellow = Colour.FromRgb(255, 255, 0)  [static]
```

Yellow #FFFF00

Definition at line 763 of file StandardColours.cs.

7.15.2.148 YellowGreen

```
Colour VectSharp.Colours.YellowGreen = Colour.FromRgb(154, 205, 50)  [static]
```

YellowGreen #9ACD32

Definition at line 447 of file StandardColours.cs.

The documentation for this class was generated from the following file:

- VectSharp/StandardColours.cs

7.16 VectSharp.Filters.CompositeLocationInvariantFilter Class Reference

Represents a filter that corresponds to applying multiple ILocationInvariantFilters one after the other.

Inheritance diagram for VectSharp.Filters.CompositeLocationInvariantFilter:

Public Member Functions

- CompositeLocationInvariantFilter (IEnumerable< ILocationInvariantFilter > filters)

  Creates a new CompositeLocationInvariantFilter with the specified filters.
- CompositeLocationInvariantFilter (params ILocationInvariantFilter[ ] filters)

  Creates a new CompositeLocationInvariantFilter with the specified filters.
- RasterImage Filter (RasterImage image, double scale)

  Applies the filter to a RasterImage.

Parameters

image	*The RasterImage to which the filter will be applied.*
scale	*The scale of the image with respect to the filter.*

Returns

A new RasterImage containing the filtered image. The source image is left unaltered.

Properties

- Point TopLeftMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.
- Point BottomRightMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

- ImmutableList< ILocationInvariantFilter > Filters [get]

  The filters that are applied by this filter.

7.16.1 Detailed Description

Represents a filter that corresponds to applying multiple ILocationInvariantFilters one after the other.

Definition at line 26 of file CompositeFilter.cs.

7.16.2 Constructor & Destructor Documentation

7.16.2.1 CompositeLocationInvariantFilter() [1/2]

```
VectSharp.Filters.CompositeLocationInvariantFilter.CompositeLocationInvariantFilter (
            IEnumerable< ILocationInvariantFilter > filters )
```

Creates a new CompositeLocationInvariantFilter with the specified filters.

Parameters

filters	The filters that will be applied by the new filter.

Definition at line 43 of file CompositeFilter.cs.

7.16.2.2 CompositeLocationInvariantFilter() [2/2]

```
VectSharp.Filters.CompositeLocationInvariantFilter.CompositeLocationInvariantFilter (
            params ILocationInvariantFilter[] filters )
```

Creates a new CompositeLocationInvariantFilter with the specified filters.

Parameters

filters	The filters that will be applied by the new filter.

Definition at line 70 of file CompositeFilter.cs.

7.16.3 Member Function Documentation

7.16.3.1 Filter()

```
RasterImage VectSharp.Filters.CompositeLocationInvariantFilter.Filter (
            RasterImage image,
            double scale )
```

Applies the filter to a RasterImage.

Parameters

image	The RasterImage to which the filter will be applied.
scale	The scale of the image with respect to the filter.

Returns

A new RasterImage containing the filtered image. The source *image* is left unaltered.

Implements VectSharp.Filters.ILocationInvariantFilter.

Definition at line 91 of file CompositeFilter.cs.

7.16.4 Property Documentation

7.16.4.1 BottomRightMargin

```
Point VectSharp.Filters.CompositeLocationInvariantFilter.BottomRightMargin  [get]
```

Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 32 of file CompositeFilter.cs.

7.16.4.2 Filters

```
ImmutableList<ILocationInvariantFilter> VectSharp.Filters.CompositeLocationInvariantFilter.↩
Filters  [get]
```

The filters that are applied by this filter.

Definition at line 37 of file CompositeFilter.cs.

7.16.4.3 TopLeftMargin

```
Point VectSharp.Filters.CompositeLocationInvariantFilter.TopLeftMargin [get]
```

Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 29 of file CompositeFilter.cs.

The documentation for this class was generated from the following file:

- VectSharp/Filters/CompositeFilter.cs

7.17 VectSharp.AnimatedPNG.CompressedFrame Class Reference

Represents an individual frame of a PNG animation.

Inheritance diagram for VectSharp.AnimatedPNG.CompressedFrame:

Public Member Functions

- unsafe CompressedFrame (DisposableIntPtr rawFrameData, int width, int height, bool hasAlpha, double duration)

  Creates a new CompressedFrame from raw pixel data in RGB or RGBA format.
- unsafe CompressedFrame (DisposableIntPtr rawFrameData, DisposableIntPtr otherFrameData, bool is↩FirstFrame, int width, int height, bool hasAlpha, double duration)

  Creates a new CompressedFrame from raw pixel data in RGB or RGBA format, applying inter-frame compression based on another frame.
- void Dispose ()

Properties

- double Duration [get]

  The duration of the frame in milliseconds.

7.17.1 Detailed Description

Represents an individual frame of a PNG animation.

Definition at line 56 of file ImageFormats.cs.

7.17.2 Constructor & Destructor Documentation

7.17.2.1 CompressedFrame() [1/2]

```
unsafe VectSharp.AnimatedPNG.CompressedFrame.CompressedFrame (
            DisposableIntPtr rawFrameData,
            int width,
            int height,
            bool hasAlpha,
            double duration )
```

Creates a new CompressedFrame from raw pixel data in RGB or RGBA format.

Parameters

rawFrameData	A pointer to the raw pixel data for the frame.
width	The width of the image in pixels.
height	The height of the image in pixels.
hasAlpha	If the image data is in RGBA format, set this to `true`; if it is in RGB format, set it to `false`.
duration	The duration of the frame in milliseconds.

Definition at line 76 of file ImageFormats.cs.

7.17.2.2 CompressedFrame() [2/2]

```
unsafe VectSharp.AnimatedPNG.CompressedFrame.CompressedFrame (
            DisposableIntPtr rawFrameData,
            DisposableIntPtr otherFrameData,
            bool isFirstFrame,
            int width,
            int height,
            bool hasAlpha,
            double duration )
```

Creates a new CompressedFrame from raw pixel data in RGB or RGBA format, applying inter-frame compression based on another frame.

Parameters

rawFrameData	A pointer to the raw pixel data for the new frame.

Parameters

otherFrameData	A pointer to the raw pixel data for the frame to use as a reference. This can either be the first frame in the animation, or the frame immediately preceeding the current frame.
isFirstFrame	If *otherFrameData* is the raw pixel data for the first frame in the animation, set this to `true`; if it is the immediately preceeding frame, set this to `false`.
width	The width of the image in pixels.
height	The height of the image in pixels.
hasAlpha	If the image data is in RGBA format, set this to `true`; if it is in RGB format, set it to `false`.
duration	The duration of the frame in milliseconds.

Definition at line 93 of file ImageFormats.cs.

7.17.3 Member Function Documentation

7.17.3.1 Dispose()

```
void VectSharp.AnimatedPNG.CompressedFrame.Dispose ( )
```

Definition at line 165 of file ImageFormats.cs.

7.17.4 Property Documentation

7.17.4.1 Duration

```
double VectSharp.AnimatedPNG.CompressedFrame.Duration  [get]
```

The duration of the frame in milliseconds.

Definition at line 64 of file ImageFormats.cs.

The documentation for this class was generated from the following file:

- VectSharp/ImageFormats.cs

7.18 VectSharp.Filters.ConvolutionFilter Class Reference

Represents a filter that applies a matrix convolution to the image.

Inheritance diagram for VectSharp.Filters.ConvolutionFilter:

Public Member Functions

- ConvolutionFilter (double[,] kernel, double scale, bool preserveAlpha=true, double normalisation=1, double bias=0)

  Creates a new ConvolutionFilter with the specified parameters.
- virtual RasterImage Filter (RasterImage image, double scale)

  Applies the filter to a RasterImage.

Parameters

image	*The RasterImage to which the filter will be applied.*
scale	*The scale of the image with respect to the filter.*

Returns

A new RasterImage containing the filtered image. The source image is left unaltered.

Properties

- Point TopLeftMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.
- Point BottomRightMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.
- virtual double[,] Kernel `[get]`

  The kernel of the ConvolutionFilter. The dimensions of this matrix should all be odd numbers. The larger the kernel, the worse the performance.

- virtual double Normalisation = 1 `[get]`

  The normalisation value that is applies to the kernel.
- virtual double Bias = 0 `[get]`

  The bias value that is added to every colour component when the filter is applied.
- virtual double Scale `[get]`

  The scale relating the size of the kernel to graphics units.
- virtual bool PreserveAlpha = true `[get]`

  If this is `true`, the alpha value of the input pixels is preserved. Otherwise, the alpha channel is subject to the same convolution process as the other colour components.

7.18.1 Detailed Description

Represents a filter that applies a matrix convolution to the image.

Definition at line 26 of file ConvolutionFilter.cs.

7.18.2 Constructor & Destructor Documentation

7.18.2.1 ConvolutionFilter()

```
VectSharp.Filters.ConvolutionFilter.ConvolutionFilter (
            double kernel[,],
            double scale,
            bool preserveAlpha = true,
            double normalisation = 1,
            double bias = 0 )
```

Creates a new ConvolutionFilter with the specified parameters.

Parameters

kernel	The kernel of the ConvolutionFilter. The dimensions of this matrix should all be odd numbers. The larger the kernel, the worse the performance.
scale	The scale relating the size of the kernel to graphics units.
preserveAlpha	If this is `true`, the alpha value of the input pixels is preserved. Otherwise, the alpha channel is subject to the same convolution process as the other colour components.
normalisation	The normalisation value that is applies to the kernel.
bias	The bias value that is added to every colour component when the filter is applied.

Exceptions

ArgumentException	This exception is thrown when the kernel dimensions are not odd numbers.

Definition at line 67 of file ConvolutionFilter.cs.

7.18.3 Member Function Documentation

7.18.3.1 Filter()

```
virtual RasterImage VectSharp.Filters.ConvolutionFilter.Filter (
            RasterImage image,
            double scale )  [virtual]
```

Applies the filter to a RasterImage.

Parameters

image	The RasterImage to which the filter will be applied.
scale	The scale of the image with respect to the filter.

Returns

A new RasterImage containing the filtered image. The source *image* is left unaltered.

Implements VectSharp.Filters.ILocationInvariantFilter.

Definition at line 88 of file ConvolutionFilter.cs.

7.18.4 Property Documentation

7.18.4.1 Bias

```
virtual double VectSharp.Filters.ConvolutionFilter.Bias = 0  [get]
```

The bias value that is added to every colour component when the filter is applied.

Definition at line 46 of file ConvolutionFilter.cs.

7.18.4.2 BottomRightMargin

```
Point VectSharp.Filters.ConvolutionFilter.BottomRightMargin  [get]
```

Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 31 of file ConvolutionFilter.cs.

7.18.4.3 Kernel

```
virtual double [,] VectSharp.Filters.ConvolutionFilter.Kernel  [get]
```

The kernel of the ConvolutionFilter. The dimensions of this matrix should all be odd numbers. The larger the kernel, the worse the performance.

Definition at line 36 of file ConvolutionFilter.cs.

7.18.4.4 Normalisation

```
virtual double VectSharp.Filters.ConvolutionFilter.Normalisation = 1  [get]
```

The normalisation value that is applies to the kernel.

Definition at line 41 of file ConvolutionFilter.cs.

7.18.4.5 PreserveAlpha

```
virtual bool VectSharp.Filters.ConvolutionFilter.PreserveAlpha = true  [get]
```

If this is `true`, the alpha value of the input pixels is preserved. Otherwise, the alpha channel is subject to the same convolution process as the other colour components.

Definition at line 56 of file ConvolutionFilter.cs.

7.18.4.6 Scale

```
virtual double VectSharp.Filters.ConvolutionFilter.Scale  [get]
```

The scale relating the size of the kernel to graphics units.

Definition at line 51 of file ConvolutionFilter.cs.

7.18.4.7 TopLeftMargin

```
Point VectSharp.Filters.ConvolutionFilter.TopLeftMargin  [get]
```

Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 29 of file ConvolutionFilter.cs.

The documentation for this class was generated from the following file:

- VectSharp/Filters/ConvolutionFilter.cs

7.19 VectSharp.DefaultFontLibrary Class Reference

A default font library that resolves standard families using the embedded fonts.

Inheritance diagram for VectSharp.DefaultFontLibrary:

Public Member Functions

- override FontFamily ResolveFontFamily (string fontFamily)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised.

Parameters

fontFamily	*The name of the font family to create, or the path to a TTF file.*

Returns

If the font family name or the true type file is valid, a FontFamily object corresponding to the specified font family.

- override FontFamily ResolveFontFamily (FontFamily.StandardFontFamilies standardFontFamily)

  Create a new font family from the specified standard font family name.

Parameters

standardFontFamily	*The standard name of the font family.*

Returns

A FontFamily object corresponding to the specified font family.

7.19.1 Detailed Description

A default font library that resolves standard families using the embedded fonts.

Definition at line 462 of file FontLibrary.cs.

7.19.2 Member Function Documentation

7.19.2.1 ResolveFontFamily() [1/2]

```
override FontFamily VectSharp.DefaultFontLibrary.ResolveFontFamily (
            FontFamily.StandardFontFamilies standardFontFamily )  [virtual]
```

Create a new font family from the specified standard font family name.

Parameters

standardFontFamily	The standard name of the font family.

Returns

A FontFamily object corresponding to the specified font family.

Implements VectSharp.FontLibrary.

Definition at line 524 of file FontLibrary.cs.

7.19.2.2 ResolveFontFamily() [2/2]

```
override FontFamily VectSharp.DefaultFontLibrary.ResolveFontFamily (
            string fontFamily )  [virtual]
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised.

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.

Returns

If the font family name or the true type file is valid, a FontFamily object corresponding to the specified font family.

Implements VectSharp.FontLibrary.

Definition at line 465 of file FontLibrary.cs.

The documentation for this class was generated from the following file:

- VectSharp/FontLibrary.cs

7.20 VectSharp.Font.DetailedFontMetrics Class Reference

Represents detailed information about the metrics of a text string when drawn with a certain font.

Properties

- double Width [get]

  Width of the text (measured on the actual glyph outlines).
- double Height [get]

  Height of the text (measured on the actual glyph outlines).
- double LeftSideBearing [get]

  How much the leftmost glyph in the string overhangs the glyph origin on the left. Positive for glyphs that hang past the origin (e.g. italic 'f').
- double RightSideBearing [get]

  How much the rightmost glyph in the string overhangs the glyph end on the right. Positive for glyphs that hang past the end (e.g. italic 'f').
- double Top [get]

  Height of the tallest glyph in the string over the baseline. Always >= 0.
- double Bottom [get]

  Depth of the deepest glyph in the string below the baseline. Always <= 0.
- double AdvanceWidth [get]

  Advance width of the text (excluding any left- or right- side bearing).

7.20.1 Detailed Description

Represents detailed information about the metrics of a text string when drawn with a certain font.

Definition at line 101 of file Font.cs.

7.20.2 Property Documentation

7.20.2.1 AdvanceWidth

```
double VectSharp.Font.DetailedFontMetrics.AdvanceWidth [get]
```

Advance width of the text (excluding any left- or right- side bearing).

Definition at line 136 of file Font.cs.

7.20.2.2 Bottom

```
double VectSharp.Font.DetailedFontMetrics.Bottom [get]
```

Depth of the deepest glyph in the string below the baseline. Always <= 0.

Definition at line 131 of file Font.cs.

7.20.2.3 Height

```
double VectSharp.Font.DetailedFontMetrics.Height [get]
```

Height of the text (measured on the actual glyph outlines).

Definition at line 111 of file Font.cs.

7.20.2.4 LeftSideBearing

```
double VectSharp.Font.DetailedFontMetrics.LeftSideBearing [get]
```

How much the leftmost glyph in the string overhangs the glyph origin on the left. Positive for glyphs that hang past the origin (e.g. italic 'f').

Definition at line 116 of file Font.cs.

7.20.2.5 RightSideBearing

```
double VectSharp.Font.DetailedFontMetrics.RightSideBearing [get]
```

How much the rightmost glyph in the string overhangs the glyph end on the right. Positive for glyphs that hang past the end (e.g. italic 'f').

Definition at line 121 of file Font.cs.

7.20.2.6 Top

```
double VectSharp.Font.DetailedFontMetrics.Top [get]
```

Height of the tallest glyph in the string over the baseline. Always >= 0.

Definition at line 126 of file Font.cs.

7.20.2.7 Width

```
double VectSharp.Font.DetailedFontMetrics.Width  [get]
```

Width of the text (measured on the actual glyph outlines).

Definition at line 106 of file Font.cs.

The documentation for this class was generated from the following file:

- VectSharp/Font.cs

7.21 VectSharp.DisposableIntPtr Class Reference

An IDisposable wrapper around an IntPtr that frees the allocated memory when it is disposed.

Inheritance diagram for VectSharp.DisposableIntPtr:

Public Member Functions

- DisposableIntPtr (IntPtr pointer)

  Create a new DisposableIntPtr.
- void Dispose ()

Public Attributes

- readonly IntPtr InternalPointer

  The pointer to the unmanaged memory.

7.21.1 Detailed Description

An IDisposable wrapper around an IntPtr that frees the allocated memory when it is disposed.

Definition at line 53 of file RasterImage.cs.

7.21.2 Constructor & Destructor Documentation

7.21.2.1 DisposableIntPtr()

```
VectSharp.DisposableIntPtr.DisposableIntPtr (
            IntPtr pointer )
```

Create a new DisposableIntPtr.

Parameters

pointer	The pointer that should be freed upon disposing of this object.

Definition at line 64 of file RasterImage.cs.

7.21.3 Member Function Documentation

7.21.3.1 Dispose()

```
void VectSharp.DisposableIntPtr.Dispose ( )
```

Definition at line 88 of file RasterImage.cs.

7.21.4 Member Data Documentation

7.21.4.1 InternalPointer

```
readonly IntPtr VectSharp.DisposableIntPtr.InternalPointer
```

The pointer to the unmanaged memory.

Definition at line 58 of file RasterImage.cs.

The documentation for this class was generated from the following file:

- VectSharp/RasterImage.cs

7.22 VectSharp.Document Class Reference

Represents a collection of pages.

Public Member Functions

- Document ()

  Create a new document.

Public Attributes

- List< Page > Pages = new List<Page>()

  The pages in the document.

7.22.1 Detailed Description

Represents a collection of pages.

Definition at line 27 of file Document.cs.

7.22.2 Constructor & Destructor Documentation

7.22.2.1 Document()

```
VectSharp.Document.Document ( )
```

Create a new document.

Definition at line 38 of file Document.cs.

7.22.3 Member Data Documentation

7.22.3.1 Pages

```
List<Page> VectSharp.Document.Pages = new List<Page>()
```

The pages in the document.

Definition at line 32 of file Document.cs.

The documentation for this class was generated from the following file:

- VectSharp/Document.cs

7.23 VectSharp.Canvas.FilterOption Class Reference

Determines how and whether image filters are rasterised.

Public Types

- enum FilterOperations

  Defines whether image filters should be rasterised or not.

Public Member Functions

- FilterOption (FilterOperations operation, double rasterisationResolution, bool rasterisationResolutionRelative)

  Create a new FilterOption object.

Static Public Attributes

- static FilterOption Default = new FilterOption(FilterOperations.RasteriseAllWithSkia, 1, true)

  The default options for image filter rasterisation.

Properties

- FilterOperations Operation = FilterOperations.RasteriseAllWithSkia `[get]`

  Defines whether image filters should be rasterised or not.
- double RasterisationResolution = 1 `[get]`

  The resolution that will be used to rasterise image filters. Depending on the value of RasterisationResolutionRelative, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.
- bool RasterisationResolutionRelative = true `[get]`

  Determines whether the value of RasterisationResolution is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).

7.23.1 Detailed Description

Determines how and whether image filters are rasterised.

Definition at line 1469 of file SKRenderContext.cs.

7.23.2 Member Enumeration Documentation

7.23.2.1 FilterOperations

`enum VectSharp.Canvas.FilterOption.FilterOperations`

Defines whether image filters should be rasterised or not.

Definition at line 1474 of file SKRenderContext.cs.

7.23.3 Constructor & Destructor Documentation

7.23.3.1 FilterOption()

```
VectSharp.Canvas.FilterOption.FilterOption (
            FilterOperations operation,
            double rasterisationResolution,
            bool rasterisationResolutionRelative )
```

Create a new FilterOption object.

Parameters

operation	Defines whether image filters should be rasterised or not.
rasterisationResolution	The resolution that will be used to rasterise image filters. Depending on the value of RasterisationResolutionRelative, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.
rasterisationResolutionRelative	Determines whether the value of RasterisationResolution is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).

Definition at line 1523 of file SKRenderContext.cs.

7.23.4 Member Data Documentation

7.23.4.1 Default

```
FilterOption VectSharp.Canvas.FilterOption.Default = new FilterOption(FilterOperations.↩
RasteriseAllWithSkia, 1, true)  [static]
```

The default options for image filter rasterisation.

Definition at line 1515 of file SKRenderContext.cs.

7.23.5 Property Documentation

7.23.5.1 Operation

```
FilterOperations VectSharp.Canvas.FilterOption.Operation = FilterOperations.RasteriseAllWith↩
Skia  [get]
```

Defines whether image filters should be rasterised or not.

Definition at line 1500 of file SKRenderContext.cs.

7.23.5.2 RasterisationResolution

```
double VectSharp.Canvas.FilterOption.RasterisationResolution = 1  [get]
```

The resolution that will be used to rasterise image filters. Depending on the value of RasterisationResolutionRelative, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.

Definition at line 1505 of file SKRenderContext.cs.

7.23.5.3 RasterisationResolutionRelative

```
bool VectSharp.Canvas.FilterOption.RasterisationResolutionRelative = true  [get]
```

Determines whether the value of RasterisationResolution is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).

Definition at line 1510 of file SKRenderContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.Canvas/SKRenderContext.cs

7.24 VectSharp.PDF.PDFContextInterpreter.FilterOption Class Reference

Determines how and whether image filters are rasterised.

Public Types

- enum FilterOperations

  Defines whether image filters should be rasterised or not.

Public Member Functions

- FilterOption (FilterOperations operation, double rasterisationResolution, bool rasterisationResolutionRelative)

  Create a new FilterOption object.

Static Public Attributes

- static FilterOption Default = new FilterOption(FilterOperations.RasteriseAll, 1, true)

  The default options for image filter rasterisation.

Properties

- FilterOperations Operation = FilterOperations.RasteriseAll `[get]`

  Defines whether image filters should be rasterised or not.
- double RasterisationResolution = 1 `[get]`

  The resolution that will be used to rasterise image filters. Depending on the value of RasterisationResolutionRelative, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.
- bool RasterisationResolutionRelative = true `[get]`

  Determines whether the value of RasterisationResolution is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).

7.24.1 Detailed Description

Determines how and whether image filters are rasterised.

Definition at line 1140 of file PDFContext.cs.

7.24.2 Member Enumeration Documentation

7.24.2.1 FilterOperations

```
enum VectSharp.PDF.PDFContextInterpreter.FilterOption.FilterOperations
```

Defines whether image filters should be rasterised or not.

Definition at line 1145 of file PDFContext.cs.

7.24.3 Constructor & Destructor Documentation

7.24.3.1 FilterOption()

```
VectSharp.PDF.PDFContextInterpreter.FilterOption.FilterOption (
            FilterOperations operation,
            double rasterisationResolution,
            bool rasterisationResolutionRelative )
```

Create a new FilterOption object.

Parameters

operation	Defines whether image filters should be rasterised or not.
rasterisationResolution	The resolution that will be used to rasterise image filters. Depending on the value of RasterisationResolutionRelative, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.
rasterisationResolutionRelative	Determines whether the value of RasterisationResolution is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).

Definition at line 1189 of file PDFContext.cs.

7.24.4 Member Data Documentation

7.24.4.1 Default

```
FilterOption VectSharp.PDF.PDFContextInterpreter.FilterOption.Default = new FilterOption(Filter↩
Operations.RasteriseAll, 1, true)  [static]
```

The default options for image filter rasterisation.

Definition at line 1181 of file PDFContext.cs.

7.24.5 Property Documentation

7.24.5.1 Operation

```
FilterOperations VectSharp.PDF.PDFContextInterpreter.FilterOption.Operation = FilterOperations.↩
RasteriseAll  [get]
```

Defines whether image filters should be rasterised or not.

Definition at line 1166 of file PDFContext.cs.

7.24.5.2 RasterisationResolution

```
double VectSharp.PDF.PDFContextInterpreter.FilterOption.RasterisationResolution = 1  [get]
```

The resolution that will be used to rasterise image filters. Depending on the value of RasterisationResolutionRelative, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.

Definition at line 1171 of file PDFContext.cs.

7.24.5.3 RasterisationResolutionRelative

```
bool VectSharp.PDF.PDFContextInterpreter.FilterOption.RasterisationResolutionRelative = true
[get]
```

Determines whether the value of RasterisationResolution is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).

Definition at line 1176 of file PDFContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.PDF/PDFContext.cs

7.25 VectSharp.SVG.SVGContextInterpreter.FilterOption Class Reference

Determines how and whether image filters are rasterised.

Public Types

- enum FilterOperations

  Defines whether image filters should be rasterised or not.

Public Member Functions

- FilterOption (FilterOperations operation, double rasterisationResolution, bool rasterisationResolutionRelative)

  Create a new FilterOption object.

Static Public Attributes

- static FilterOption Default = new FilterOption(FilterOperations.RasteriseIfNecessary, 1, true)

  The default options for image filter rasterisation.

Properties

- FilterOperations Operation = FilterOperations.RasteriseIfNecessary `[get]`

  Defines whether image filters should be rasterised or not.
- double RasterisationResolution = 1 `[get]`

  The resolution that will be used to rasterise image filters. Depending on the value of RasterisationResolutionRelative, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.
- bool RasterisationResolutionRelative = true `[get]`

  Determines whether the value of RasterisationResolution is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).

7.25.1 Detailed Description

Determines how and whether image filters are rasterised.

Definition at line 2142 of file SVGContext.cs.

7.25.2 Member Enumeration Documentation

7.25.2.1 FilterOperations

```
enum VectSharp.SVG.SVGContextInterpreter.FilterOption.FilterOperations
```

Defines whether image filters should be rasterised or not.

Definition at line 2147 of file SVGContext.cs.

7.25.3 Constructor & Destructor Documentation

7.25.3.1 FilterOption()

```
VectSharp.SVG.SVGContextInterpreter.FilterOption.FilterOption (
            FilterOperations operation,
            double rasterisationResolution,
            bool rasterisationResolutionRelative )
```

Create a new FilterOption object.

Parameters

operation	Defines whether image filters should be rasterised or not.
rasterisationResolution	The resolution that will be used to rasterise image filters. Depending on the value of RasterisationResolutionRelative, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.
rasterisationResolutionRelative	Determines whether the value of RasterisationResolution is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).

Definition at line 2206 of file SVGContext.cs.

7.25.4 Member Data Documentation

7.25.4.1 Default

```
FilterOption VectSharp.SVG.SVGContextInterpreter.FilterOption.Default = new FilterOption(Filter↩
Operations.RasteriseIfNecessary, 1, true)  [static]
```

The default options for image filter rasterisation.

Definition at line 2198 of file SVGContext.cs.

7.25.5 Property Documentation

7.25.5.1 Operation

```
FilterOperations VectSharp.SVG.SVGContextInterpreter.FilterOption.Operation = FilterOperations.↩
RasteriseIfNecessary  [get]
```

Defines whether image filters should be rasterised or not.

Definition at line 2183 of file SVGContext.cs.

7.25.5.2 RasterisationResolution

```
double VectSharp.SVG.SVGContextInterpreter.FilterOption.RasterisationResolution = 1  [get]
```

The resolution that will be used to rasterise image filters. Depending on the value of RasterisationResolutionRelative, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.

Definition at line 2188 of file SVGContext.cs.

7.25.5.3 RasterisationResolutionRelative

```
bool VectSharp.SVG.SVGContextInterpreter.FilterOption.RasterisationResolutionRelative = true
[get]
```

Determines whether the value of RasterisationResolution is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).

Definition at line 2193 of file SVGContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.SVG/SVGContext.cs

7.26 VectSharp.Filters.FilterWithRasterisableParameter Class Reference

Represents a filter with a parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter.

Inheritance diagram for VectSharp.Filters.FilterWithRasterisableParameter:

Public Member Functions

- virtual void RasteriseParameter (Func< Graphics, Rectangle, double, bool, RasterImage > rasterisation↩ Method, double scale)

  Rasterises the filter's parameter at the specified scale, using the specified rasterisation method.

Parameters

rasterisationMethod	*The method used to rasterise the image. The first argument of this method is the Graphics to be rasterised, the second is a Rectangle representing the region to rasterise, the third is a double representing the scale, and the third is a boolean value indicating whether the resulting RasterImage should be interpolated.*
scale	*The scale factor at which the parameter is rasterised.*

- void Dispose ()

7.26.1 Detailed Description

Represents a filter with a parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter.

Definition at line 86 of file Filters.cs.

7.26.2 Member Function Documentation

7.26.2.1 Dispose()

```
void VectSharp.Filters.FilterWithRasterisableParameter.Dispose ( )
```

Definition at line 187 of file Filters.cs.

7.26.2.2 RasteriseParameter()

```
virtual void VectSharp.Filters.FilterWithRasterisableParameter.RasteriseParameter (
            Func< Graphics, Rectangle, double, bool, RasterImage > rasterisationMethod,
            double scale )  [virtual]
```

Rasterises the filter's parameter at the specified scale, using the specified rasterisation method.

Parameters

rasterisationMethod	The method used to rasterise the image. The first argument of this method is the Graphics to be rasterised, the second is a Rectangle representing the region to rasterise, the third is a double representing the scale, and the third is a boolean value indicating whether the resulting RasterImage should be interpolated.
scale	The scale factor at which the parameter is rasterised.

Implements VectSharp.Filters.IFilterWithRasterisableParameter.

Definition at line 135 of file Filters.cs.

The documentation for this class was generated from the following file:

- VectSharp/Filters/Filters.cs

7.27 VectSharp.Font Class Reference

Represents a typeface with a specific size.

Classes

- class DetailedFontMetrics

  Represents detailed information about the metrics of a text string when drawn with a certain font.
- class FontUnderline

  Represents options to underline text.

Public Member Functions

- Font (FontFamily fontFamily, double fontSize)

  Create a new Font object, given the base typeface and the font size.
- Font (FontFamily fontFamily, double fontSize, bool underlined)

  Create a new Font object, given the base typeface, the font size, and a boolean value determining whether text using this font should be underlined.
- Font (FontFamily fontFamily, double fontSize, FontUnderline underline)

  Create a new Font object, given the base typeface, the font size, and an object describing the underline properties of text drawn using this font.
- Size MeasureText (string text)

  Measure the size of a text string when typeset with this font.
- DetailedFontMetrics MeasureTextAdvanced (string text)

  Measure all the metrics of a text string when typeset with this font.

Static Public Attributes

- static bool EnableKerning = true

  Determines whether text kerning is enabled. Note that, even when this is set to `false`, text kerning will be applied on some platforms. For the best consistency, leave this set to `true`.

Properties

- double FontSize `[get]`

  Font size, in graphics units.
- FontFamily FontFamily `[get]`

  Font typeface.
- double Ascent `[get]`

  Maximum height over the baseline of the usual glyphs in the font (there may be glyphs taller than this). Always >= 0.
- double WinAscent `[get]`

  Height above the baseline for a clipping region (Windows ascent). Always >= 0.
- double Descent `[get]`

  Maximum depth below the baseline of the usual glyphs in the font (there may be glyphs deeper than this). Always <= 0.
- double YMax `[get]`

  Absolute maximum height over the baseline of the glyphs in the font. Always >= 0.
- double YMin `[get]`

  Absolute maximum depth below the baseline of the glyphs in the font. Always <= 0.
- FontUnderline Underline `[get]`

  Determines the underline style of text drawn using this font. If this is `null`, the text is not underlined.

7.27.1 Detailed Description

Represents a typeface with a specific size.

Definition at line 28 of file Font.cs.

7.27.2 Constructor & Destructor Documentation

7.27.2.1 Font() [1/3]

```
VectSharp.Font.Font (
            FontFamily fontFamily,
            double fontSize )
```

Create a new Font object, given the base typeface and the font size.

Parameters

fontFamily	Base typeface. See FontFamily.
fontSize	The font size, in graphics units.

Definition at line 165 of file Font.cs.

7.27.2.2 Font() [2/3]

```
VectSharp.Font.Font (
            FontFamily fontFamily,
            double fontSize,
            bool underlined )
```

Create a new Font object, given the base typeface, the font size, and a boolean value determining whether text using this font should be underlined.

Parameters

fontFamily	Base typeface. See FontFamily.
fontSize	The font size, in graphics units.
underlined	A boolean value determining whether text drawn using this font should be underlined. The appearance of the underline can be tweaked by changing the properties of the Underline property after the font has been created.

Definition at line 177 of file Font.cs.

7.27.2.3 Font() [3/3]

```
VectSharp.Font.Font (
            FontFamily fontFamily,
            double fontSize,
            FontUnderline underline )
```

Create a new Font object, given the base typeface, the font size, and an object describing the underline properties of text drawn using this font.

Parameters

fontFamily	Base typeface. See FontFamily.
fontSize	The font size, in graphics units.
underline	A FontUnderline object describing the underline properties of text drawn using this font.

Definition at line 194 of file Font.cs.

7.27.3 Member Function Documentation

7.27.3.1 MeasureText()

```
Size VectSharp.Font.MeasureText (
            string text )
```

Measure the size of a text string when typeset with this font.

Parameters

text	The string to measure.

Returns

A Size object representing the width and height of the text.

Definition at line 301 of file Font.cs.

7.27.3.2 MeasureTextAdvanced()

```
DetailedFontMetrics VectSharp.Font.MeasureTextAdvanced (
            string text )
```

Measure all the metrics of a text string when typeset with this font.

Parameters

text	The string to measure.

Returns

A DetailedFontMetrics object representing the metrics of the text.

Definition at line 358 of file Font.cs.

7.27.4 Member Data Documentation

7.27.4.1 EnableKerning

```
bool VectSharp.Font.EnableKerning = true  [static]
```

Determines whether text kerning is enabled. Note that, even when this is set to `false`, text kerning will be applied on some platforms. For the best consistency, leave this set to `true`.

Definition at line 33 of file Font.cs.

7.27.5 Property Documentation

7.27.5.1 Ascent

```
double VectSharp.Font.Ascent  [get]
```

Maximum height over the baseline of the usual glyphs in the font (there may be glyphs taller than this). Always >= 0.

Definition at line 204 of file Font.cs.

7.27.5.2 Descent

```
double VectSharp.Font.Descent  [get]
```

Maximum depth below the baseline of the usual glyphs in the font (there may be glyphs deeper than this). Always <= 0.

Definition at line 240 of file Font.cs.

7.27.5.3 FontFamily

```
FontFamily VectSharp.Font.FontFamily  [get]
```

Font typeface.

Definition at line 158 of file Font.cs.

7.27.5.4 FontSize

```
double VectSharp.Font.FontSize [get]
```

Font size, in graphics units.

Definition at line 153 of file Font.cs.

7.27.5.5 Underline

```
FontUnderline VectSharp.Font.Underline [get]
```

Determines the underline style of text drawn using this font. If this is `null`, the text is not underlined.

Definition at line 294 of file Font.cs.

7.27.5.6 WinAscent

```
double VectSharp.Font.WinAscent [get]
```

Height above the baseline for a clipping region (Windows ascent). Always >= 0.

Definition at line 222 of file Font.cs.

7.27.5.7 YMax

```
double VectSharp.Font.YMax [get]
```

Absolute maximum height over the baseline of the glyphs in the font. Always >= 0.

Definition at line 258 of file Font.cs.

7.27.5.8 YMin

```
double VectSharp.Font.YMin [get]
```

Absolute maximum depth below the baseline of the glyphs in the font. Always <= 0.

Definition at line 276 of file Font.cs.

The documentation for this class was generated from the following file:

- VectSharp/Font.cs

7.28 VectSharp.FontFamily Class Reference

Represents a typeface.

Inheritance diagram for VectSharp.FontFamily:

Public Types

- enum StandardFontFamilies

  The 14 standard font families.

Public Member Functions

- FontFamily (string fileName)

  Create a new FontFamily.
- FontFamily (Stream ttfStream)

  Create a new FontFamily.
- FontFamily (TrueTypeFile ttf)

  Create a new FontFamily.
- FontFamily (StandardFontFamilies standardFontFamily)

  Create a new standard FontFamily.

Static Public Member Functions

- static FontFamily ResolveFontFamily (string fontFamily)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised. Equivalent to DefaultFontLibrary.ResolveFontFamily.
- static FontFamily ResolveFontFamily (StandardFontFamilies standardFontFamily)

  Create a new font family from the specified standard font family name. Equivalent to DefaultFontLibrary.ResolveFontFamily.
- static FontFamily ResolveFontFamily (string fontFamily, params string[ ] fallback)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the fallback . If none of the fallback family names or true type files are valid, an exception might be raised. Equivalent to DefaultFontLibrary.ResolveFontFamily.
- static FontFamily ResolveFontFamily (string fontFamily, StandardFontFamilies finalFallback, params string[ ] fallback)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the fallback . If none of the fallback family names or true type files are valid, instantiate a standard font family using the finalFallback . Equivalent to DefaultFontLibrary.ResolveFontFamily.

Static Public Attributes

- static string[ ] StandardFamilies = new string[ ] { "Times-Roman", "Times-Bold", "Times-Italic", "Times-Bold↩ Italic", "Helvetica", "Helvetica-Bold", "Helvetica-Oblique", "Helvetica-BoldOblique", "Courier", "Courier-Bold", "Courier-Oblique", "Courier-BoldOblique", "Symbol", "ZapfDingbats" }

  The names of the 14 standard families that are guaranteed to be displayed correctly.
- static string[ ] StandardFontFamilyResources

  The names of the resource streams pointing to the included TrueType font files for each of the standard 14 font families.

Properties

- static IFontLibrary DefaultFontLibrary = new DefaultFontLibrary() `[get, set]`

  The default font library used to resolve font family names.
- bool IsStandardFamily `[get]`

  Whether this is one of the 14 standard font families or not.
- string FileName `[get]`

  Full path to the TrueType font file for this font family (or, if this is a standard font family, name of the font family).
- string FamilyName `[get]`

  Name of the font family, including any variantes.
- TrueTypeFile TrueTypeFile `[get]`

  Parsed TrueType font file for this font family. See also:
  See also

  VectSharp.TrueTypeFile

  .
- bool IsBold `[get]`

  Whether this font is bold or not. This is set based on the information included in the OS/2 table of the TrueType file.
- bool IsItalic `[get]`

  Whether this font is italic or oblique or not. This is set based on the information included in the OS/2 table of the TrueType file.
- bool IsOblique `[get]`

  Whether this font is oblique or not. This is set based on the information included in the OS/2 table of the TrueType file.

7.28.1 Detailed Description

Represents a typeface.

Definition at line 420 of file Font.cs.

7.28.2 Member Enumeration Documentation

7.28.2.1 StandardFontFamilies

`enum VectSharp.FontFamily.StandardFontFamilies`

The 14 standard font families.

Definition at line 499 of file Font.cs.

7.28.3 Constructor & Destructor Documentation

7.28.3.1 FontFamily() [1/4]

```
VectSharp.FontFamily.FontFamily (
            string fileName )
```

Create a new FontFamily.

Parameters

fileName	The full path to the TrueType font file for this font family or the name of a standard font family.

Definition at line 608 of file Font.cs.

7.28.3.2 FontFamily() [2/4]

```
VectSharp.FontFamily.FontFamily (
            Stream ttfStream )
```

Create a new FontFamily.

Parameters

ttfStream	A stream containing a file in TTF format.

Definition at line 633 of file Font.cs.

7.28.3.3 FontFamily() [3/4]

```
VectSharp.FontFamily.FontFamily (
            TrueTypeFile ttf )
```

Create a new FontFamily.

Parameters

ttf	A font file in TTF format.

Definition at line 653 of file Font.cs.

7.28.3.4 FontFamily() [4/4]

```
VectSharp.FontFamily.FontFamily (
            StandardFontFamilies standardFontFamily )
```

Create a new standard FontFamily.

Parameters

standardFontFamily	The standard font family.

Definition at line 674 of file Font.cs.

7.28.4 Member Function Documentation

7.28.4.1 ResolveFontFamily() [1/4]

```
static FontFamily VectSharp.FontFamily.ResolveFontFamily (
            StandardFontFamilies standardFontFamily )  [static]
```

Create a new font family from the specified standard font family name. Equivalent to DefaultFontLibrary.ResolveFontFamily.

Parameters

standardFontFamily	The standard name of the font family.

Returns

A FontFamily object corresponding to the specified font family.

7.28.4.2 ResolveFontFamily() [2/4]

```
static FontFamily VectSharp.FontFamily.ResolveFontFamily (
            string fontFamily )  [static]
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised. Equivalent to DefaultFontLibrary.ResolveFontFamily.

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.

Returns

If the font family name or the true type file is valid, a FontFamily object corresponding to the specified font family.

7.28.4.3 ResolveFontFamily() [3/4]

```
static FontFamily VectSharp.FontFamily.ResolveFontFamily (
            string fontFamily,
            params string[] fallback )  [static]
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the *fallback* . If none of the fallback family names or true type files are valid, an exception might be raised. Equivalent to DefaultFontLibrary.ResolveFontFamily.

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.
fallback	Names of additional font families or TTF files, which will be tried if the first *fontFamily* is not valid.

Returns

A FontFamily object corresponding to the first of the specified font families that is valid.

7.28.4.4 ResolveFontFamily() [4/4]

```
static FontFamily VectSharp.FontFamily.ResolveFontFamily (
            string fontFamily,
            StandardFontFamilies finalFallback,
            params string[] fallback )  [static]
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the *fallback* . If none of the fallback family names or true type files are valid, instantiate a standard font family using the *finalFallback* . Equivalent to DefaultFontLibrary.ResolveFontFamily.

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.
fallback	Names of additional font families or TTF files, which will be tried if the first *fontFamily* is not valid.
finalFallback	The standard name of the font family that will be used if none of the fallback families are valid.

Returns

A FontFamily object corresponding to the first of the specified font families that is valid.

7.28.5 Member Data Documentation

7.28.5.1 StandardFamilies

```
string [] VectSharp.FontFamily.StandardFamilies = new string[] { "Times-Roman", "Times-Bold",
"Times-Italic", "Times-BoldItalic", "Helvetica", "Helvetica-Bold", "Helvetica-Oblique", "Helvetica-Bold↩
Oblique", "Courier", "Courier-Bold", "Courier-Oblique", "Courier-BoldOblique", "Symbol", "Zapf↩
Dingbats" }  [static]
```

The names of the 14 standard families that are guaranteed to be displayed correctly.

Definition at line 478 of file Font.cs.

7.28.5.2 StandardFontFamilyResources

```
string [] VectSharp.FontFamily.StandardFontFamilyResources  [static]
```

Initial value:

```
= new string[]
        {
            "VectSharp.StandardFonts.Tinos-Regular.ttf", "VectSharp.StandardFonts.Tinos-Bold.ttf",
       "VectSharp.StandardFonts.Tinos-Italic.ttf", "VectSharp.StandardFonts.Tinos-BoldItalic.ttf",
            "VectSharp.StandardFonts.Arimo-Regular.ttf", "VectSharp.StandardFonts.Arimo-Bold.ttf",
       "VectSharp.StandardFonts.Arimo-Italic.ttf", "VectSharp.StandardFonts.Arimo-BoldItalic.ttf",
            "VectSharp.StandardFonts.Cousine-Regular.ttf", "VectSharp.StandardFonts.Cousine-Bold.ttf",
       "VectSharp.StandardFonts.Cousine-Italic.ttf", "VectSharp.StandardFonts.Cousine-BoldItalic.ttf",
            "VectSharp.StandardFonts.SymbolNeu_GB.ttf", "VectSharp.StandardFonts.Levibats-Regular_GB.ttf"
        }
```

The names of the resource streams pointing to the included TrueType font files for each of the standard 14 font families.

Definition at line 483 of file Font.cs.

7.28.6 Property Documentation

7.28.6.1 DefaultFontLibrary

```
IFontLibrary VectSharp.FontFamily.DefaultFontLibrary = new DefaultFontLibrary()  [static],
[get], [set]
```

The default font library used to resolve font family names.

Definition at line 425 of file Font.cs.

7.28.6.2 FamilyName

```
string VectSharp.FontFamily.FamilyName [get]
```

Name of the font family, including any variantes.

Definition at line 580 of file Font.cs.

7.28.6.3 FileName

```
string VectSharp.FontFamily.FileName [get]
```

Full path to the TrueType font file for this font family (or, if this is a standard font family, name of the font family).

Definition at line 575 of file Font.cs.

7.28.6.4 IsBold

```
bool VectSharp.FontFamily.IsBold [get]
```

Whether this font is bold or not. This is set based on the information included in the OS/2 table of the TrueType file.

Definition at line 591 of file Font.cs.

7.28.6.5 IsItalic

```
bool VectSharp.FontFamily.IsItalic [get]
```

Whether this font is italic or oblique or not. This is set based on the information included in the OS/2 table of the TrueType file.

Definition at line 596 of file Font.cs.

7.28.6.6 IsOblique

```
bool VectSharp.FontFamily.IsOblique [get]
```

Whether this font is oblique or not. This is set based on the information included in the OS/2 table of the TrueType file.

Definition at line 601 of file Font.cs.

7.28.6.7 IsStandardFamily

```
bool VectSharp.FontFamily.IsStandardFamily [get]
```

Whether this is one of the 14 standard font families or not.

Definition at line 494 of file Font.cs.

7.28.6.8 TrueTypeFile

```
TrueTypeFile VectSharp.FontFamily.TrueTypeFile [get]
```

Parsed TrueType font file for this font family. See also:

See also

VectSharp.TrueTypeFile

.

Definition at line 586 of file Font.cs.

The documentation for this class was generated from the following file:

- VectSharp/Font.cs

7.29 VectSharp.FontFamilyCreationException Class Reference

An exception that occurs while creating a FontFamily.

Inheritance diagram for VectSharp.FontFamilyCreationException:

Public Member Functions

- FontFamilyCreationException (string fontFamily)

  Create a new FontFamilyCreationException instance.

Properties

- string FontFamily `[get]`

  The name of the font family that was being created.

7.29.1 Detailed Description

An exception that occurs while creating a FontFamily.

Definition at line 442 of file FontLibrary.cs.

7.29.2 Constructor & Destructor Documentation

7.29.2.1 FontFamilyCreationException()

```
VectSharp.FontFamilyCreationException.FontFamilyCreationException (
            string fontFamily )
```

Create a new FontFamilyCreationException instance.

Parameters

fontFamily	The name of the font family that was being created.

Definition at line 453 of file FontLibrary.cs.

7.29.3 Property Documentation

7.29.3.1 FontFamily

```
string VectSharp.FontFamilyCreationException.FontFamily [get]
```

The name of the font family that was being created.

Definition at line 447 of file FontLibrary.cs.

The documentation for this class was generated from the following file:

- VectSharp/FontLibrary.cs

7.30 VectSharp.FontLibrary Class Reference

Abstract class with a default implementation of font family fallbacks.

Inheritance diagram for VectSharp.FontLibrary:

Public Member Functions

- abstract FontFamily ResolveFontFamily (string fontFamily)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised.

Parameters

fontFamily	*The name of the font family to create, or the path to a TTF file.*

Returns

If the font family name or the true type file is valid, a FontFamily object corresponding to the specified font family.

- abstract FontFamily ResolveFontFamily (FontFamily.StandardFontFamilies standardFontFamily)

  Create a new font family from the specified standard font family name.

Parameters

standardFontFamily	*The standard name of the font family.*

Returns

A FontFamily object corresponding to the specified font family.

- virtual FontFamily ResolveFontFamily (string fontFamily, params string[] fallback)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the fallback . If none of the fallback family names or true type files are valid, an exception might be raised.

Parameters

fontFamily	*The name of the font family to create, or the path to a TTF file.*
fallback	*Names of additional font families or TTF files, which will be tried if the first fontFamily is not valid.*

Returns

A FontFamily object corresponding to the first of the specified font families that is valid.

- virtual FontFamily ResolveFontFamily (string fontFamily, FontFamily.StandardFontFamilies finalFallback, params string[ ] fallback)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the fallback . If none of the fallback family names or true type files are valid, instantiate a standard font family using the finalFallback .

Parameters

fontFamily	*The name of the font family to create, or the path to a TTF file.*
fallback	*Names of additional font families or TTF files, which will be tried if the first fontFamily is not valid.*
finalFallback	*The standard name of the font family that will be used if none of the fallback families are valid.*

Returns

A FontFamily object corresponding to the first of the specified font families that is valid.

7.30.1 Detailed Description

Abstract class with a default implementation of font family fallbacks.

Definition at line 68 of file FontLibrary.cs.

7.30.2 Member Function Documentation

7.30.2.1 ResolveFontFamily() [1/4]

```
abstract FontFamily VectSharp.FontLibrary.ResolveFontFamily (
            FontFamily.StandardFontFamilies standardFontFamily )  [pure virtual]
```

Create a new font family from the specified standard font family name.

Parameters

standardFontFamily	The standard name of the font family.

Returns

A FontFamily object corresponding to the specified font family.

Implements VectSharp.IFontLibrary.

Implemented in VectSharp.SimpleFontLibrary, and VectSharp.DefaultFontLibrary.

7.30.2.2 ResolveFontFamily() [2/4]

```
abstract FontFamily VectSharp.FontLibrary.ResolveFontFamily (
            string fontFamily )  [pure virtual]
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised.

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.

Returns

If the font family name or the true type file is valid, a FontFamily object corresponding to the specified font family.

Implements VectSharp.IFontLibrary.

Implemented in VectSharp.SimpleFontLibrary, and VectSharp.DefaultFontLibrary.

7.30.2.3 ResolveFontFamily() [3/4]

```
virtual FontFamily VectSharp.FontLibrary.ResolveFontFamily (
            string fontFamily,
            FontFamily.StandardFontFamilies finalFallback,
            params string[] fallback )  [virtual]
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the *fallback* . If none of the fallback family names or true type files are valid, instantiate a standard font family using the *finalFallback* .

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.
fallback	Names of additional font families or TTF files, which will be tried if the first *fontFamily* is not valid.
finalFallback	The standard name of the font family that will be used if none of the fallback families are valid.

Returns

A FontFamily object corresponding to the first of the specified font families that is valid.

Implements VectSharp.IFontLibrary.

Definition at line 130 of file FontLibrary.cs.

7.30.2.4 ResolveFontFamily() [4/4]

```
virtual FontFamily VectSharp.FontLibrary.ResolveFontFamily (
            string fontFamily,
            params string[] fallback )  [virtual]
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the *fallback* . If none of the fallback family names or true type files are valid, an exception might be raised.

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.
fallback	Names of additional font families or TTF files, which will be tried if the first *fontFamily* is not valid.

Returns

A FontFamily object corresponding to the first of the specified font families that is valid.

Implements VectSharp.IFontLibrary.

Definition at line 77 of file FontLibrary.cs.

The documentation for this class was generated from the following file:

- VectSharp/FontLibrary.cs

7.31 VectSharp.Font.FontUnderline Class Reference

Represents options to underline text.

Properties

- bool SkipDescenders [get, set]

  Determines whether the underline skips the parts of the glyph that would intersect with it or not.
- double Position [get, set]

  Determines the position of the top of the underline with respect to the text baseline. Positive values are below the baseline, negative values are above it. This is expressed as a fraction of the font size.
- double Thickness [get, set]

  Determines the thickness of the underline, expressed as a fraction of the font size.
- LineCaps LineCap [get, set]

  Determines the caps at the start and end of the underline.
- bool FollowItalicAngle [get, set]

  Determine whether the shape of the underline is slanted to follow the angle of italic fonts.

7.31.1 Detailed Description

Represents options to underline text.

Definition at line 38 of file Font.cs.

7.31.2 Property Documentation

7.31.2.1 FollowItalicAngle

```
bool VectSharp.Font.FontUnderline.FollowItalicAngle  [get], [set]
```

Determine whether the shape of the underline is slanted to follow the angle of italic fonts.

Definition at line 63 of file Font.cs.

7.31.2.2 LineCap

```
LineCaps VectSharp.Font.FontUnderline.LineCap  [get], [set]
```

Determines the caps at the start and end of the underline.

Definition at line 58 of file Font.cs.

7.31.2.3 Position

```
double VectSharp.Font.FontUnderline.Position  [get], [set]
```

Determines the position of the top of the underline with respect to the text baseline. Positive values are below the baseline, negative values are above it. This is expressed as a fraction of the font size.

Definition at line 48 of file Font.cs.

7.31.2.4 SkipDescenders

```
bool VectSharp.Font.FontUnderline.SkipDescenders  [get], [set]
```

Determines whether the underline skips the parts of the glyph that would intersect with it or not.

Definition at line 43 of file Font.cs.

7.31.2.5 Thickness

```
double VectSharp.Font.FontUnderline.Thickness [get], [set]
```

Determines the thickness of the underline, expressed as a fraction of the font size.

Definition at line 53 of file Font.cs.

The documentation for this class was generated from the following file:

- VectSharp/Font.cs

7.32 VectSharp.Markdown.FormattedString Struct Reference

Represents a string with associated formatting information.

Public Member Functions

- FormattedString (string text, Colour colour, bool isBold, bool isItalic)

  Creates a new FormattedString instance.

Properties

- string Text [get]

  The text represented by this object.
- Colour Colour [get]

  The colour of the text.
- bool IsBold [get]

  Whether the text should be rendered as bold or not.
- bool IsItalic [get]

  Whether the text should be rendered as italic or not.

7.32.1 Detailed Description

Represents a string with associated formatting information.

Definition at line 32 of file SyntaxHighlighting.cs.

7.32.2 Constructor & Destructor Documentation

7.32.2.1 FormattedString()

```
VectSharp.Markdown.FormattedString.FormattedString (
            string text,
            Colour colour,
            bool isBold,
            bool isItalic )
```

Creates a new FormattedString instance.

Parameters

text	The text of the object.
colour	The colour of the text.
isBold	Whether the text should be rendered as bold or not.
isItalic	Whether the text should be rendered as italic or not.

Definition at line 61 of file SyntaxHighlighting.cs.

7.32.3 Property Documentation

7.32.3.1 Colour

```
Colour VectSharp.Markdown.FormattedString.Colour  [get]
```

The colour of the text.

Definition at line 42 of file SyntaxHighlighting.cs.

7.32.3.2 IsBold

```
bool VectSharp.Markdown.FormattedString.IsBold  [get]
```

Whether the text should be rendered as bold or not.

Definition at line 47 of file SyntaxHighlighting.cs.

7.32.3.3 IsItalic

```
bool VectSharp.Markdown.FormattedString.IsItalic  [get]
```

Whether the text should be rendered as italic or not.

Definition at line 52 of file SyntaxHighlighting.cs.

7.32.3.4 Text

```
string VectSharp.Markdown.FormattedString.Text [get]
```

The text represented by this object.

Definition at line 37 of file SyntaxHighlighting.cs.

The documentation for this struct was generated from the following file:

- VectSharp.Markdown/SyntaxHighlighting.cs

7.33 VectSharp.FormattedText Class Reference

Represents a run of text that should be drawn with the same style.

Public Member Functions

- FormattedText (string text, Font font, Script script=Script.Normal, Brush brush=null)

  Creates a new FormattedText instance with the specified text , font , script position and brush .

Static Public Member Functions

- static IEnumerable< FormattedText > Format (string text, Font normalFont, Font boldFont, Font italicFont, Font boldItalicFont, Brush defaultBrush=null)

  Parse the formatting information contained in a text string into a collection of FormattedText objects.
- static IEnumerable< FormattedText > Format (string text, FontFamily.StandardFontFamilies fontFamily, double fontSize, bool defaultUnderline=false, Brush defaultBrush=null)

  Parse the formatting information contained in a text string into a collection of FormattedText objects, using fonts from a standard font family.

Properties

- string Text [get]

  Represents the text represented by this instance.
- Font Font [get]

  Represents the font that should be used to draw the text.
- Script Script [get]

  Represents the position of the text.
- Brush Brush [get]

  Represents the brush that should be used to draw the text. If this is null, the default brush is used.

7.33.1 Detailed Description

Represents a run of text that should be drawn with the same style.

Definition at line 50 of file FormattedText.cs.

7.33.2 Constructor & Destructor Documentation

7.33.2.1 FormattedText()

```
VectSharp.FormattedText.FormattedText (
            string text,
            Font font,
            Script script = Script.Normal,
            Brush brush = null )
```

Creates a new FormattedText instance with the specified *text* , *font* , *script* position and *brush* .

Parameters

text	The text that will be contained in the new FormattedText.
font	The font that will be used by the new FormattedText.
script	The script position of the new FormattedText.
brush	The brush that will be used by the new FormattedText.

Definition at line 79 of file FormattedText.cs.

7.33.3 Member Function Documentation

7.33.3.1 Format() [1/2]

```
static IEnumerable< FormattedText > VectSharp.FormattedText.Format (
            string text,
            Font normalFont,
            Font boldFont,
            Font italicFont,
            Font boldItalicFont,
            Brush defaultBrush = null )  [static]
```

Parse the formatting information contained in a text string into a collection of FormattedText objects.

Parameters

text	The string containing formatting information. Format information is specified using HTML-like tags: • `<b></b>` or `<strong></strong>` are used for bold text; • `<i></i>` or `<em></em>` are used for text in italics; • `<u></u>` are used for underlined text; • `<sup></sup>` and `<sub></sub>` are used, respectively, for superscript and subscript text; • `<#COLOUR></#>` is used to specify the colour of the text, where `COLOUR` is a CSS colour string (e.g. `<#red>`, `<#0080FF>`, or `<#rgba(128, 80, 52, 0.5)>`).
normalFont	The font that will be used for text that is neither bold nor italic.
boldFont	The font that will be used for text that is bold. Note that this does not necessarily have to be a bold font; this is just the font that is applied to text contained within `<b></b>` tags.
italicFont	The font that will be used for text that is in italics. Note that this does not necessarily have to be an italic font; this is just the font that is applied to text contained within `<i></i>` tags.
boldItalicFont	The font that will be used for text that is both in bold and in italics.
defaultBrush	The default Brush that will be used for text runs that do not specify a colour. If this is `null`, the default Brush will be the one specified in the painting call.

Returns

A lazy collection of FormattedText objects. Note that every enumeration of this collection causes the text to be parsed again; if you need to enumerate this collection more than once, you should probably convert it e.g. to a List<T>.

Definition at line 105 of file FormattedText.cs.

7.33.3.2 Format() [2/2]

```
static IEnumerable< FormattedText > VectSharp.FormattedText.Format (
            string text,
            FontFamily.StandardFontFamilies fontFamily,
            double fontSize,
            bool defaultUnderline = false,
            Brush defaultBrush = null )  [static]
```

Parse the formatting information contained in a text string into a collection of FormattedText objects, using fonts from a standard font family.

Parameters

text	The string containing formatting information. Format information is specified using HTML-like tags: • or are used for bold text; • or are used for text in italics; • <u></u> are used for underlined text; • and are used, respectively, for superscript and subscript text; • <#COLOUR></#> is used to specify the colour of the text, where COLOUR is a CSS colour string (e.g. <#red>, <#0080FF>, or <#rgba(128, 80, 52, 0.5)>).
fontFamily	The font family from which the fonts will be created. If this is a regular font family, the bold, italic and bold-italic versions of the font will be used for the formatted text. Otherwise, the relevant font styles will be toggled (e.g. if the supplied font family is bold, then regular text in the formatted string will be displayed as bold, while bold text in the formatted string will be displayed as regular text).
fontSize	The size of the fonts to use.
defaultUnderline	Determines whether text should be underlined by default. This is toggled by <u></u> tags.
defaultBrush	The default Brush that will be used for text runs that do not specify a colour. If this is null, the default Brush will be the one specified in the painting call.

Returns

A lazy collection of FormattedText objects. Note that every enumeration of this collection causes the text to be parsed again; if you need to enumerate this collection more than once, you should probably convert it e.g. to a List<T>.

Definition at line 282 of file FormattedText.cs.

7.33.4 Property Documentation

7.33.4.1 Brush

```
Brush VectSharp.FormattedText.Brush  [get]
```

Represents the brush that should be used to draw the text. If this is null, the default brush is used.

Definition at line 70 of file FormattedText.cs.

7.33.4.2 Font

```
Font VectSharp.FormattedText.Font [get]
```

Represents the font that should be used to draw the text.

Definition at line 60 of file FormattedText.cs.

7.33.4.3 Script

```
Script VectSharp.FormattedText.Script [get]
```

Represents the position of the text.

Definition at line 65 of file FormattedText.cs.

7.33.4.4 Text

```
string VectSharp.FormattedText.Text [get]
```

Represents the text represented by this instance.

Definition at line 55 of file FormattedText.cs.

The documentation for this class was generated from the following file:

- VectSharp/FormattedText.cs

7.34 VectSharp.FormattedTextExtensions Class Reference

Contains extension methods for collections of FormattedText objects.

Static Public Member Functions

- static Font.DetailedFontMetrics Measure (this IEnumerable< FormattedText > text)
  Measures a collection of FormattedText objects.

7.34.1 Detailed Description

Contains extension methods for collections of FormattedText objects.

Definition at line 537 of file FormattedText.cs.

7.34.2 Member Function Documentation

7.34.2.1 Measure()

```
static Font.DetailedFontMetrics VectSharp.FormattedTextExtensions.Measure (
            this IEnumerable< FormattedText > text ) [static]
```

Measures a collection of FormattedText objects.

Parameters

text	The collection of FormattedText objects to be measured.

Returns

A Font.DetailedFontMetrics containing detailed measurements for the text obtained by composing the elements in the FormattedText collection.

Definition at line 626 of file FormattedText.cs.

The documentation for this class was generated from the following file:

- VectSharp/FormattedText.cs

7.35 VectSharp.Frame Class Reference

A key frame for an animation.

Public Member Functions

- Frame (Graphics graphics, double duration)

  Creates a new Frame with the specified contents and duration.

Properties

- double Duration `[get]`

  The duration of the frame, in milliseconds.
- Graphics Graphics `[get]`

  The contents of the frame.

7.35.1 Detailed Description

A key frame for an animation.

Definition at line 1234 of file Animation.cs.

7.35.2 Constructor & Destructor Documentation

7.35.2.1 Frame()

```
VectSharp.Frame.Frame (
            Graphics graphics,
            double duration )
```

Creates a new Frame with the specified contents and duration.

Parameters

graphics	The contents of the frame.
duration	The duration of the frame, in milliseconds.

Definition at line 1251 of file Animation.cs.

7.35.3 Property Documentation

7.35.3.1 Duration

```
double VectSharp.Frame.Duration [get]
```

The duration of the frame, in milliseconds.

Definition at line 1239 of file Animation.cs.

7.35.3.2 Graphics

```
Graphics VectSharp.Frame.Graphics [get]
```

The contents of the frame.

Definition at line 1244 of file Animation.cs.

The documentation for this class was generated from the following file:

- VectSharp/Animation.cs

7.36 VectSharp.Filters.GaussianBlurFilter Class Reference

Represents a filter that applies a Gaussian blur effect.

Inheritance diagram for VectSharp.Filters.GaussianBlurFilter:

Public Member Functions

- GaussianBlurFilter (double standardDeviation)

  Creates a new GaussianBlurFilter with the specified standard deviation.
- RasterImage Filter (RasterImage image, double scale)

  Applies the filter to a RasterImage.

Parameters

image	*The RasterImage to which the filter will be applied.*
scale	*The scale of the image with respect to the filter.*

Returns

A new RasterImage containing the filtered image. The source image is left unaltered.

Properties

- double StandardDeviation `[get]`

  The standard deviation of the Gaussian blur.
- Point TopLeftMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.
- Point BottomRightMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

7.36.1 Detailed Description

Represents a filter that applies a Gaussian blur effect.

Definition at line 26 of file GaussianBlurFilter.cs.

7.36.2 Constructor & Destructor Documentation

7.36.2.1 GaussianBlurFilter()

```
VectSharp.Filters.GaussianBlurFilter.GaussianBlurFilter (
            double standardDeviation )
```

Creates a new GaussianBlurFilter with the specified standard deviation.

Parameters

standardDeviation	The standard deviation of the Gaussian blur.

Definition at line 43 of file GaussianBlurFilter.cs.

7.36.3 Member Function Documentation

7.36.3.1 Filter()

```
RasterImage VectSharp.Filters.GaussianBlurFilter.Filter (
            RasterImage image,
            double scale )
```

Applies the filter to a RasterImage.

Parameters

image	The RasterImage to which the filter will be applied.
scale	The scale of the image with respect to the filter.

Returns

A new RasterImage containing the filtered image. The source *image* is left unaltered.

Implements VectSharp.Filters.ILocationInvariantFilter.

Definition at line 51 of file GaussianBlurFilter.cs.

7.36.4 Property Documentation

7.36.4.1 BottomRightMargin

```
Point VectSharp.Filters.GaussianBlurFilter.BottomRightMargin [get]
```

Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 37 of file GaussianBlurFilter.cs.

7.36.4.2 StandardDeviation

```
double VectSharp.Filters.GaussianBlurFilter.StandardDeviation [get]
```

The standard deviation of the Gaussian blur.

Definition at line 31 of file GaussianBlurFilter.cs.

7.36.4.3 TopLeftMargin

```
Point VectSharp.Filters.GaussianBlurFilter.TopLeftMargin [get]
```

Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 34 of file GaussianBlurFilter.cs.

The documentation for this class was generated from the following file:

- VectSharp/Filters/GaussianBlurFilter.cs

7.37 VectSharp.GradientBrush Class Reference

Represents a brush painting with a gradient.

Inheritance diagram for VectSharp.GradientBrush:

Properties

- GradientStops GradientStops [get]
  The colour stops in the gradient.

Additional Inherited Members

7.37.1 Detailed Description

Represents a brush painting with a gradient.

Definition at line 264 of file Brush.cs.

7.37.2 Property Documentation

7.37.2.1 GradientStops

```
GradientStops VectSharp.GradientBrush.GradientStops [get]
```

The colour stops in the gradient.

Definition at line 269 of file Brush.cs.

The documentation for this class was generated from the following file:

- VectSharp/Brush.cs

7.38 VectSharp.GradientStop Struct Reference

Represents a colour stop in a gradient.

Public Member Functions

- GradientStop (Colour colour, double offset)

  Creates a new GradientStop instance.
- GradientStop MultiplyOpacity (double opacity)

  Returns a GradientStop corresponding to the current instance, whose colour's opacity has been multiplied by the specified value.

Properties

- Colour Colour [get]

  The Colour at the gradient stop.
- double Offset [get]

  The offset of the gradient stop. Range: [0, 1].

7.38.1 Detailed Description

Represents a colour stop in a gradient.

Definition at line 109 of file Brush.cs.

7.38.2 Constructor & Destructor Documentation

7.38.2.1 GradientStop()

```
VectSharp.GradientStop.GradientStop (
            Colour colour,
            double offset )
```

Creates a new GradientStop instance.

Parameters

colour	The Colour at the gradient stop.
offset	The offset of the gradient stop. Range: [0, 1].

Definition at line 126 of file Brush.cs.

7.38.3 Member Function Documentation

7.38.3.1 MultiplyOpacity()

```
GradientStop VectSharp.GradientStop.MultiplyOpacity (
            double opacity )
```

Returns a GradientStop corresponding to the current instance, whose colour's opacity has been multiplied by the specified value.

Parameters

opacity	The value that will be used to multiply the colour's opacity.

Returns

A GradientStop corresponding to the current instance, whose colour's opacity has been multiplied by the specified value.

Definition at line 137 of file Brush.cs.

7.38.4 Property Documentation

7.38.4.1 Colour

```
Colour VectSharp.GradientStop.Colour  [get]
```

The Colour at the gradient stop.

Definition at line 114 of file Brush.cs.

7.38.4.2 Offset

```
double VectSharp.GradientStop.Offset  [get]
```

The offset of the gradient stop. Range: [0, 1].

Definition at line 119 of file Brush.cs.

The documentation for this struct was generated from the following file:

- VectSharp/Brush.cs

7.39 VectSharp.GradientStops Class Reference

Represents a read-only list of GradientStops.

Inheritance diagram for VectSharp.GradientStops:

Public Member Functions

- IEnumerator< GradientStop > GetEnumerator ()
- GradientStops (IEnumerable< GradientStop > gradientStops)

  Creates a new GradientStops instance containing the specified gradient stops.
- GradientStops (params GradientStop[ ] gradientStops)

  Creates a new GradientStops instance containing the specified gradient stops.
- Colour GetColourAt (double position)

  Gets the colour at a certain position on the gradient.

Static Public Attributes

- static readonly double StopTolerance = 1e-7

  The minimum distance that is enforced between consecutive gradient stops.

Properties

- GradientStop this[int index] [get]
- int Count [get]

7.39.1 Detailed Description

Represents a read-only list of GradientStops.

Definition at line 146 of file Brush.cs.

7.39.2 Constructor & Destructor Documentation

7.39.2.1 GradientStops() [1/2]

```
VectSharp.GradientStops.GradientStops (
            IEnumerable< GradientStop > gradientStops )
```

Creates a new GradientStops instance containing the specified gradient stops.

Parameters

gradientStops	The gradient stops that will be contained in the GradientStops object.

Definition at line 176 of file Brush.cs.

7.39.2.2 GradientStops() [2/2]

```
VectSharp.GradientStops.GradientStops (
            params GradientStop[] gradientStops )
```

Creates a new GradientStops instance containing the specified gradient stops.

Parameters

gradientStops	The gradient stops that will be contained in the GradientStops object.

Definition at line 222 of file Brush.cs.

7.39.3 Member Function Documentation

7.39.3.1 GetColourAt()

```
Colour VectSharp.GradientStops.GetColourAt (
            double position )
```

Gets the colour at a certain position on the gradient.

Parameters

position	The position in the gradient (ranging from 0 to 1).

Returns

The colour of the gradient at the specified position.

Definition at line 232 of file Brush.cs.

7.39.3.2 GetEnumerator()

```
IEnumerator< GradientStop > VectSharp.GradientStops.GetEnumerator ( )
```

Definition at line 162 of file Brush.cs.

7.39.4 Member Data Documentation

7.39.4.1 StopTolerance

```
readonly double VectSharp.GradientStops.StopTolerance = 1e-7  [static]
```

The minimum distance that is enforced between consecutive gradient stops.

Definition at line 151 of file Brush.cs.

7.39.5 Property Documentation

7.39.5.1 Count

```
int VectSharp.GradientStops.Count  [get]
```

Definition at line 157 of file Brush.cs.

7.39.5.2 this[int index]

```
GradientStop VectSharp.GradientStops.this[int index]  [get]
```

Definition at line 154 of file Brush.cs.

The documentation for this class was generated from the following file:

- VectSharp/Brush.cs

7.40 VectSharp.Graphics Class Reference

Represents an abstract drawing surface.

Public Member Functions

- void FillPath (GraphicsPath path, Brush fillColour, string tag=null)

  Fill a GraphicsPath.
- void StrokePath (GraphicsPath path, Brush strokeColour, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins lineJoin=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke a GraphicsPath.
- void SetClippingPath (GraphicsPath path, string tag=null)

  Intersect the current clipping path with the specified GraphicsPath.
- void SetClippingPath (double leftX, double topY, double width, double height, string tag=null)

  Intersect the current clipping path with the specified rectangle.
- void SetClippingPath (Point topLeft, Size size, string tag=null)

  Intersect the current clipping path with the specified rectangle.
- void Rotate (double angle, string tag=null)

  Rotate the coordinate system around the origin.
- void RotateAt (double angle, Point pivot, string tag=null)

  Rotate the coordinate system around a pivot point.
- void Transform (double a, double b, double c, double d, double e, double f, string tag=null)

  Transform the coordinate system with the specified transformation matrix [[a, c, e], [b, d, f], [0, 0, 1]].
- void Translate (double x, double y, string tag=null)

  Translate the coordinate system origin.
- void Translate (Point delta, string tag=null)

  Translate the coordinate system origin.
- void Scale (double scaleX, double scaleY, string tag=null)

  Scale the coordinate system with respect to the origin.
- void FillRectangle (Point topLeft, Size size, Brush fillColour, string tag=null)

  Fill a rectangle.
- void FillRectangle (double leftX, double topY, double width, double height, Brush fillColour, string tag=null)

  Fill a rectangle.
- void StrokeRectangle (Point topLeft, Size size, Brush strokeColour, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins lineJoin=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke a rectangle.
- void StrokeRectangle (double leftX, double topY, double width, double height, Brush strokeColour, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins lineJoin=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

Stroke a rectangle.

- void DrawRasterImage (int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image, string tag=null)

  Draw a raster image.
- void DrawRasterImage (double x, double y, RasterImage image, string tag=null)

  Draw a raster image.
- void DrawRasterImage (Point position, RasterImage image, string tag=null)

  Draw a raster image.
- void DrawRasterImage (double x, double y, double width, double height, RasterImage image, string tag=null)

  Draw a raster image.
- void DrawRasterImage (Point position, Size size, RasterImage image, string tag=null)

  Draw a raster image.
- void Save ()

  Save the current transform state (rotation, translation, scale).
- void Restore ()

  Restore the previous transform state (rotation, translation scale).
- void CopyToIGraphicsContext (IGraphicsContext destinationContext)

  Copy the current graphics to an instance of a class implementing IGraphicsContext.
- void DrawGraphics (Point origin, Graphics graphics)

  Draws a Graphics object on the current Graphics object.
- void DrawGraphics (double originX, double originY, Graphics graphics, string tag)

  Draws a Graphics object on the current Graphics object, prepending the supplied tag to the tags contained in the Graphics object being drawn.
- void DrawGraphics (Point origin, Graphics graphics, string tag)

  Draws a Graphics object on the current Graphics object, prepending the supplied tag to the tags contained in the Graphics object being drawn.
- void DrawGraphics (double originX, double originY, Graphics graphics)

  Draws a Graphics object on the current Graphics object.
- void DrawGraphics (Point origin, Graphics graphics, IFilter filter, string tag=null)

  Draws a Graphics object on the current Graphics object, applying the specified filter .
- void DrawGraphics (double originX, double originY, Graphics graphics, IFilter filter, string tag=null)

  Draws a Graphics object on the current Graphics object, applying the specified filter .
- Graphics Transform (Func< Point, Point > transformationFunction, double linearisationResolution)

  Creates a new Graphics object in which all the graphics actions have been transformed using an arbitrary transformation function. Raster images are replaced by grey rectangles.
- Graphics Transform (Func< Point, Point > transformationFunction, double linearisationResolution, double maxSegmentLength)

  Creates a new Graphics object in which all the graphics actions have been transformed using an arbitrary transformation function. Raster images are replaced by grey rectangles.
- Graphics Linearise (double resolution)

  Creates a new Graphics object by linearising all of the elements of the current instance, i.e. replacing curve segments with series of line segments that approximate them. Raster images are left unchanged.
- Rectangle GetBounds ()

  Computes the rectangular bounds of the region affected by the drawing operations performed on the Graphics object.
- bool TryRasterise (Rectangle region, double scale, bool interpolate, out RasterImage output)

  Tries to rasterise specified region of this Graphics object using the default rasterisation method.
- void Crop (Rectangle region)

  Removes graphics actions that fall completely outside of the specified region .
- void Crop (Point topLeft, Size size)

  Removes graphics actions that fall completely outside of the specified region.
- IEnumerable< string > GetTags ()

  Gets all the tags that have been defined in the Graphics.

- void FillText (Point origin, string text, Font font, Brush fillColour, TextBaselines textBaseline=TextBaselines.↩ Top, string tag=null)

  Fill a text string.
- void FillText (double originX, double originY, string text, Font font, Brush fillColour, TextBaselines text↩ Baseline=TextBaselines.Top, string tag=null)

  Fill a text string.
- void StrokeText (Point origin, string text, Font font, Brush strokeColour, TextBaselines textBaseline=Text↩ Baselines.Top, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins lineJoin=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke a text string.
- void StrokeText (double originX, double originY, string text, Font font, Brush strokeColour, TextBaselines textBaseline=TextBaselines.Top, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins line↩ Join=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke a text string.
- void FillTextOnPath (GraphicsPath path, string text, Font font, Brush fillColour, double reference=0, TextAnchors anchor=TextAnchors.Left, TextBaselines textBaseline=TextBaselines.Top, string tag=null)

  Fill a text string along a GraphicsPath.
- void StrokeTextOnPath (GraphicsPath path, string text, Font font, Brush strokeColour, double reference=0, TextAnchors anchor=TextAnchors.Left, TextBaselines textBaseline=TextBaselines.Top, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins lineJoin=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke a text string along a GraphicsPath.
- void FillText (Point origin, IEnumerable< FormattedText > text, Brush fillColour, TextBaselines text↩ Baseline=TextBaselines.Top, string tag=null)

  Fill a formatted text string.
- void FillText (double originX, double originY, IEnumerable< FormattedText > text, Brush fillColour, TextBaselines textBaseline=TextBaselines.Top, string tag=null)

  Fill a formatted text string.
- void StrokeText (Point origin, IEnumerable< FormattedText > text, Brush strokeColour, TextBaselines textBaseline=TextBaselines.Top, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins line↩ Join=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke a formatted text string.
- void StrokeText (double originX, double originY, IEnumerable< FormattedText > text, Brush stroke↩ Colour, TextBaselines textBaseline=TextBaselines.Top, double lineWidth=1, LineCaps lineCap=LineCaps.↩ Butt, LineJoins lineJoin=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke a formatted text string.
- Size MeasureText (string text, Font font)

  Measure a text string. See also

  See also

  Font.MeasureText(string), Font.MeasureTextAdvanced(string)

  and .
- Size MeasureText (IEnumerable< FormattedText > text)

  Measure a formatted text string. See also

  See also

  FormattedTextExtensions.Measure(IEnumerable<FormattedText>)

  .
- void FillTextUnderline (double originX, double originY, string text, Font font, Brush fillColour, TextBaselines textBaseline=TextBaselines.Top, string tag=null)

  Fills the underline of the specified text string.
- void FillTextUnderline (Point origin, string text, Font font, Brush fillColour, TextBaselines textBaseline=Text↩ Baselines.Top, string tag=null)

  Fills the underline of the specified text string.

- void StrokeTextUnderline (double originX, double originY, string text, Font font, Brush strokeColour, TextBaselines textBaseline=TextBaselines.Top, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins lineJoin=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke the underline of the specified text string.
- void StrokeTextUnderline (Point origin, string text, Font font, Brush strokeColour, TextBaselines textBaseline=TextBaselines.Top, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins lineJoin=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke the underline of the specified text string.
- void FillTextUnderline (double originX, double originY, IEnumerable< FormattedText > text, Brush fillColour, TextBaselines textBaseline=TextBaselines.Top, string tag=null)

  Fill the underline of the specified formatted text string.
- void FillTextUnderline (Point origin, IEnumerable< FormattedText > text, Brush fillColour, TextBaselines textBaseline=TextBaselines.Top, string tag=null)

  Fill the underline of the specified formatted text string.
- void StrokeTextUnderline (double originX, double originY, IEnumerable< FormattedText > text, Brush strokeColour, TextBaselines textBaseline=TextBaselines.Top, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins lineJoin=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke the underline of the specified formatted text string.
- void StrokeTextUnderline (Point origin, IEnumerable< FormattedText > text, Brush strokeColour, TextBaselines textBaseline=TextBaselines.Top, double lineWidth=1, LineCaps lineCap=LineCaps.Butt, LineJoins lineJoin=LineJoins.Miter, LineDash? lineDash=null, string tag=null)

  Stroke the underline of the specified formatted text string.

Static Public Attributes

- static Func< Graphics, Rectangle, double, bool, RasterImage > RasterisationMethod = null

  A method that is used to rasterise a region of a Graphics object. Set this to `null` if you wish to use the default rasterisation methods (implemented by either VectSharp.Raster, or VectSharp.Raster.ImageSharp). You will have to provide your own implementation of this method if neither VectSharp.Raster nor VectSharp.Raster.ImageSharp are referenced by your project. The first argument of this method is the Graphics to be rasterised, the second is a Rectangle representing the region to rasterise, the third is a double representing the scale, and the third is a boolean value indicating whether the resulting RasterImage should be interpolated.

Properties

- static UnbalancedStackActions UnbalancedStackAction = UnbalancedStackActions.Throw `[get, set]`

  Determines how an unbalanced graphics state stack (which occurs if the number of calls to Save and Restore is not equal) will be treated. The default is UnbalancedStackActions.Throw.

7.40.1 Detailed Description

Represents an abstract drawing surface.

Definition at line 261 of file Graphics.cs.

7.40.2 Member Function Documentation

7.40.2.1 CopyToIGraphicsContext()

```
void VectSharp.Graphics.CopyToIGraphicsContext (
            IGraphicsContext destinationContext )
```

Copy the current graphics to an instance of a class implementing IGraphicsContext.

Parameters

destinationContext	The IGraphicsContext on which the graphics are to be copied.

Definition at line 619 of file Graphics.cs.

7.40.2.2 Crop() [1/2]

```
void VectSharp.Graphics.Crop (
            Point topLeft,
            Size size )
```

Removes graphics actions that fall completely outside of the specified region.

Parameters

topLeft	The top-left corner of the area to preserve.
size	The size of the area to preserve.

Definition at line 1645 of file Graphics.cs.

7.40.2.3 Crop() [2/2]

```
void VectSharp.Graphics.Crop (
            Rectangle region )
```

Removes graphics actions that fall completely outside of the specified *region* .

Parameters

region	The area to preserve.

Definition at line 1556 of file Graphics.cs.

7.40.2.4 DrawGraphics() [1/6]

```
void VectSharp.Graphics.DrawGraphics (
            double originX,
            double originY,
            Graphics graphics )
```

Draws a Graphics object on the current Graphics object.

Parameters

originX	The horizontal coordinate at which to place the origin of *graphics* .
originY	The vertical coordinate at which to place the origin of *graphics* .
graphics	The Graphics object to draw on the current Graphics object.

Definition at line 900 of file Graphics.cs.

7.40.2.5 DrawGraphics() [2/6]

```
void VectSharp.Graphics.DrawGraphics (
            double originX,
            double originY,
            Graphics graphics,
            IFilter filter,
            string tag = null )
```

Draws a Graphics object on the current Graphics object, applying the specified *filter* .

Parameters

originX	The horizontal coordinate at which to place the origin of *graphics* .
originY	The vertical coordinate at which to place the origin of *graphics* .
graphics	The Graphics object to draw on the current Graphics object.
filter	An IFilter object, representing the filter to apply to the *graphics* object.
tag	A tag to identify the filter.

Definition at line 942 of file Graphics.cs.

7.40.2.6 DrawGraphics() [3/6]

```
void VectSharp.Graphics.DrawGraphics (
            double originX,
            double originY,
            Graphics graphics,
            string tag )
```

Draws a Graphics object on the current Graphics object, prepending the supplied *tag* to the tags contained in the Graphics object being drawn.

Parameters

originX	The horizontal coordinate at which to place the origin of *graphics* .
originY	The vertical coordinate at which to place the origin of *graphics* .
graphics	The Graphics object to draw on the current Graphics object.
tag	The tag to prepend to the tags contained in the *graphics* object.

Definition at line 846 of file Graphics.cs.

7.40.2.7 DrawGraphics() [4/6]

```
void VectSharp.Graphics.DrawGraphics (
            Point origin,
            Graphics graphics )
```

Draws a Graphics object on the current Graphics object.

Parameters

origin	The point at which to place the origin of *graphics* .
graphics	The Graphics object to draw on the current Graphics object.

Definition at line 827 of file Graphics.cs.

7.40.2.8 DrawGraphics() [5/6]

```
void VectSharp.Graphics.DrawGraphics (
            Point origin,
            Graphics graphics,
            IFilter filter,
            string tag = null )
```

Draws a Graphics object on the current Graphics object, applying the specified *filter* .

Parameters

origin	The point at which to place the origin of *graphics* .
graphics	The Graphics object to draw on the current Graphics object.
filter	An IFilter object, representing the filter to apply to the *graphics* object.
tag	A tag to identify the filter.

Definition at line 912 of file Graphics.cs.

7.40.2.9 DrawGraphics() [6/6]

```
void VectSharp.Graphics.DrawGraphics (
            Point origin,
            Graphics graphics,
            string tag )
```

Draws a Graphics object on the current Graphics object, prepending the supplied *tag* to the tags contained in the Graphics object being drawn.

Parameters

origin	The point at which to place the origin of *graphics* .
graphics	The Graphics object to draw on the current Graphics object.
tag	The tag to prepend to the tags contained in the *graphics* object.

Definition at line 857 of file Graphics.cs.

7.40.2.10 DrawRasterImage() [1/5]

```
void VectSharp.Graphics.DrawRasterImage (
            double x,
            double y,
            double width,
            double height,
            RasterImage image,
            string tag = null )
```

Draw a raster image.

Parameters

x	The horizontal coordinate of the top-left corner of the rectangle delimiting the destination area of the image.
y	The vertical coordinate of the top-left corner of the rectangle delimiting the destination area of the image.
width	The width of the rectangle delimiting the destination area of the image.
height	The height of the rectangle delimiting the destination area of the image.
image	The image to draw.
tag	A tag to identify the drawn image.

Definition at line 524 of file Graphics.cs.

7.40.2.11 DrawRasterImage() [2/5]

```
void VectSharp.Graphics.DrawRasterImage (
            double x,
            double y,
            RasterImage image,
            string tag = null )
```

Draw a raster image.

Parameters

x	The horizontal coordinate of the top-left corner of the rectangle delimiting the destination area of the image.
y	The vertical coordinate of the top-left corner of the rectangle delimiting the destination area of the image.
image	The image to draw.
tag	A tag to identify the drawn image.

Definition at line 499 of file Graphics.cs.

7.40.2.12 DrawRasterImage() [3/5]

```
void VectSharp.Graphics.DrawRasterImage (
            int sourceX,
            int sourceY,
            int sourceWidth,
            int sourceHeight,
            double destinationX,
            double destinationY,
            double destinationWidth,
            double destinationHeight,
            RasterImage image,
            string tag = null )
```

Draw a raster image.

Parameters

sourceX	The horizontal coordinate of the top-left corner of the rectangle delimiting the source area of the image.
sourceY	The vertical coordinate of the top-left corner of the rectangle delimiting the source area of the image.
sourceWidth	The width of the rectangle delimiting the source area of the image.
sourceHeight	The height of the rectangle delimiting the source area of the image.
destinationX	The horizontal coordinate of the top-left corner of the rectangle delimiting the destination area of the image.
destinationY	The vertical coordinate of the top-left corner of the rectangle delimiting the destination area of the image.
destinationWidth	The width of the rectangle delimiting the destination area of the image.
destinationHeight	The height of the rectangle delimiting the destination area of the image.
image	The image to draw.
tag	A tag to identify the drawn image.

Definition at line 487 of file Graphics.cs.

7.40.2.13 DrawRasterImage() [4/5]

```
void VectSharp.Graphics.DrawRasterImage (
            Point position,
            RasterImage image,
            string tag = null )
```

Draw a raster image.

Parameters

position	The the top-left corner of the rectangle delimiting the destination area of the image.
image	The image to draw.
tag	A tag to identify the drawn image.

Definition at line 510 of file Graphics.cs.

7.40.2.14 DrawRasterImage() [5/5]

```
void VectSharp.Graphics.DrawRasterImage (
            Point position,
            Size size,
            RasterImage image,
            string tag = null )
```

Draw a raster image.

Parameters

position	The the top-left corner of the rectangle delimiting the destination area of the image.
size	The size of the rectangle delimiting the destination area of the image.
image	The image to draw.
tag	A tag to identify the drawn image.

Definition at line 536 of file Graphics.cs.

7.40.2.15 FillPath()

```
void VectSharp.Graphics.FillPath (
            GraphicsPath path,
            Brush fillColour,
            string tag = null )
```

Fill a GraphicsPath.

Parameters

path	The GraphicsPath to fill.
fillColour	The Brush with which to fill the GraphicsPath.
tag	A tag to identify the filled path.

Definition at line 276 of file Graphics.cs.

7.40.2.16 FillRectangle() [1/2]

```
void VectSharp.Graphics.FillRectangle (
            double leftX,
            double topY,
            double width,
            double height,
            Brush fillColour,
            string tag = null )
```

Fill a rectangle.

Parameters

leftX	The horizontal coordinate of the top-left corner of the rectangle.
topY	The vertical coordinate of the top-left corner of the rectangle.
width	The width of the rectangle.
height	The height of the rectangle.
fillColour	The colour with which to fill the rectangle.
tag	A tag to identify the filled rectangle.

Definition at line 435 of file Graphics.cs.

7.40.2.17 FillRectangle() [2/2]

```
void VectSharp.Graphics.FillRectangle (
            Point topLeft,
            Size size,
            Brush fillColour,
            string tag = null )
```

Fill a rectangle.

Parameters

topLeft	The top-left corner of the rectangle.
size	The size of the rectangle.
fillColour	The colour with which to fill the rectangle.
tag	A tag to identify the filled rectangle.

Definition at line 421 of file Graphics.cs.

7.40.2.18 FillText() [1/4]

```
void VectSharp.Graphics.FillText (
            double originX,
            double originY,
            IEnumerable< FormattedText > text,
            Brush fillColour,
            TextBaselines textBaseline = TextBaselines.Top,
            string tag = null )
```

Fill a formatted text string.

Parameters

originX	The horizontal coordinate of the text origin.
originY	The vertical coordinate of the text origin. See *textBaseline* .
text	The FormattedText to draw.
fillColour	The default Brush to use to fill the text. This can be overridden by each *text* element.
textBaseline	The text baseline (determines what *originY* represents).
tag	A tag to identify the filled text.

Definition at line 485 of file Graphics.Text.cs.

7.40.2.19 FillText() [2/4]

```
void VectSharp.Graphics.FillText (
            double originX,
            double originY,
            string text,
            Font font,
            Brush fillColour,
            TextBaselines textBaseline = TextBaselines.Top,
            string tag = null )
```

Fill a text string.

Parameters

originX	The horizontal coordinate of the text origin.
originY	The vertical coordinate of the text origin. See *textBaseline* .
text	The string to draw.
font	The font with which to draw the text.
fillColour	The Brush to use to fill the text.
textBaseline	The text baseline (determines what *originY* represents).
tag	A tag to identify the filled text.

Definition at line 57 of file Graphics.Text.cs.

7.40.2.20 FillText() [3/4]

```
void VectSharp.Graphics.FillText (
            Point origin,
            IEnumerable< FormattedText > text,
            Brush fillColour,
            TextBaselines textBaseline = TextBaselines.Top,
            string tag = null )
```

Fill a formatted text string.

Parameters

origin	The text origin. See *textBaseline* .
text	The FormattedText to draw.
fillColour	The default Brush to use to fill the text. This can be overridden by each *text* element.
textBaseline	The text baseline (determines what the vertical component of *origin* represents).
tag	A tag to identify the filled text.

Definition at line 392 of file Graphics.Text.cs.

7.40.2.21 FillText() [4/4]

```
void VectSharp.Graphics.FillText (
            Point origin,
            string text,
            Font font,
            Brush fillColour,
            TextBaselines textBaseline = TextBaselines.Top,
            string tag = null )
```

Fill a text string.

Parameters

origin	The text origin. See *textBaseline* .
text	The string to draw.
font	The font with which to draw the text.
fillColour	The Brush to use to fill the text.
textBaseline	The text baseline (determines what the vertical component of *origin* represents).
tag	A tag to identify the filled text.

Definition at line 37 of file Graphics.Text.cs.

7.40.2.22 FillTextOnPath()

```
void VectSharp.Graphics.FillTextOnPath (
            GraphicsPath path,
            string text,
            Font font,
            Brush fillColour,
            double reference = 0,
            TextAnchors anchor = TextAnchors.Left,
            TextBaselines textBaseline = TextBaselines.Top,
            string tag = null )
```

Fill a text string along a GraphicsPath.

Parameters

path	The GraphicsPath along which the text will flow.
text	The string to draw.
font	The font with which to draw the text.
fillColour	The Brush to use to fill the text.
reference	The (relative) starting point on the path starting from which the text should be drawn (0 is the start of the path, 1 is the end of the path).
anchor	The anchor in the text string that will correspond to the point specified by the *reference* .
textBaseline	The text baseline (determines which the position of the text in relation to the *path* .
tag	A tag to identify the filled text.

Definition at line 125 of file Graphics.Text.cs.

7.40.2.23 FillTextUnderline() [1/4]

```
void VectSharp.Graphics.FillTextUnderline (
            double originX,
            double originY,
            IEnumerable< FormattedText > text,
            Brush fillColour,
            TextBaselines textBaseline = TextBaselines.Top,
            string tag = null )
```

Fill the underline of the specified formatted text string.

Parameters

originX	The horizontal coordinate of the text origin.
originY	The vertical coordinate of the text origin. See *textBaseline* .
text	The FormattedText whose underline will be drawn.
fillColour	The default Brush to use to fill the underline. This can be overridden by each *text* element.
textBaseline	The text baseline (determines what *originY* represents).
tag	A tag to identify the filled underlined.

Definition at line 707 of file Graphics.Text.cs.

7.40.2.24 FillTextUnderline() [2/4]

```
void VectSharp.Graphics.FillTextUnderline (
            double originX,
            double originY,
            string text,
            Font font,
            Brush fillColour,
            TextBaselines textBaseline = TextBaselines.Top,
            string tag = null )
```

Fills the underline of the specified text string.

Parameters

originX	The horizontal coordinate of the text origin.
originY	The vertical coordinate of the text origin. See *textBaseline* .
text	The string whose underline will be draw.
font	The font with which to draw the text.
fillColour	The Brush to use to fill the underline.
textBaseline	The text baseline (determines what *originY* represents).
tag	A tag to identify the filled underline.

Definition at line 639 of file Graphics.Text.cs.

7.40.2.25 FillTextUnderline() [3/4]

```
void VectSharp.Graphics.FillTextUnderline (
            Point origin,
            IEnumerable< FormattedText > text,
            Brush fillColour,
            TextBaselines textBaseline = TextBaselines.Top,
            string tag = null )
```

Fill the underline of the specified formatted text string.

Parameters

origin	The text origin. See *textBaseline* .
text	The FormattedText whose underline will be drawn.
fillColour	The default Brush to use to fill the underline. This can be overridden by each *text* element.
textBaseline	The text baseline (determines what the vertical component of *origin* represents).
tag	A tag to identify the filled underlined.

Definition at line 720 of file Graphics.Text.cs.

7.40.2.26 FillTextUnderline() [4/4]

```
void VectSharp.Graphics.FillTextUnderline (
            Point origin,
            string text,
            Font font,
            Brush fillColour,
            TextBaselines textBaseline = TextBaselines.Top,
            string tag = null )
```

Fills the underline of the specified text string.

Parameters

origin	The text origin. See *textBaseline* .
text	The string whose underline will be draw.
font	The font with which to draw the text.
fillColour	The Brush to use to fill the underline.
textBaseline	The text baseline (determines what the vertical component of *origin* represents).
tag	A tag to identify the filled underline.

Definition at line 653 of file Graphics.Text.cs.

7.40.2.27 GetBounds()

```
Rectangle VectSharp.Graphics.GetBounds ( )
```

Computes the rectangular bounds of the region affected by the drawing operations performed on the Graphics object.

Returns

The smallest rectangle that contains all the elements drawn on the Graphics.

Definition at line 1403 of file Graphics.cs.

7.40.2.28 GetTags()

```
IEnumerable< string > VectSharp.Graphics.GetTags ( )
```

Gets all the tags that have been defined in the Graphics.

Returns

An IEnumerable<T> of strings that, when enumerated, returns all the tags that have been defined in the Graphics.

Definition at line 1655 of file Graphics.cs.

7.40.2.29 Linearise()

```
Graphics VectSharp.Graphics.Linearise (
            double resolution )
```

Creates a new Graphics object by linearising all of the elements of the current instance, i.e. replacing curve segments with series of line segments that approximate them. Raster images are left unchanged.

Parameters

resolution	The resolution that will be used to linearise curve segments.

Returns

A new Graphics object containing the linearised elements.

Definition at line 1347 of file Graphics.cs.

7.40.2.30 MeasureText() [1/2]

```
Size VectSharp.Graphics.MeasureText (
            IEnumerable< FormattedText > text )
```

Measure a formatted text string. See also

See also

FormattedTextExtensions.Measure(IEnumerable<FormattedText>)

.

Parameters

text	The collection of FormattedText objects to measure.

Returns

The size of the measured *text* .

Definition at line 622 of file Graphics.Text.cs.

7.40.2.31 MeasureText() [2/2]

```
Size VectSharp.Graphics.MeasureText (
            string text,
            Font font )
```

Measure a text string. See also

See also

Font.MeasureText(string), Font.MeasureTextAdvanced(string)

and .

Parameters

text	The string to measure.
font	The font to use to measure the string.

Returns

The size of the measured *text* .

Definition at line 611 of file Graphics.Text.cs.

7.40.2.32 Restore()

```
void VectSharp.Graphics.Restore ( )
```

Restore the previous transform state (rotation, translation scale).

Definition at line 552 of file Graphics.cs.

7.40.2.33 Rotate()

```
void VectSharp.Graphics.Rotate (
            double angle,
            string tag = null )
```

Rotate the coordinate system around the origin.

Parameters

angle	The angle (in radians) by which to rotate the coordinate system.
tag	A tag to identify the transform.

Definition at line 336 of file Graphics.cs.

7.40.2.34 RotateAt()

```
void VectSharp.Graphics.RotateAt (
            double angle,
```

```
            Point pivot,
            string tag = null )
```

Rotate the coordinate system around a pivot point.

Parameters

angle	The angle (in radians) by which to rotate the coordinate system.
pivot	The pivot around which the coordinate system is to be rotated.
tag	A tag to identify the transform.

Definition at line 347 of file Graphics.cs.

7.40.2.35 Save()

```
void VectSharp.Graphics.Save ( )
```

Save the current transform state (rotation, translation, scale).

Definition at line 544 of file Graphics.cs.

7.40.2.36 Scale()

```
void VectSharp.Graphics.Scale (
            double scaleX,
            double scaleY,
            string tag = null )
```

Scale the coordinate system with respect to the origin.

Parameters

scaleX	The horizontal scale.
scaleY	The vertical scale.
tag	A tag to identify the transform.

Definition at line 409 of file Graphics.cs.

7.40.2.37 SetClippingPath() [1/3]

```
void VectSharp.Graphics.SetClippingPath (
            double leftX,
            double topY,
```

```
            double width,
            double height,
            string tag = null )
```

Intersect the current clipping path with the specified rectangle.

Parameters

leftX	The horizontal coordinate of the top-left corner of the rectangle.
topY	The vertical coordinate of the top-left corner of the rectangle.
width	The width of the rectangle.
height	The height of the rectangle.
tag	A tag to identify the clipping path.

Definition at line 315 of file Graphics.cs.

7.40.2.38 SetClippingPath() [2/3]

```
void VectSharp.Graphics.SetClippingPath (
            GraphicsPath path,
            string tag = null )
```

Intersect the current clipping path with the specified GraphicsPath.

Parameters

path	The GraphicsPath to intersect with the current clipping path.
tag	A tag to identify the clipping path.

Definition at line 302 of file Graphics.cs.

7.40.2.39 SetClippingPath() [3/3]

```
void VectSharp.Graphics.SetClippingPath (
            Point topLeft,
            Size size,
            string tag = null )
```

Intersect the current clipping path with the specified rectangle.

Parameters

topLeft	The top-left corner of the rectangle.
size	The size of the rectangle.
tag	A tag to identify the clipping path.

Definition at line 326 of file Graphics.cs.

7.40.2.40 StrokePath()

```
void VectSharp.Graphics.StrokePath (
            GraphicsPath path,
            Brush strokeColour,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke a GraphicsPath.

Parameters

path	The GraphicsPath to stroke.
strokeColour	The Brush with which to stroke the GraphicsPath.
lineWidth	The width of the line with which the path is stroked.
lineCap	The line cap to use to stroke the path.
lineJoin	The line join to use to stroke the path.
lineDash	The line dash to use to stroke the path.
tag	A tag to identify the stroked path.

Definition at line 292 of file Graphics.cs.

7.40.2.41 StrokeRectangle() [1/2]

```
void VectSharp.Graphics.StrokeRectangle (
            double leftX,
            double topY,
            double width,
            double height,
            Brush strokeColour,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke a rectangle.

Parameters

leftX	The horizontal coordinate of the top-left corner of the rectangle.
topY	The vertical coordinate of the top-left corner of the rectangle.
width	The width of the rectangle.

Parameters

height	The height of the rectangle.
strokeColour	The colour with which to stroke the rectangle.
lineWidth	The width of the line with which the rectangle is stroked.
lineCap	The line cap to use to stroke the rectangle.
lineJoin	The line join to use to stroke the rectangle.
lineDash	The line dash to use to stroke the rectangle.
tag	A tag to identify the filled rectangle.

Definition at line 469 of file Graphics.cs.

7.40.2.42 StrokeRectangle() [2/2]

```
void VectSharp.Graphics.StrokeRectangle (
            Point topLeft,
            Size size,
            Brush strokeColour,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke a rectangle.

Parameters

topLeft	The top-left corner of the rectangle.
size	The size of the rectangle.
strokeColour	The colour with which to stroke the rectangle.
lineWidth	The width of the line with which the rectangle is stroked.
lineCap	The line cap to use to stroke the rectangle.
lineJoin	The line join to use to stroke the rectangle.
lineDash	The line dash to use to stroke the rectangle.
tag	A tag to identify the filled rectangle.

Definition at line 451 of file Graphics.cs.

7.40.2.43 StrokeText() [1/4]

```
void VectSharp.Graphics.StrokeText (
            double originX,
            double originY,
            IEnumerable< FormattedText > text,
```

```
            Brush strokeColour,
            TextBaselines textBaseline = TextBaselines.Top,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke a formatted text string.

Parameters

originX	The horizontal coordinate of the text origin.
originY	The vertical coordinate of the text origin. See *textBaseline* .
text	The FormattedText to draw.
strokeColour	The default Brush with which to stroke the text.
lineWidth	The width of the line with which the text is stroked.
lineCap	The line cap to use to stroke the text.
lineJoin	The line join to use to stroke the text.
lineDash	The line dash to use to stroke the text.
textBaseline	The text baseline (determines what *originY* represents).
tag	A tag to identify the stroked text.

Definition at line 599 of file Graphics.Text.cs.

7.40.2.44 StrokeText() [2/4]

```
void VectSharp.Graphics.StrokeText (
            double originX,
            double originY,
            string text,
            Font font,
            Brush strokeColour,
            TextBaselines textBaseline = TextBaselines.Top,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke a text string.

Parameters

originX	The horizontal coordinate of the text origin.
originY	The vertical coordinate of the text origin. See *textBaseline* .
text	The string to draw.
font	The font with which to draw the text.
strokeColour	The Brush with which to stroke the text.
lineWidth	The width of the line with which the text is stroked.
lineCap	The line cap to use to stroke the text.

Parameters

lineJoin	The line join to use to stroke the text.
lineDash	The line dash to use to stroke the text.
textBaseline	The text baseline (determines what *originY* represents).
tag	A tag to identify the stroked text.

Definition at line 104 of file Graphics.Text.cs.

7.40.2.45 StrokeText() [3/4]

```
void VectSharp.Graphics.StrokeText (
            Point origin,
            IEnumerable< FormattedText > text,
            Brush strokeColour,
            TextBaselines textBaseline = TextBaselines.Top,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke a formatted text string.

Parameters

origin	The text origin. See *textBaseline* .
text	The FormattedText to draw.
strokeColour	The default Brush with which to stroke the text.
lineWidth	The width of the line with which the text is stroked.
lineCap	The line cap to use to stroke the text.
lineJoin	The line join to use to stroke the text.
lineDash	The line dash to use to stroke the text.
textBaseline	The text baseline (determines what the vertical component of *origin* represents).
tag	A tag to identify the stroked text.

Definition at line 502 of file Graphics.Text.cs.

7.40.2.46 StrokeText() [4/4]

```
void VectSharp.Graphics.StrokeText (
            Point origin,
            string text,
            Font font,
            Brush strokeColour,
            TextBaselines textBaseline = TextBaselines.Top,
```

```
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke a text string.

Parameters

origin	The text origin. See *textBaseline* .
text	The string to draw.
font	The font with which to draw the text.
strokeColour	The Brush with which to stroke the text.
lineWidth	The width of the line with which the text is stroked.
lineCap	The line cap to use to stroke the text.
lineJoin	The line join to use to stroke the text.
lineDash	The line dash to use to stroke the text.
textBaseline	The text baseline (determines what the vertical component of *origin* represents).
tag	A tag to identify the stroked text.

Definition at line 80 of file Graphics.Text.cs.

7.40.2.47 StrokeTextOnPath()

```
void VectSharp.Graphics.StrokeTextOnPath (
            GraphicsPath path,
            string text,
            Font font,
            Brush strokeColour,
            double reference = 0,
            TextAnchors anchor = TextAnchors.Left,
            TextBaselines textBaseline = TextBaselines.Top,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke a text string along a GraphicsPath.

Parameters

path	The GraphicsPath along which the text will flow.
text	The string to draw.
font	The font with which to draw the text.
strokeColour	The Brush with which to stroke the text.
lineWidth	The width of the line with which the text is stroked.
lineCap	The line cap to use to stroke the text.
lineJoin	The line join to use to stroke the text.
lineDash	The line dash to use to stroke the text.

Parameters

reference	The (relative) starting point on the path starting from which the text should be drawn (0 is the start of the path, 1 is the end of the path).
anchor	The anchor in the text string that will correspond to the point specified by the *reference* .
textBaseline	The text baseline (determines which the position of the text in relation to the *path* .
tag	A tag to identify the stroked text.

Definition at line 262 of file Graphics.Text.cs.

7.40.2.48 StrokeTextUnderline() [1/4]

```
void VectSharp.Graphics.StrokeTextUnderline (
            double originX,
            double originY,
            IEnumerable< FormattedText > text,
            Brush strokeColour,
            TextBaselines textBaseline = TextBaselines.Top,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke the underline of the specified formatted text string.

Parameters

originX	The horizontal coordinate of the text origin.
originY	The vertical coordinate of the text origin. See *textBaseline* .
text	The FormattedText to draw.
strokeColour	The default Brush with which to stroke the underline.
lineWidth	The width of the line with which the underline is stroked.
lineCap	The line cap to use to stroke the underline.
lineJoin	The line join to use to stroke the underline.
lineDash	The line dash to use to stroke the underline.
textBaseline	The text baseline (determines what *originY* represents).
tag	A tag to identify the stroked underline.

Definition at line 817 of file Graphics.Text.cs.

7.40.2.49 StrokeTextUnderline() [2/4]

```
void VectSharp.Graphics.StrokeTextUnderline (
            double originX,
```

```
            double originY,
            string text,
            Font font,
            Brush strokeColour,
            TextBaselines textBaseline = TextBaselines.Top,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke the underline of the specified text string.

Parameters

originX	The horizontal coordinate of the text origin.
originY	The vertical coordinate of the text origin. See *textBaseline* .
text	The string whose underline will be drawn.
font	The font with which to draw the text.
strokeColour	The Brush with which to stroke the underline.
lineWidth	The width of the line with which the underline is stroked.
lineCap	The line cap to use to stroke the underline.
lineJoin	The line join to use to stroke the underline.
lineDash	The line dash to use to stroke the underline.
textBaseline	The text baseline (determines what *originY* represents).
tag	A tag to identify the stroked underline.

Definition at line 673 of file Graphics.Text.cs.

7.40.2.50 StrokeTextUnderline() [3/4]

```
void VectSharp.Graphics.StrokeTextUnderline (
            Point origin,
            IEnumerable< FormattedText > text,
            Brush strokeColour,
            TextBaselines textBaseline = TextBaselines.Top,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke the underline of the specified formatted text string.

Parameters

origin	The text origin. See *textBaseline* .
text	The FormattedText to draw.
strokeColour	The default Brush with which to stroke the underline.
lineWidth	The width of the line with which the underline is stroked.
lineCap	The line cap to use to stroke the underline.

Parameters

lineJoin	The line join to use to stroke the underline.
lineDash	The line dash to use to stroke the underline.
textBaseline	The text baseline (determines what the vertical component of *origin* represents).
tag	A tag to identify the stroked underline.

Definition at line 834 of file Graphics.Text.cs.

7.40.2.51 StrokeTextUnderline() [4/4]

```
void VectSharp.Graphics.StrokeTextUnderline (
            Point origin,
            string text,
            Font font,
            Brush strokeColour,
            TextBaselines textBaseline = TextBaselines.Top,
            double lineWidth = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineJoins lineJoin = LineJoins.Miter,
            LineDash?  lineDash = null,
            string tag = null )
```

Stroke the underline of the specified text string.

Parameters

origin	The text origin. See *textBaseline* .
text	The string whose underline will be drawn.
font	The font with which to draw the text.
strokeColour	The Brush with which to stroke the underline.
lineWidth	The width of the line with which the underline is stroked.
lineCap	The line cap to use to stroke the underline.
lineJoin	The line join to use to stroke the underline.
lineDash	The line dash to use to stroke the underline.
textBaseline	The text baseline (determines what the vertical component of *origin* represents).
tag	A tag to identify the stroked underline.

Definition at line 691 of file Graphics.Text.cs.

7.40.2.52 Transform() [1/3]

```
void VectSharp.Graphics.Transform (
            double a,
            double b,
            double c,
```

```
            double d,
            double e,
            double f,
            string tag = null )
```

Transform the coordinate system with the specified transformation matrix [[a, c, e], [b, d, f], [0, 0, 1]].

Parameters

a	The first element of the first column.
b	The second element of the first column.
c	The first element of the second column.
d	The second element of the second column.
e	The first element of the third column.
f	The second element of the third column.
tag	A tag to identify the transform.

Definition at line 376 of file Graphics.cs.

7.40.2.53 Transform() [2/3]

```
Graphics VectSharp.Graphics.Transform (
            Func< Point, Point > transformationFunction,
            double linearisationResolution )
```

Creates a new Graphics object in which all the graphics actions have been transformed using an arbitrary transformation function. Raster images are replaced by grey rectangles.

Parameters

transformationFunction	An arbitrary transformation function.
linearisationResolution	The resolution that will be used to linearise curve segments.

Returns

A new Graphics object in which all graphics actions have been linearised and transformed using the *transformationFunction* .

Definition at line 1020 of file Graphics.cs.

7.40.2.54 Transform() [3/3]

```
Graphics VectSharp.Graphics.Transform (
            Func< Point, Point > transformationFunction,
            double linearisationResolution,
            double maxSegmentLength )
```

Creates a new Graphics object in which all the graphics actions have been transformed using an arbitrary transformation function. Raster images are replaced by grey rectangles.

Parameters

transformationFunction	An arbitrary transformation function.
linearisationResolution	The resolution that will be used to linearise curve segments.
maxSegmentLength	The maximum length of line segments.

Returns

A new Graphics object in which all graphics actions have been linearised and transformed using the *transformationFunction* .

Definition at line 1218 of file Graphics.cs.

7.40.2.55 Translate() [1/2]

```
void VectSharp.Graphics.Translate (
            double x,
            double y,
            string tag = null )
```

Translate the coordinate system origin.

Parameters

x	The horizontal translation.
y	The vertical translation.
tag	A tag to identify the transform.

Definition at line 388 of file Graphics.cs.

7.40.2.56 Translate() [2/2]

```
void VectSharp.Graphics.Translate (
            Point delta,
            string tag = null )
```

Translate the coordinate system origin.

Parameters

delta	The new origin point.
tag	A tag to identify the transform.

Definition at line 398 of file Graphics.cs.

7.40.2.57 TryRasterise()

```
bool VectSharp.Graphics.TryRasterise (
            Rectangle region,
            double scale,
            bool interpolate,
            out RasterImage output )
```

Tries to rasterise specified region of this Graphics object using the default rasterisation method.

Parameters

region	The region of the Graphics to rasterise.
scale	The scale at which the image is rasterised.
interpolate	Determines whether the resulting RasterImage should be interpolated or not.
output	When this method returns, this will contain the rasterised image (or `null` if the image could not be rasterised.

Returns

`true` if the image could be rasterised; `false` if it could not be rasterised.

Definition at line 1511 of file Graphics.cs.

7.40.3 Member Data Documentation

7.40.3.1 RasterisationMethod

```
Func<Graphics, Rectangle, double, bool, RasterImage> VectSharp.Graphics.RasterisationMethod =
null [static]
```

A method that is used to rasterise a region of a Graphics object. Set this to `null` if you wish to use the default rasterisation methods (implemented by either VectSharp.Raster, or VectSharp.Raster.ImageSharp). You will have to provide your own implementation of this method if neither VectSharp.Raster nor VectSharp.Raster.ImageSharp are referenced by your project. The first argument of this method is the Graphics to be rasterised, the second is a Rectangle representing the region to rasterise, the third is a double representing the scale, and the third is a boolean value indicating whether the resulting RasterImage should be interpolated.

Definition at line 1501 of file Graphics.cs.

7.40.4 Property Documentation

7.40.4.1 UnbalancedStackAction

```
UnbalancedStackActions VectSharp.Graphics.UnbalancedStackAction = UnbalancedStackActions.Throw
[static], [get], [set]
```

Determines how an unbalanced graphics state stack (which occurs if the number of calls to Save and Restore is not equal) will be treated. The default is UnbalancedStackActions.Throw.

Definition at line 266 of file Graphics.cs.

The documentation for this class was generated from the following files:

- VectSharp/Graphics.cs
- VectSharp/Graphics.Text.cs

7.41 VectSharp.GraphicsPath Class Reference

Represents a graphics path that can be filled or stroked.

Public Member Functions

- GraphicsPath MoveTo (Point p)

  Move the current point without tracing a segment from the previous point.
- GraphicsPath MoveTo (double x, double y)

  Move the current point without tracing a segment from the previous point.
- GraphicsPath LineTo (Point p)

  Move the current point and trace a segment from the previous point.
- GraphicsPath LineTo (double x, double y)

  Move the current point and trace a segment from the previous point.
- GraphicsPath Arc (Point center, double radius, double startAngle, double endAngle)

  Trace an arc segment from a circle with the specified center and radius , starting at startAngle and ending at endAngle . The current point is updated to the end point of the arc.
- GraphicsPath Arc (double centerX, double centerY, double radius, double startAngle, double endAngle)

  Trace an arc segment from a circle with the specified center and radius , starting at startAngle and ending at endAngle . The current point is updated to the end point of the arc.
- GraphicsPath EllipticalArc (double radiusX, double radiusY, double axisAngle, bool largeArc, bool sweep↩Clockwise, Point endPoint)

  Trace an arc from an ellipse with the specified radii, rotated by axisAngle with respect to the x-axis, starting at the current point and ending at the endPoint .
- GraphicsPath CubicBezierTo (Point control1, Point control2, Point endPoint)

  Trace a cubic Bezier curve from the current point to a destination point, with two control points. The current point is updated to the end point of the Bezier curve.
- GraphicsPath CubicBezierTo (double control1X, double control1Y, double control2X, double control2Y, double endPointX, double endPointY)

  Trace a cubic Bezier curve from the current point to a destination point, with two control points. The current point is updated to the end point of the Bezier curve.
- GraphicsPath QuadraticBezierTo (Point control, Point endPoint)

  Trace a quadratic Bezier curve from the current point to a destination point, with a single control point. The current point is updated to the end point of the Bezier curve.
- GraphicsPath QuadraticBezierTo (double controlX, double controlY, double endPointX, double endPointY)

Trace a quadratic Bezier curve from the current point to a destination point, with a single control point. The current point is updated to the end point of the Bezier curve.

- GraphicsPath Close ()

  Trace a segment from the current point to the start point of the figure and flag the figure as closed.

- GraphicsPath AddText (double originX, double originY, string text, Font font, TextBaselines text↩ Baseline=TextBaselines.Top)

  Add the contour of a text string to the current path.

- GraphicsPath AddText (Point origin, string text, Font font, TextBaselines textBaseline=TextBaselines.Top)

  Add the contour of a text string to the current path.

- GraphicsPath AddTextOnPath (GraphicsPath path, string text, Font font, double reference=0, TextAnchors anchor=TextAnchors.Left, TextBaselines textBaseline=TextBaselines.Top)

  Add the contour of a text string flowing along a GraphicsPath to the current path.

- GraphicsPath AddTextUnderline (Point origin, string text, Font font, TextBaselines textBaseline=Text↩ Baselines.Top)

  Add the contour of the underline of the specified text string to the current path.

- GraphicsPath AddSmoothSpline (params Point[ ] points)

  Adds a smooth spline composed of cubic bezier segments that pass through the specified points.

- GraphicsPath AddPath (GraphicsPath path)

  Adds another GraphicsPath to the current GraphicsPath.

- double MeasureLength ()

  Measures the length of the GraphicsPath.

- Point GetPointAtRelative (double position)

  Gets the point at the relative position specified on the GraphicsPath.

- Point GetPointAtAbsolute (double length)

  Gets the point at the absolute position specified on the GraphicsPath.

- Point GetTangentAtRelative (double position)

  Gets the tangent to the point at the relative position specified on the GraphicsPath.

- Point GetTangentAtAbsolute (double length)

  Gets the tangent to the point at the absolute position specified on the GraphicsPath.

- Point GetNormalAtAbsolute (double length)

  Gets the normal to the point at the absolute position specified on the GraphicsPath.

- Point GetNormalAtRelative (double position)

  Gets the normal to the point at the relative position specified on the GraphicsPath.

- GraphicsPath Linearise (double resolution)

  Linearises a GraphicsPath, replacing curve segments with series of line segments that approximate them.

- GraphicsPath Discretise (double resolution)

  Discretises a GraphicsPath, replacing curve segments with series of line segments that approximate them and ensuring that all line segments are shorter than the specified resolution .

- IEnumerable< List< Point > > GetPoints ()

  Gets a collection of the end points of all the segments in the GraphicsPath, divided by figure.

- IEnumerable< GraphicsPath > GetFigures ()

  Gets a collection of all the figures in the GraphicsPath, returned as individual GraphicsPaths.

- IEnumerable< List< Point > > GetLinearisationPointsNormals (double resolution)

  Gets a collection of the tangents at the end point of the segments in which the GraphicsPath would be linearised, divided by figure.

- IEnumerable< GraphicsPath > Triangulate (double resolution, bool clockwise)

  Divides a GraphicsPath into triangles.

- GraphicsPath Transform (Func< Point, Point > transformationFunction)

  Transforms all of the Points in the GraphicsPath with an arbitrary transformation function.

- Rectangle GetBounds ()

  Compute the rectangular bounds of the path.

Properties

- List< Segment > Segments = new List() `[get, set]`

  The segments that make up the path.

7.41.1 Detailed Description

Represents a graphics path that can be filled or stroked.

Definition at line 28 of file GraphicsPath.cs.

7.41.2 Member Function Documentation

7.41.2.1 AddPath()

```
GraphicsPath VectSharp.GraphicsPath.AddPath (
            GraphicsPath path )
```

Adds another GraphicsPath to the current GraphicsPath.

Parameters

path	The existing GraphicsPath that should be added to the current GraphicsPath.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 1201 of file GraphicsPath.cs.

7.41.2.2 AddSmoothSpline()

```
GraphicsPath VectSharp.GraphicsPath.AddSmoothSpline (
            params Point[] points )
```

Adds a smooth spline composed of cubic bezier segments that pass through the specified points.

Parameters

points	The points through which the spline should pass.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 1169 of file GraphicsPath.cs.

7.41.2.3 AddText() [1/2]

```
GraphicsPath VectSharp.GraphicsPath.AddText (
            double originX,
            double originY,
            string text,
            Font font,
            TextBaselines textBaseline = TextBaselines.Top )
```

Add the contour of a text string to the current path.

Parameters

originX	The horizontal coordinate of the text origin.
originY	The vertical coordinate of the text origin. See *textBaseline* .
text	The string to draw.
font	The font with which to draw the text.
textBaseline	The text baseline (determines what *originY* represents).

///

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 358 of file GraphicsPath.cs.

7.41.2.4 AddText() [2/2]

```
GraphicsPath VectSharp.GraphicsPath.AddText (
            Point origin,
            string text,
            Font font,
            TextBaselines textBaseline = TextBaselines.Top )
```

Add the contour of a text string to the current path.

Parameters

origin	The text origin. See *textBaseline* .
text	The string to draw.
font	The font with which to draw the text.
textBaseline	The text baseline (determines what the vertical component of *origin* represents).

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 371 of file GraphicsPath.cs.

7.41.2.5 AddTextOnPath()

```
GraphicsPath VectSharp.GraphicsPath.AddTextOnPath (
            GraphicsPath path,
            string text,
            Font font,
            double reference = 0,
            TextAnchors anchor = TextAnchors.Left,
            TextBaselines textBaseline = TextBaselines.Top )
```

Add the contour of a text string flowing along a GraphicsPath to the current path.

Parameters

path	The GraphicsPath along which the text will flow.
text	The string to draw.
font	The font with which to draw the text.
reference	The (relative) starting point on the path starting from which the text should be drawn (0 is the start of the path, 1 is the end of the path).
anchor	The anchor in the text string that will correspond to the point specified by the *reference* .
textBaseline	The text baseline (determines which the position of the text in relation to the *path* .

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 473 of file GraphicsPath.cs.

7.41.2.6 AddTextUnderline()

```
GraphicsPath VectSharp.GraphicsPath.AddTextUnderline (
            Point origin,
            string text,
            Font font,
            TextBaselines textBaseline = TextBaselines.Top )
```

Add the contour of the underline of the specified text string to the current path.

Parameters

origin	The text origin. See *textBaseline* .
text	The string whose underline will be drawn.
font	The font with which to draw the text.
textBaseline	The text baseline (determines what the vertical component of *origin* represents).

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 618 of file GraphicsPath.cs.

7.41.2.7 Arc() [1/2]

```
GraphicsPath VectSharp.GraphicsPath.Arc (
            double centerX,
            double centerY,
            double radius,
            double startAngle,
            double endAngle )
```

Trace an arc segment from a circle with the specified center and *radius* , starting at *startAngle* and ending at *endAngle* . The current point is updated to the end point of the arc.

Parameters

centerX	The horizontal coordinate of the center of the arc.
centerY	The vertical coordinate of the center of the arc.
radius	The radius of the arc.
startAngle	The start angle (in radians) of the arc.
endAngle	The end angle (in radians) of the arc.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 126 of file GraphicsPath.cs.

7.41.2.8 Arc() [2/2]

```
GraphicsPath VectSharp.GraphicsPath.Arc (
            Point center,
            double radius,
            double startAngle,
            double endAngle )
```

Trace an arc segment from a circle with the specified *center* and *radius* , starting at *startAngle* and ending at *endAngle* . The current point is updated to the end point of the arc.

Parameters

center	The center of the arc.
radius	The radius of the arc.
startAngle	The start angle (in radians) of the arc.
endAngle	The end angle (in radians) of the arc.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 103 of file GraphicsPath.cs.

7.41.2.9 Close()

```
GraphicsPath VectSharp.GraphicsPath.Close ( )
```

Trace a segment from the current point to the start point of the figure and flag the figure as closed.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 341 of file GraphicsPath.cs.

7.41.2.10 CubicBezierTo() [1/2]

```
GraphicsPath VectSharp.GraphicsPath.CubicBezierTo (
            double control1X,
            double control1Y,
            double control2X,
            double control2Y,
            double endPointX,
            double endPointY )
```

Trace a cubic Bezier curve from the current point to a destination point, with two control points. The current point is updated to the end point of the Bezier curve.

Parameters

control1X	The horizontal coordinate of the first control point.
control1Y	The vertical coordinate of the first control point.
control2X	The horizontal coordinate of the second control point.
control2Y	The vertical coordinate of the second control point.
endPointX	The horizontal coordinate of the destination point.
endPointY	The vertical coordinate of the destination point.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 285 of file GraphicsPath.cs.

7.41.2.11 CubicBezierTo() [2/2]

```
GraphicsPath VectSharp.GraphicsPath.CubicBezierTo (
            Point control1,
            Point control2,
            Point endPoint )
```

Trace a cubic Bezier curve from the current point to a destination point, with two control points. The current point is updated to the end point of the Bezier curve.

Parameters

control1	The first control point.
control2	The second control point.
endPoint	The destination point.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 261 of file GraphicsPath.cs.

7.41.2.12 Discretise()

```
GraphicsPath VectSharp.GraphicsPath.Discretise (
            double resolution )
```

Discretises a GraphicsPath, replacing curve segments with series of line segments that approximate them and ensuring that all line segments are shorter than the specified *resolution* .

Parameters

resolution	The maximum length (in absolute units) of line segments in the resulting GraphicsPath.

Returns

A GraphicsPath composed only of linear segments that are shorter than *resolution* and approximate the current GraphicsPath.

Definition at line 1955 of file GraphicsPath.cs.

7.41.2.13 EllipticalArc()

```
GraphicsPath VectSharp.GraphicsPath.EllipticalArc (
            double radiusX,
```

```
double radiusY,
double axisAngle,
bool largeArc,
bool sweepClockwise,
Point endPoint )
```

Trace an arc from an ellipse with the specified radii, rotated by *axisAngle* with respect to the x-axis, starting at the current point and ending at the *endPoint* .

Parameters

radiusX	The horizontal radius of the ellipse.
radiusY	The vertical radius of the ellipse.
axisAngle	The angle of the horizontal axis of the ellipse with respect to the horizontal axis.
largeArc	Determines whether the large or the small arc is drawn.
sweepClockwise	Determines whether the clockwise or anticlockwise arc is drawn.
endPoint	The end point of the arc.

Returns

Definition at line 142 of file GraphicsPath.cs.

7.41.2.14 GetBounds()

```
Rectangle VectSharp.GraphicsPath.GetBounds ( )
```

Compute the rectangular bounds of the path.

Returns

The smallest Rectangle that contains the path.

Definition at line 3081 of file GraphicsPath.cs.

7.41.2.15 GetFigures()

```
IEnumerable< GraphicsPath > VectSharp.GraphicsPath.GetFigures ( )
```

Gets a collection of all the figures in the GraphicsPath, returned as individual GraphicsPaths.

Returns

A collection of all the figures in the GraphicsPath, returned as individual GraphicsPaths.

Definition at line 2026 of file GraphicsPath.cs.

7.41.2.16 GetLinearisationPointsNormals()

```
IEnumerable< List< Point > > VectSharp.GraphicsPath.GetLinearisationPointsNormals (
            double resolution )
```

Gets a collection of the tangents at the end point of the segments in which the GraphicsPath would be linearised, divided by figure.

Parameters

resolution	The absolute length between successive samples in curve segments.

Returns

A collection of the tangents at the end point of the segments in which the GraphicsPath would be linearised, divided by figure.

Definition at line 2069 of file GraphicsPath.cs.

7.41.2.17 GetNormalAtAbsolute()

```
Point VectSharp.GraphicsPath.GetNormalAtAbsolute (
            double length )
```

Gets the normal to the point at the absolute position specified on the GraphicsPath.

Parameters

length	The distance to the point from the start of the GraphicsPath.

Returns

The normal to the point at the specified position.

Definition at line 1879 of file GraphicsPath.cs.

7.41.2.18 GetNormalAtRelative()

```
Point VectSharp.GraphicsPath.GetNormalAtRelative (
            double position )
```

Gets the normal to the point at the relative position specified on the GraphicsPath.

Parameters

position	The position on the GraphicsPath (0 is the start of the path, 1 is the end of the path).

Returns

The normal to the point at the specified position.

Definition at line 1890 of file GraphicsPath.cs.

7.41.2.19 GetPointAtAbsolute()

```
Point VectSharp.GraphicsPath.GetPointAtAbsolute (
            double length )
```

Gets the point at the absolute position specified on the GraphicsPath.

Parameters

length	The distance to the point from the start of the GraphicsPath.

Returns

The point at the specified position.

Definition at line 1295 of file GraphicsPath.cs.

7.41.2.20 GetPointAtRelative()

```
Point VectSharp.GraphicsPath.GetPointAtRelative (
            double position )
```

Gets the point at the relative position specified on the GraphicsPath.

Parameters

position	The position on the GraphicsPath (0 is the start of the path, 1 is the end of the path).

Returns

The point at the specified position.

Definition at line 1285 of file GraphicsPath.cs.

7.41.2.21 GetPoints()

```
IEnumerable< List< Point > > VectSharp.GraphicsPath.GetPoints ( )
```

Gets a collection of the end points of all the segments in the GraphicsPath, divided by figure.

Returns

A collection of the end points of all the segments in the GraphicsPath, divided by figure.

Definition at line 1983 of file GraphicsPath.cs.

7.41.2.22 GetTangentAtAbsolute()

```
Point VectSharp.GraphicsPath.GetTangentAtAbsolute (
            double length )
```

Gets the tangent to the point at the absolute position specified on the GraphicsPath.

Parameters

length	The distance to the point from the start of the GraphicsPath.

Returns

The tangent to the point at the specified position.

Definition at line 1592 of file GraphicsPath.cs.

7.41.2.23 GetTangentAtRelative()

```
Point VectSharp.GraphicsPath.GetTangentAtRelative (
            double position )
```

Gets the tangent to the point at the relative position specified on the GraphicsPath.

Parameters

position	The position on the GraphicsPath (0 is the start of the path, 1 is the end of the path).

Returns

The tangent to the point at the specified position.

Definition at line 1582 of file GraphicsPath.cs.

7.41.2.24 Linearise()

```
GraphicsPath VectSharp.GraphicsPath.Linearise (
            double resolution )
```

Linearises a GraphicsPath, replacing curve segments with series of line segments that approximate them.

Parameters

resolution	The absolute length between successive samples in curve segments.

Returns

A GraphicsPath composed only of linear segments that approximates the current GraphicsPath.

Definition at line 1901 of file GraphicsPath.cs.

7.41.2.25 LineTo() [1/2]

```
GraphicsPath VectSharp.GraphicsPath.LineTo (
            double x,
            double y )
```

Move the current point and trace a segment from the previous point.

Parameters

x	The horizontal coordinate of the new point.
y	The vertical coordinate of the new point.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 88 of file GraphicsPath.cs.

7.41.2.26 LineTo() [2/2]

```
GraphicsPath VectSharp.GraphicsPath.LineTo (
            Point p )
```

Move the current point and trace a segment from the previous point.

Parameters

p	The new point.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 66 of file GraphicsPath.cs.

7.41.2.27 MeasureLength()

```
double VectSharp.GraphicsPath.MeasureLength ( )
```

Measures the length of the GraphicsPath.

Returns

The length of the GraphicsPath

Definition at line 1213 of file GraphicsPath.cs.

7.41.2.28 MoveTo() [1/2]

```
GraphicsPath VectSharp.GraphicsPath.MoveTo (
            double x,
            double y )
```

Move the current point without tracing a segment from the previous point.

Parameters

x	The horizontal coordinate of the new point.
y	The vertical coordinate of the new point.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 55 of file GraphicsPath.cs.

7.41.2.29 MoveTo() [2/2]

```
GraphicsPath VectSharp.GraphicsPath.MoveTo (
            Point p )
```

Move the current point without tracing a segment from the previous point.

Parameters

p	The new point.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 41 of file GraphicsPath.cs.

7.41.2.30 QuadraticBezierTo() [1/2]

```
GraphicsPath VectSharp.GraphicsPath.QuadraticBezierTo (
            double controlX,
            double controlY,
            double endPointX,
            double endPointY )
```

Trace a quadratic Bezier curve from the current point to a destination point, with a single control point. The current point is updated to the end point of the Bezier curve.

Parameters

controlX	The horizontal coordinate of the control point.
controlY	The vertical coordinate of the control point.
endPointX	The horizontal coordinate of the destination point.
endPointY	The vertical coordinate of the destination point.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 331 of file GraphicsPath.cs.

7.41.2.31 QuadraticBezierTo() [2/2]

```
GraphicsPath VectSharp.GraphicsPath.QuadraticBezierTo (
            Point control,
            Point endPoint )
```

Trace a quadratic Bezier curve from the current point to a destination point, with a single control point. The current point is updated to the end point of the Bezier curve.

Parameters

control	The control point.
endPoint	The destination point.

Returns

The GraphicsPath, to allow for chained calls.

Definition at line 298 of file GraphicsPath.cs.

7.41.2.32 Transform()

```
GraphicsPath VectSharp.GraphicsPath.Transform (
            Func< Point, Point > transformationFunction )
```

Transforms all of the Points in the GraphicsPath with an arbitrary transformation function.

Parameters

transformationFunction	An arbitrary transformation function.

Returns

A new GraphicsPath in which all points have been replaced using the *transformationFunction* .

Definition at line 3062 of file GraphicsPath.cs.

7.41.2.33 Triangulate()

```
IEnumerable< GraphicsPath > VectSharp.GraphicsPath.Triangulate (
            double resolution,
            bool clockwise )
```

Divides a GraphicsPath into triangles.

Parameters

resolution	The resolution that will be used to linearise curve segments in the GraphicsPath.
clockwise	If this is `true`, the triangles will have their vertices in a clockwise order, otherwise they will be in anticlockwise order.

Returns

A collection of distinct GraphicsPaths, each representing one triangle.

Definition at line 2162 of file GraphicsPath.cs.

7.41.3 Property Documentation

7.41.3.1 Segments

```
List<Segment> VectSharp.GraphicsPath.Segments = new List<Segment>()  [get], [set]
```

The segments that make up the path.

Definition at line 33 of file GraphicsPath.cs.

The documentation for this class was generated from the following file:

- VectSharp/GraphicsPath.cs

7.42 VectSharp.Markdown.HTTPUtils Class Reference

Contains utilities to resolve absolute and relative URIs.

Static Public Member Functions

- static string bool wasDownloaded ResolveImageURI (string uri, string baseUriString)

Static Public Attributes

- static string path

  Resolves an image Uri, by downloading the image file if necessary. It also takes care of ensuring that the file extension matches the format of the file.

Properties

- static bool LogDownloads = true `[get, set]`

  Determines whether every file that is downloaded should be logged to the standard error stream.

7.42.1 Detailed Description

Contains utilities to resolve absolute and relative URIs.

Definition at line 244 of file HtmlTag.cs.

7.42.2 Member Function Documentation

7.42.2.1 ResolveImageURI()

```
static string bool wasDownloaded VectSharp.Markdown.HTTPUtils.ResolveImageURI (
            string uri,
            string baseUriString )  [static]
```

Definition at line 257 of file HtmlTag.cs.

7.42.3 Member Data Documentation

7.42.3.1 path

```
string VectSharp.Markdown.HTTPUtils.path  [static]
```

Resolves an image Uri, by downloading the image file if necessary. It also takes care of ensuring that the file extension matches the format of the file.

Parameters

uri	The address of the image.
baseUriString	The base uri to use for relative uris.

Returns

A tuple containing the local path of the image file (either the original image, or a local copy of a remote file) and a boolean value indicating whether the image was fetched from a remote location and should be deleted after the program is done with it.

Definition at line 257 of file HtmlTag.cs.

7.42.4 Property Documentation

7.42.4.1 LogDownloads

```
bool VectSharp.Markdown.HTTPUtils.LogDownloads = true  [static], [get], [set]
```

Determines whether every file that is downloaded should be logged to the standard error stream.

Definition at line 249 of file HtmlTag.cs.

The documentation for this class was generated from the following file:

- VectSharp.Markdown/HtmlTag.cs

7.43 VectSharp.IEasing Interface Reference

Describes a function used to transform the transition speed.

Inheritance diagram for VectSharp.IEasing:

Public Member Functions

- double Ease (double value)

  Applies the easing to the specified transition offset.

7.43.1 Detailed Description

Describes a function used to transform the transition speed.

Definition at line 1261 of file Animation.cs.

7.43.2 Member Function Documentation

7.43.2.1 Ease()

```
double VectSharp.IEasing.Ease (
            double value )
```

Applies the easing to the specified transition offset.

Parameters

value	The transition offset (ranging from 0 to 1).

Returns

The eased transition offset value.

Implemented in VectSharp.SplineEasing, and VectSharp.LinearEasing.

The documentation for this interface was generated from the following file:

- VectSharp/Animation.cs

7.44 VectSharp.Filters.IFilter Interface Reference

Represents a filter. Do not implement this interface directly; instead, implement ILocationInvariantFilter or IFilterWithLocation.

Inheritance diagram for VectSharp.Filters.IFilter:

Properties

- Point TopLeftMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.
- Point BottomRightMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

7.44.1 Detailed Description

Represents a filter. Do not implement this interface directly; instead, implement ILocationInvariantFilter or IFilterWithLocation.

Definition at line 25 of file Filters.cs.

7.44.2 Property Documentation

7.44.2.1 BottomRightMargin

```
Point VectSharp.Filters.IFilter.BottomRightMargin [get]
```

Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

Implemented in VectSharp.Filters.BoxBlurFilter, VectSharp.Filters.ColourMatrixFilter, VectSharp.Filters.CompositeLocationInvariantFil VectSharp.Filters.ConvolutionFilter, VectSharp.Filters.GaussianBlurFilter, and VectSharp.Filters.MaskFilter.

Definition at line 35 of file Filters.cs.

7.44.2.2 TopLeftMargin

```
Point VectSharp.Filters.IFilter.TopLeftMargin [get]
```

Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.

Implemented in VectSharp.Filters.BoxBlurFilter, VectSharp.Filters.ColourMatrixFilter, VectSharp.Filters.CompositeLocationInvariantFil VectSharp.Filters.ConvolutionFilter, VectSharp.Filters.GaussianBlurFilter, and VectSharp.Filters.MaskFilter.

Definition at line 30 of file Filters.cs.

The documentation for this interface was generated from the following file:

- VectSharp/Filters/Filters.cs

7.45 VectSharp.Filters.IFilterWithLocation Interface Reference

Represents a filter whose results depend on the position of the subject image on the graphics surface.

Inheritance diagram for VectSharp.Filters.IFilterWithLocation:

Public Member Functions

- RasterImage Filter (RasterImage image, Rectangle bounds, double scale)

  Applies the filter to a RasterImage.

Additional Inherited Members

7.45.1 Detailed Description

Represents a filter whose results depend on the position of the subject image on the graphics surface.

Definition at line 55 of file Filters.cs.

7.45.2 Member Function Documentation

7.45.2.1 Filter()

```
RasterImage VectSharp.Filters.IFilterWithLocation.Filter (
            RasterImage image,
            Rectangle bounds,
            double scale )
```

Applies the filter to a RasterImage.

Parameters

image	The RasterImage to which the filter will be applied.
bounds	The region on the graphics surface where the image will be drawn.
scale	The scale of the image with respect to the filter.

Returns

A new RasterImage containing the filtered image. The source *image* is left unaltered.

Implemented in VectSharp.Filters.MaskFilter.

The documentation for this interface was generated from the following file:

- VectSharp/Filters/Filters.cs

7.46 VectSharp.Filters.IFilterWithRasterisableParameter Interface Reference

Represents a filter with a parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter. The FilterWithRasterisableParameter abstract class provides a default implementation of this interface.

Inheritance diagram for VectSharp.Filters.IFilterWithRasterisableParameter:

Public Member Functions

- void RasteriseParameter (Func< Graphics, Rectangle, double, bool, RasterImage > rasterisationMethod, double scale)

  Rasterises the filter's parameter at the specified scale, using the specified rasterisation method.

7.46.1 Detailed Description

Represents a filter with a parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter. The FilterWithRasterisableParameter abstract class provides a default implementation of this interface.

Definition at line 71 of file Filters.cs.

7.46.2 Member Function Documentation

7.46.2.1 RasteriseParameter()

```
void VectSharp.Filters.IFilterWithRasterisableParameter.RasteriseParameter (
            Func< Graphics, Rectangle, double, bool, RasterImage > rasterisationMethod,
            double scale )
```

Rasterises the filter's parameter at the specified scale, using the specified rasterisation method.

Parameters

rasterisationMethod	The method used to rasterise the image. The first argument of this method is the Graphics to be rasterised, the second is a Rectangle representing the region to rasterise, the third is a double representing the scale, and the third is a boolean value indicating whether the resulting RasterImage should be interpolated.
scale	The scale factor at which the parameter is rasterised.

Implemented in VectSharp.Filters.FilterWithRasterisableParameter.

The documentation for this interface was generated from the following file:

- VectSharp/Filters/Filters.cs

7.47 VectSharp.IFontLibrary Interface Reference

Represents a font library with methods to create FontFamily objects from a string or from FontFamily.StandardFontFamilies.

Inheritance diagram for VectSharp.IFontLibrary:

Public Member Functions

- FontFamily ResolveFontFamily (string fontFamily)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised.
- FontFamily ResolveFontFamily (string fontFamily, params string[ ] fallback)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the fallback . If none of the fallback family names or true type files are valid, an exception might be raised.
- FontFamily ResolveFontFamily (FontFamily.StandardFontFamilies standardFontFamily)

  Create a new font family from the specified standard font family name.
- FontFamily ResolveFontFamily (string fontFamily, FontFamily.StandardFontFamilies finalFallback, params string[ ] fallback)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the fallback . If none of the fallback family names or true type files are valid, instantiate a standard font family using the finalFallback .

7.47.1 Detailed Description

Represents a font library with methods to create FontFamily objects from a string or from FontFamily.StandardFontFamilies.

Definition at line 29 of file FontLibrary.cs.

7.47.2 Member Function Documentation

7.47.2.1 ResolveFontFamily() [1/4]

```
FontFamily VectSharp.IFontLibrary.ResolveFontFamily (
            FontFamily.StandardFontFamilies standardFontFamily )
```

Create a new font family from the specified standard font family name.

Parameters

standardFontFamily	The standard name of the font family.

Returns

A FontFamily object corresponding to the specified font family.

Implemented in VectSharp.FontLibrary, VectSharp.SimpleFontLibrary, and VectSharp.DefaultFontLibrary.

7.47.2.2 ResolveFontFamily() [2/4]

```
FontFamily VectSharp.IFontLibrary.ResolveFontFamily (
            string fontFamily )
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised.

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.

Returns

If the font family name or the true type file is valid, a FontFamily object corresponding to the specified font family.

Implemented in VectSharp.FontLibrary, VectSharp.SimpleFontLibrary, and VectSharp.DefaultFontLibrary.

7.47.2.3 ResolveFontFamily() [3/4]

```
FontFamily VectSharp.IFontLibrary.ResolveFontFamily (
            string fontFamily,
            FontFamily.StandardFontFamilies finalFallback,
            params string[] fallback )
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the *fallback* . If none of the fallback family names or true type files are valid, instantiate a standard font family using the *finalFallback* .

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.
fallback	Names of additional font families or TTF files, which will be tried if the first *fontFamily* is not valid.
finalFallback	The standard name of the font family that will be used if none of the fallback families are valid.

Returns

A FontFamily object corresponding to the first of the specified font families that is valid.

Implemented in VectSharp.FontLibrary.

7.47.2.4 ResolveFontFamily() [4/4]

```
FontFamily VectSharp.IFontLibrary.ResolveFontFamily (
            string fontFamily,
            params string[] fallback )
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using the *fallback* . If none of the fallback family names or true type files are valid, an exception might be raised.

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.
fallback	Names of additional font families or TTF files, which will be tried if the first *fontFamily* is not valid.

Returns

A FontFamily object corresponding to the first of the specified font families that is valid.

Implemented in VectSharp.FontLibrary.

The documentation for this interface was generated from the following file:

- VectSharp/FontLibrary.cs

7.48 VectSharp.IGraphicsContext Interface Reference

This interface should be implemented by classes intended to provide graphics output capability to a Graphics object.

Public Member Functions

- void **Save** ()

  Save the current transform state (rotation, translation, scale). This should be implemented as a LIFO stack.
- void **Restore** ()

  Restore the previous transform state (rotation, translation, scale). This should be implemented as a LIFO stack.
- void Translate (double x, double y)

  Translate the coordinate system origin.
- void Rotate (double angle)

  Rotate the coordinate system around the origin.
- void Scale (double scaleX, double scaleY)

  Scale the coordinate system with respect to the origin.
- void Transform (double a, double b, double c, double d, double e, double f)

  Transform the coordinate system with the specified transformation matrix [[a, c, e], [b, d, f], [0, 0, 1]].
- void FillText (string text, double x, double y)

  Fill a text string using the current Font and TextBaseline.
- void StrokeText (string text, double x, double y)

  Stroke the outline of a text string using the current Font and TextBaseline.
- void MoveTo (double x, double y)

  Change the current point without drawing a line from the previous point. If necessary, start a new figure.
- void LineTo (double x, double y)

  Draw a line from the previous point to the specified point.
- void **Close** ()

  Close the current figure.
- void **Stroke** ()

  Stroke the current path using the current StrokeStyle, LineWidth, LineCap, LineJoin and LineDash.
- void **SetClippingPath** ()

  Set the current clipping path as the intersection of the previous clipping path and the current path.
- void SetFillStyle ((int r, int g, int b, double a) style)

  Set the current FillStyle.
- void SetFillStyle (Brush style)

  Set the current FillStyle.
- void SetStrokeStyle ((int r, int g, int b, double a) style)

  Set the current StrokeStyle.
- void SetStrokeStyle (Brush style)

  Set the current StrokeStyle.
- void CubicBezierTo (double p1X, double p1Y, double p2X, double p2Y, double p3X, double p3Y)

  Add to the current figure a cubic Bezier from the current point to a destination point, with two control points.
- void Rectangle (double x0, double y0, double width, double height)

  Add a rectangle figure to the current path.
- void **Fill** ()

  Fill the current path using the current FillStyle.
- void SetLineDash (LineDash dash)

  Set the current line dash pattern.
- void DrawRasterImage (int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image)

  Draw a raster image.
- void DrawFilteredGraphics (Graphics graphics, IFilter filter)

  Draws a Graphics object, applying the specified filter .

Properties

- double Width [get]

  Width of the graphic surface.
- double Height [get]

  Height of the graphic surface.
- Font Font [get, set]

  The current font.
- TextBaselines TextBaseline [get, set]

  The current text baseline.
- Brush FillStyle [get]

  Current brush used to fill paths.
- Brush StrokeStyle [get]

  Current brush used to stroke paths.
- double LineWidth [get, set]

  Current line width used to stroke paths.
- LineCaps LineCap [set]

  Current line cap used to stroke paths.
- LineJoins LineJoin [set]

  Current line join used to stroke paths.
- string Tag [get, set]

  The current tag. How this can be used depends on each implementation.

7.48.1 Detailed Description

This interface should be implemented by classes intended to provide graphics output capability to a Graphics object.

Definition at line 35 of file Graphics.cs.

7.48.2 Member Function Documentation

7.48.2.1 CubicBezierTo()

```
void VectSharp.IGraphicsContext.CubicBezierTo (
            double p1X,
            double p1Y,
            double p2X,
            double p2Y,
            double p3X,
            double p3Y )
```

Add to the current figure a cubic Bezier from the current point to a destination point, with two control points.

Parameters

p1X	The horizontal coordinate of the first control point.
p1Y	The vertical coordinate of the first control point.
p2X	The horizontal coordinate of the second control point.
p2Y	The vertical coordinate of the second control point.
p3X	The horizontal coordinate of the destination point.
p3Y	The vertical coordinate of the destination point.

7.48.2.2 DrawFilteredGraphics()

```
void VectSharp.IGraphicsContext.DrawFilteredGraphics (
            Graphics graphics,
            IFilter filter )
```

Draws a Graphics object, applying the specified *filter* .

Parameters

graphics	The Graphics object to draw on the current Graphics object.
filter	An IFilter object, representing the filter to apply to the *graphics* object.

7.48.2.3 DrawRasterImage()

```
void VectSharp.IGraphicsContext.DrawRasterImage (
            int sourceX,
            int sourceY,
            int sourceWidth,
            int sourceHeight,
            double destinationX,
            double destinationY,
            double destinationWidth,
            double destinationHeight,
            RasterImage image )
```

Draw a raster image.

Parameters

sourceX	The horizontal coordinate of the top-left corner of the rectangle delimiting the source area of the image.
sourceY	The vertical coordinate of the top-left corner of the rectangle delimiting the source area of the image.
sourceWidth	The width of the rectangle delimiting the source area of the image.
sourceHeight	The height of the rectangle delimiting the source area of the image.
destinationX	The horizontal coordinate of the top-left corner of the rectangle delimiting the destination area of the image.
destinationY	The vertical coordinate of the top-left corner of the rectangle delimiting the destination area of the image.
destinationWidth	The width of the rectangle delimiting the destination area of the image.
destinationHeight	The height of the rectangle delimiting the destination area of the image.
image	The image to draw.

7.48.2.4 FillText()

```
void VectSharp.IGraphicsContext.FillText (
            string text,
            double x,
            double y )
```

Fill a text string using the current Font and TextBaseline.

Parameters

text	The string to draw.
x	The horizontal coordinate of the text origin.
y	The vertical coordinate of the text origin.

7.48.2.5 LineTo()

```
void VectSharp.IGraphicsContext.LineTo (
            double x,
            double y )
```

Draw a line from the previous point to the specified point.

Parameters

x	The horizontal coordinate of the point.
y	The vertical coordinate of the point.

7.48.2.6 MoveTo()

```
void VectSharp.IGraphicsContext.MoveTo (
            double x,
            double y )
```

Change the current point without drawing a line from the previous point. If necessary, start a new figure.

Parameters

x	The horizontal coordinate of the point.
y	The vertical coordinate of the point.

7.48.2.7 Rectangle()

```
void VectSharp.IGraphicsContext.Rectangle (
            double x0,
            double y0,
            double width,
            double height )
```

Add a rectangle figure to the current path.

Parameters

x0	The horizontal coordinate of the top-left corner of the rectangle.
y0	The vertical coordinate of the top-left corner of the rectangle.
width	The width of corner of the rectangle.
height	The height of corner of the rectangle.

7.48.2.8 Rotate()

```
void VectSharp.IGraphicsContext.Rotate (
            double angle )
```

Rotate the coordinate system around the origin.

Parameters

angle	The angle (in radians) by which to rotate the coordinate system.

7.48.2.9 Scale()

```
void VectSharp.IGraphicsContext.Scale (
            double scaleX,
            double scaleY )
```

Scale the coordinate system with respect to the origin.

Parameters

scaleX	The horizontal scale.
scaleY	The vertical scale.

7.48.2.10 SetFillStyle() [1/2]

```
void VectSharp.IGraphicsContext.SetFillStyle (
            (int r, int g, int b, double a) style )
```

Set the current FillStyle.

Parameters

style	A ValueTuple<Int32, Int32, Int32, Double> containing component information for the colour. For r, g, and b, range: [0, 255]; for a, range: [0, 1].

7.48.2.11 SetFillStyle() [2/2]

```
void VectSharp.IGraphicsContext.SetFillStyle (
            Brush style )
```

Set the current FillStyle.

Parameters

style	The new fill style.

7.48.2.12 SetLineDash()

```
void VectSharp.IGraphicsContext.SetLineDash (
            LineDash dash )
```

Set the current line dash pattern.

Parameters

dash	The line dash pattern.

7.48.2.13 SetStrokeStyle() [1/2]

```
void VectSharp.IGraphicsContext.SetStrokeStyle (
            (int r, int g, int b, double a) style )
```

Set the current StrokeStyle.

Parameters

style	A ValueTuple<Int32, Int32, Int32, Double> containing component information for the colour. For r, g, and b, range: [0, 255]; for a, range: [0, 1].

7.48.2.14 SetStrokeStyle() [2/2]

```
void VectSharp.IGraphicsContext.SetStrokeStyle (
            Brush style )
```

Set the current StrokeStyle.

Parameters

style	The new stroke style.

7.48.2.15 StrokeText()

```
void VectSharp.IGraphicsContext.StrokeText (
            string text,
            double x,
            double y )
```

Stroke the outline of a text string using the current Font and TextBaseline.

Parameters

text	The string to draw.
x	The horizontal coordinate of the text origin.
y	The vertical coordinate of the text origin.

7.48.2.16 Transform()

```
void VectSharp.IGraphicsContext.Transform (
            double a,
            double b,
            double c,
            double d,
            double e,
            double f )
```

Transform the coordinate system with the specified transformation matrix [[a, c, e], [b, d, f], [0, 0, 1]].

Parameters

a	The first element of the first column.
b	The second element of the first column.
c	The first element of the second column.
d	The second element of the second column.
e	The first element of the third column.
f	The second element of the third column.

7.48.2.17 Translate()

```
void VectSharp.IGraphicsContext.Translate (
            double x,
            double y )
```

Translate the coordinate system origin.

Parameters

x	The horizontal translation.
y	The vertical translation.

7.48.3 Property Documentation

7.48.3.1 FillStyle

```
Brush VectSharp.IGraphicsContext.FillStyle  [get]
```

Current brush used to fill paths.

Definition at line 146 of file Graphics.cs.

7.48.3.2 Font

```
Font VectSharp.IGraphicsContext.Font  [get], [set]
```

The current font.

Definition at line 91 of file Graphics.cs.

7.48.3.3 Height

```
double VectSharp.IGraphicsContext.Height [get]
```

Height of the graphic surface.

Definition at line 45 of file Graphics.cs.

7.48.3.4 LineCap

```
LineCaps VectSharp.IGraphicsContext.LineCap [set]
```

Current line cap used to stroke paths.

Definition at line 210 of file Graphics.cs.

7.48.3.5 LineJoin

```
LineJoins VectSharp.IGraphicsContext.LineJoin [set]
```

Current line join used to stroke paths.

Definition at line 215 of file Graphics.cs.

7.48.3.6 LineWidth

```
double VectSharp.IGraphicsContext.LineWidth [get], [set]
```

Current line width used to stroke paths.

Definition at line 205 of file Graphics.cs.

7.48.3.7 StrokeStyle

```
Brush VectSharp.IGraphicsContext.StrokeStyle [get]
```

Current brush used to stroke paths.

Definition at line 163 of file Graphics.cs.

7.48.3.8 Tag

```
string VectSharp.IGraphicsContext.Tag  [get], [set]
```

The current tag. How this can be used depends on each implementation.

Definition at line 226 of file Graphics.cs.

7.48.3.9 TextBaseline

```
TextBaselines VectSharp.IGraphicsContext.TextBaseline  [get], [set]
```

The current text baseline.

Definition at line 96 of file Graphics.cs.

7.48.3.10 Width

```
double VectSharp.IGraphicsContext.Width  [get]
```

Width of the graphic surface.

Definition at line 40 of file Graphics.cs.

The documentation for this interface was generated from the following file:

- VectSharp/Graphics.cs

7.49 VectSharp.ThreeD.ILightSource Interface Reference

Represents a light source.

Inheritance diagram for VectSharp.ThreeD.ILightSource:

Public Member Functions

- LightIntensity GetLightAt (Point3D point)

  Computes the light intensity at the specified point, without taking into account any obstructions.
- double GetObstruction (Point3D point, IEnumerable< Triangle3DElement > shadowingTriangles)

  Determines the amount of obstruction of the light that results at point due to the specified shadowingTriangles .

Properties

- bool CastsShadow `[get]`

  Determines whether the light casts a shadow or not.

7.49.1 Detailed Description

Represents a light source.

Definition at line 65 of file Lights.cs.

7.49.2 Member Function Documentation

7.49.2.1 GetLightAt()

```
LightIntensity VectSharp.ThreeD.ILightSource.GetLightAt (
            Point3D point )
```

Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	The Point3DElement at which the light intensity should be computed.

Returns

Implemented in VectSharp.ThreeD.AmbientLightSource, VectSharp.ThreeD.ParallelLightSource, VectSharp.ThreeD.PointLightSource, VectSharp.ThreeD.SpotlightLightSource, VectSharp.ThreeD.MaskedLightSource, and VectSharp.ThreeD.AreaLightSource.

7.49.2.2 GetObstruction()

```
double VectSharp.ThreeD.ILightSource.GetObstruction (
            Point3D point,
            IEnumerable< Triangle3DElement > shadowingTriangles )
```

Determines the amount of obstruction of the light that results at *point* due to the specified *shadowingTriangles* .

Parameters

point	The Point3D at which the obstruction should be computed.
shadowingTriangles	A collection of Triangle3DElement casting shadows.

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Implemented in VectSharp.ThreeD.AmbientLightSource, VectSharp.ThreeD.ParallelLightSource, VectSharp.ThreeD.PointLightSource, VectSharp.ThreeD.SpotlightLightSource, VectSharp.ThreeD.MaskedLightSource, and VectSharp.ThreeD.AreaLightSource.

7.49.3 Property Documentation

7.49.3.1 CastsShadow

```
bool VectSharp.ThreeD.ILightSource.CastsShadow  [get]
```

Determines whether the light casts a shadow or not.

Implemented in VectSharp.ThreeD.AmbientLightSource, VectSharp.ThreeD.ParallelLightSource, VectSharp.ThreeD.PointLightSource, VectSharp.ThreeD.SpotlightLightSource, VectSharp.ThreeD.MaskedLightSource, and VectSharp.ThreeD.AreaLightSource.

Definition at line 77 of file Lights.cs.

The documentation for this interface was generated from the following file:

- VectSharp.ThreeD/Lights.cs

7.50 VectSharp.Filters.ILocationInvariantFilter Interface Reference

Represents a filter that can be applied to an image regardless of its location on the graphics surface.

Inheritance diagram for VectSharp.Filters.ILocationInvariantFilter:

Public Member Functions

- RasterImage Filter (RasterImage image, double scale)

  Applies the filter to a RasterImage.

Additional Inherited Members

7.50.1 Detailed Description

Represents a filter that can be applied to an image regardless of its location on the graphics surface.

Definition at line 41 of file Filters.cs.

7.50.2 Member Function Documentation

7.50.2.1 Filter()

```
RasterImage VectSharp.Filters.ILocationInvariantFilter.Filter (
            RasterImage image,
            double scale )
```

Applies the filter to a RasterImage.

Parameters

image	The RasterImage to which the filter will be applied.
scale	The scale of the image with respect to the filter.

Returns

A new RasterImage containing the filtered image. The source *image* is left unaltered.

Implemented in VectSharp.Filters.BoxBlurFilter, VectSharp.Filters.ColourMatrixFilter, VectSharp.Filters.CompositeLocationInvariantFil
VectSharp.Filters.ConvolutionFilter, and VectSharp.Filters.GaussianBlurFilter.

The documentation for this interface was generated from the following file:

- VectSharp/Filters/Filters.cs

7.51 VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter Class Reference

Contains methods to render a Page to an Image.

Classes

- class UnknownFormatException

  The exception that is raised when the output file format is not specified and the file name does not have an extension corresponding to a known file format.

Static Public Member Functions

- static Image< SixLabors.ImageSharp.PixelFormats.Rgba32 > SaveAsImage (this Page page, double scale=1)

  Render the page to an Image object.
- static void SaveAsImage (this Page page, Stream imageStream, OutputFormats outputFormat, double scale=1)

  Render the page to an image stream.
- static void SaveAsImage (this Page page, string fileName, OutputFormats? outputFormat=null, double scale=1)

  Render the page to an image file.
- static DisposableIntPtr SaveAsRawBytes (this Page pag, out int width, out int height, out int totalSize, double scale=1)

  Render the page to raw pixel data, in 32bpp RGBA format.
- static byte[] SaveAsRawBytes (this Page pag, out int width, out int height, double scale=1)

  Return the page to raw pixel data, in 32bpp RGBA format.
- static RasterImage Rasterise (this Graphics graphics, Rectangle region, double scale, bool interpolate)

  Rasterise a region of a Graphics object.
- static Image SaveAsAnimatedGIF (this Animation animation, double scale=1, double frameRate=50, double durationScaling=1, GifColorTableMode colorTableMode=GifColorTableMode.Local)

  Saves the animation to an animated GIF.
- static void SaveAsAnimatedGIF (this Animation animation, Stream imageStream, double scale=1, double frameRate=50, double durationScaling=1, GifColorTableMode colorTableMode=GifColorTableMode.Local)

  Saves the animation to an animated GIF stream.
- static void SaveAsAnimatedGIF (this Animation animation, string fileName, double scale=1, double frame↩Rate=50, double durationScaling=1, GifColorTableMode colorTableMode=GifColorTableMode.Local)

  Saves the animation to an animated GIF file.
- static void SaveAsAnimatedPNG (this Animation animation, Stream imageStream, double scale=1, double frameRate=60, double durationScaling=1, AnimatedPNG.InterframeCompression interframe↩Compression=AnimatedPNG.InterframeCompression.First)

  Saves the animation to a stream in animated PNG format.
- static void SaveAsAnimatedPNG (this Animation animation, string fileName, double scale=1, double frameRate=60, double durationScaling=1, AnimatedPNG.InterframeCompression interframe↩Compression=AnimatedPNG.InterframeCompression.First)

  Saves the animation to an animated PNG file.

7.51.1 Detailed Description

Contains methods to render a Page to an Image.

Definition at line 1038 of file ImageSharpContext.cs.

7.51.2 Member Function Documentation

7.51.2.1 Rasterise()

```
static RasterImage VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.Rasterise (
            this Graphics graphics,
            Rectangle region,
            double scale,
            bool interpolate )  [static]
```

Rasterise a region of a Graphics object.

Parameters

graphics	The Graphics object that will be rasterised.
region	The region of the *graphics* that will be rasterised.
scale	The scale at which the image will be rendered.
interpolate	Whether the resulting image should be interpolated or not when it is drawn on another Graphics surface.

Returns

A RasterImage containing the rasterised graphics.

Definition at line 1270 of file ImageSharpContext.cs.

7.51.2.2 SaveAsAnimatedGIF() [1/3]

```
static Image VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.SaveAsAnimatedGIF (
            this Animation animation,
            double scale = 1,
            double frameRate = 50,
            double durationScaling = 1,
            GifColorTableMode colorTableMode = GifColorTableMode.Local )  [static]
```

Saves the animation to an animated GIF.

Parameters

animation	The animation to export.
scale	The scale at which the animation will be rendered.
frameRate	The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated GIF specification at 50 fps.
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
colorTableMode	Determines whether a single colour table should be used for the whole image, or if a different colour table should be used for each frame.

Returns

An Image containing the animated GIF.

Definition at line 1318 of file ImageSharpContext.cs.

7.51.2.3 SaveAsAnimatedGIF() [2/3]

```
static void VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.SaveAsAnimatedGIF (
            this Animation animation,
            Stream imageStream,
            double scale = 1,
            double frameRate = 50,
            double durationScaling = 1,
            GifColorTableMode colorTableMode = GifColorTableMode.Local )  [static]
```

Saves the animation to an animated GIF stream.

Parameters

animation	The animation to export.
imageStream	The stream on which the animated GIF will be written.
scale	The scale at which the animation will be rendered.
frameRate	The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated GIF specification at 50 fps.
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
colorTableMode	Determines whether a single colour table should be used for the whole image, or if a different colour table should be used for each frame.

Definition at line 1377 of file ImageSharpContext.cs.

7.51.2.4 SaveAsAnimatedGIF() [3/3]

```
static void VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.SaveAsAnimatedGIF (
            this Animation animation,
            string fileName,
            double scale = 1,
            double frameRate = 50,
            double durationScaling = 1,
            GifColorTableMode colorTableMode = GifColorTableMode.Local )  [static]
```

Saves the animation to an animated GIF file.

Parameters

animation	The animation to export.
fileName	The output file to create.

Parameters

scale	The scale at which the animation will be rendered.
frameRate	The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated GIF specification at 50 fps.
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
colorTableMode	Determines whether a single colour table should be used for the whole image, or if a different colour table should be used for each frame.

Definition at line 1393 of file ImageSharpContext.cs.

7.51.2.5 SaveAsAnimatedPNG() [1/2]

```
static void VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.SaveAsAnimatedPNG (
            this Animation animation,
            Stream imageStream,
            double scale = 1,
            double frameRate = 60,
            double durationScaling = 1,
            AnimatedPNG.InterframeCompression interframeCompression = AnimatedPNG.InterframeCompression.First
)  [static]
```

Saves the animation to a stream in animated PNG format.

Parameters

animation	The animation to export.
imageStream	The stream on which the animated PNG will be written.
scale	The scale at which the animation will be rendered.
frameRate	The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated PNG specification at 90 fps.
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
interframeCompression	The kind of compression that will be used to reduce file size. Note that if the animation has a transparent background, no compression can be performed, and the value of this parameter is ignored.

Definition at line 1410 of file ImageSharpContext.cs.

7.51.2.6 SaveAsAnimatedPNG() [2/2]

```
static void VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.SaveAsAnimatedPNG (
            this Animation animation,
            string fileName,
```

```
            double scale = 1,
            double frameRate = 60,
            double durationScaling = 1,
            AnimatedPNG.InterframeCompression interframeCompression = AnimatedPNG.InterframeCompression.First
)  [static]
```

Saves the animation to an animated PNG file.

Parameters

animation	The animation to export.
fileName	The output file to create.
scale	The scale at which the animation will be rendered.
frameRate	The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated PNG specification at 90 fps.
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
interframeCompression	The kind of compression that will be used to reduce file size. Note that if the animation has a transparent background, no compression can be performed, and the value of this parameter is ignored.

Definition at line 1498 of file ImageSharpContext.cs.

7.51.2.7 SaveAsImage() [1/3]

```
static Image< SixLabors.ImageSharp.PixelFormats.Rgba32 > VectSharp.Raster.ImageSharp.Image↩
SharpContextInterpreter.SaveAsImage (
            this Page page,
            double scale = 1 )  [static]
```

Render the page to an Image object.

Parameters

page	The Page to render.
scale	The scale to be used when rasterising the page. This will determine the width and height of the Image.

Returns

An Image containing the rasterised page.

Definition at line 1046 of file ImageSharpContext.cs.

7.51.2.8 SaveAsImage() [2/3]

```
static void VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.SaveAsImage (
            this Page page,
```

```
            Stream imageStream,
            OutputFormats outputFormat,
            double scale = 1 )  [static]
```

Render the page to an image stream.

Parameters

page	The Page to render.
imageStream	The Stream on which the image data will be written.
outputFormat	The format of the image that will be created.
scale	The scale to be used when rasterising the page. This will determine the width and height of the image.

Definition at line 1062 of file ImageSharpContext.cs.

7.51.2.9 SaveAsImage() [3/3]

```
static void VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.SaveAsImage (
            this Page page,
            string fileName,
            OutputFormats?  outputFormat = null,
            double scale = 1 )  [static]
```

Render the page to an image file.

Parameters

page	The Page to render.
fileName	The path of the file where the image will be saved.
outputFormat	The format of the image that will be created. If this is `null` (the default), the format is desumed from the extension of the file.
scale	The scale to be used when rasterising the page. This will determine the width and height of the image.

Definition at line 1120 of file ImageSharpContext.cs.

7.51.2.10 SaveAsRawBytes() [1/2]

```
static byte[] VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.SaveAsRawBytes (
            this Page pag,
            out int width,
            out int height,
            double scale = 1 )  [static]
```

Return the page to raw pixel data, in 32bpp RGBA format.

Parameters

pag	The Page to render.
scale	The scale to be used when rasterising the page. This will determine the width and height of the image.
width	The width of the rendered image.
height	The height of the rendered image.

Returns

A byte array containing the raw pixel data.

Definition at line 1233 of file ImageSharpContext.cs.

7.51.2.11 SaveAsRawBytes() [2/2]

```
static DisposableIntPtr VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.SaveAsRaw↩
Bytes (
            this Page pag,
            out int width,
            out int height,
            out int totalSize,
            double scale = 1 )  [static]
```

Render the page to raw pixel data, in 32bpp RGBA format.

Parameters

pag	The Page to render.
scale	The scale to be used when rasterising the page. This will determine the width and height of the image.
width	The width of the rendered image.
height	The height of the rendered image.
totalSize	The size in bytes of the raw pixel data.

Returns

A DisposableIntPtr containing a pointer to the raw pixel data, stored in unmanaged memory. Dispose this object to release the unmanaged memory.

Definition at line 1190 of file ImageSharpContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.Raster.ImageSharp/ImageSharpContext.cs

7.52 VectSharp.ImageSharpUtils.ImageURIParser Class Reference

Provides a method to parse an image URI into a page.

Static Public Member Functions

- static Func< string, bool, Page > Parser (Func< string, bool, Page > parseSVG)

  Parses an image URI into a page. This is intended to replace the default image URI interpreter in `VectSharp.SVG.Parser.ParseImageURI`. To do this, use something like:

7.52.1 Detailed Description

Provides a method to parse an image URI into a page.

Definition at line 29 of file ImageUriParser.cs.

7.52.2 Member Function Documentation

7.52.2.1 Parser()

```
static Func< string, bool, Page > VectSharp.ImageSharpUtils.ImageURIParser.Parser (
            Func< string, bool, Page > parseSVG )  [static]
```

Parses an image URI into a page. This is intended to replace the default image URI interpreter in `VectSharp.SVG.Parser.ParseImageURI`. To do this, use something like:

```
VectSharp.SVG.Parser.ParseImageURI = VectSharp.ImageSharpUtils.ImageURIParser.Parser(Vec
```

Parameters

parseSVG	A function to parse an SVG image uri into a page. You should pass `VectSharp.SVG.Parser.ParseSVGURI` as this argument.

Returns

A function to parse an image URI into a page.

Definition at line 37 of file ImageUriParser.cs.

The documentation for this class was generated from the following file:

- VectSharp.ImageSharpUtils/ImageUriParser.cs

7.53 VectSharp.MuPDFUtils.ImageURIParser Class Reference

Provides a method to parse an image URI into a page.

Static Public Member Functions

- static Func< string, bool, Page > Parser (Func< string, bool, Page > parseSVG)

  Parses an image URI into a page. This is intended to replace the default image URI interpreter in `VectSharp.SVG.Parser.ParseImageURI`. To do this, use something like:

7.53.1 Detailed Description

Provides a method to parse an image URI into a page.

Definition at line 29 of file ImageURIParser.cs.

7.53.2 Member Function Documentation

7.53.2.1 Parser()

```
static Func< string, bool, Page > VectSharp.MuPDFUtils.ImageURIParser.Parser (
            Func< string, bool, Page > parseSVG )  [static]
```

Parses an image URI into a page. This is intended to replace the default image URI interpreter in `VectSharp.SVG.Parser.ParseImageURI`. To do this, use something like:

```
VectSharp.SVG.Parser.ParseImageURI = VectSharp.MuPDFUtils.ImageURIParser.Parser(VectShar
```

Parameters

parseSVG	A function to parse an SVG image uri into a page. You should pass `VectSharp.SVG.Parser.ParseSVGURI` as this argument.

Returns

A function to parse an image URI into a page.

Definition at line 37 of file ImageURIParser.cs.

The documentation for this class was generated from the following file:

- VectSharp.MuPDFUtils/ImageURIParser.cs

7.54 VectSharp.ThreeD.IMaterial Interface Reference

Represents a material used to the determine the appearance of Triangle3DElement.

Inheritance diagram for VectSharp.ThreeD.IMaterial:

Public Member Functions

- Colour GetColour (Point3D point, NormalizedVector3D surfaceNormal, Camera camera, IList< ILightSource > lights, IList< double > obstructions)

  Obtains the Colour at the specified point.

7.54.1 Detailed Description

Represents a material used to the determine the appearance of Triangle3DElement.

Definition at line 31 of file Materials.cs.

7.54.2 Member Function Documentation

7.54.2.1 GetColour()

```
Colour VectSharp.ThreeD.IMaterial.GetColour (
            Point3D point,
            NormalizedVector3D surfaceNormal,
            Camera camera,
            IList< ILightSource > lights,
            IList< double > obstructions )
```

Obtains the Colour at the specified point.

Parameters

point	The point whose colour should be determined.
surfaceNormal	The normal to the surface at the specified *point* .
camera	The camera being used to render the scene.
lights	A list of light sources that are present in the scene.
obstructions	A list of values indicating how obstructed each light source is.

Returns

The Colour of the specified point.

Implemented in VectSharp.ThreeD.ColourMaterial, and VectSharp.ThreeD.PhongMaterial.

The documentation for this interface was generated from the following file:

- VectSharp.ThreeD/Materials.cs

7.55 VectSharp.ThreeD.IScene Interface Reference

Represents a 3D scene.

Inheritance diagram for VectSharp.ThreeD.IScene:

Public Member Functions

- void AddElement (Element3D element)

  Adds the specified element to the scene.
- void AddRange (IEnumerable< Element3D > elements)

  Adds the specified elements to the scene.
- void Replace (Func< Element3D, Element3D > replacementFunction)

  Replaces each element in the scene with the element returned by the replacementFunction .
- void Replace (Func< Element3D, IEnumerable< Element3D > > replacementFunction)

  Replaces each element in the scene with the element(s) returned by the replacementFunction .

Properties

- IEnumerable< Element3D > SceneElements `[get]`

  The Element3Ds constituting the scene.
- object SceneLock `[get]`

  An object used to synchronise multithreaded rendering of the same scene.

7.55.1 Detailed Description

Represents a 3D scene.

Definition at line 26 of file Scene.cs.

7.55.2 Member Function Documentation

7.55.2.1 AddElement()

```
void VectSharp.ThreeD.IScene.AddElement (
            Element3D element )
```

Adds the specified *element* to the scene.

Parameters

element	The Element3D to add.

Implemented in VectSharp.ThreeD.Scene.

7.55.2.2 AddRange()

```
void VectSharp.ThreeD.IScene.AddRange (
            IEnumerable< Element3D > elements )
```

Adds the specified *elements* to the scene.

Parameters

elements	A collection of Element3Ds to add.

Implemented in VectSharp.ThreeD.Scene.

7.55.2.3 Replace() [1/2]

```
void VectSharp.ThreeD.IScene.Replace (
            Func< Element3D, Element3D > replacementFunction )
```

Replaces each element in the scene with the element returned by the *replacementFunction* .

Parameters

replacementFunction	A function replacing each Element3D in the scene with another Element3D.

Implemented in VectSharp.ThreeD.Scene.

7.55.2.4 Replace() [2/2]

```
void VectSharp.ThreeD.IScene.Replace (
            Func< Element3D, IEnumerable< Element3D > > replacementFunction )
```

Replaces each element in the scene with the element(s) returned by the *replacementFunction* .

Parameters

replacementFunction	A function replacing each Element3D in the scene with 0 or more Element3Ds.

Implemented in VectSharp.ThreeD.Scene.

7.55.3 Property Documentation

7.55.3.1 SceneElements

```
IEnumerable<Element3D> VectSharp.ThreeD.IScene.SceneElements [get]
```

The Element3Ds constituting the scene.

Implemented in VectSharp.ThreeD.Scene.

Definition at line 31 of file Scene.cs.

7.55.3.2 SceneLock

```
object VectSharp.ThreeD.IScene.SceneLock [get]
```

An object used to synchronise multithreaded rendering of the same scene.

Implemented in VectSharp.ThreeD.Scene.

Definition at line 60 of file Scene.cs.

The documentation for this interface was generated from the following file:

- VectSharp.ThreeD/Scene.cs

7.56 VectSharp.ThreeD.LightIntensity Struct Reference

Represents the intensity of a light source at a particular point.

Public Member Functions

- LightIntensity (double intensity, NormalizedVector3D direction)

  Creates a new LightIntensity.
- void Deconstruct (out double intensity, out NormalizedVector3D direction)

  Deconstructs the struct.

Public Attributes

- double Intensity

  The intensity of the light.
- NormalizedVector3D Direction

  The direction towards from which the light comes.

7.56.1 Detailed Description

Represents the intensity of a light source at a particular point.

Definition at line 27 of file Lights.cs.

7.56.2 Constructor & Destructor Documentation

7.56.2.1 LightIntensity()

```
VectSharp.ThreeD.LightIntensity.LightIntensity (
            double intensity,
            NormalizedVector3D direction )
```

Creates a new LightIntensity.

Parameters

intensity	The intensity of the light.
direction	The direction from which the light comes.

Definition at line 44 of file Lights.cs.

7.56.3 Member Function Documentation

7.56.3.1 Deconstruct()

```
void VectSharp.ThreeD.LightIntensity.Deconstruct (
            out double intensity,
            out NormalizedVector3D direction )
```

Deconstructs the struct.

Parameters

intensity	This parameter will hold the Intensity of the light.
direction	This parameter will hold the Direction of the light.

Definition at line 55 of file Lights.cs.

7.56.4 Member Data Documentation

7.56.4.1 Direction

```
NormalizedVector3D VectSharp.ThreeD.LightIntensity.Direction
```

The direction towards from which the light comes.

Definition at line 37 of file Lights.cs.

7.56.4.2 Intensity

```
double VectSharp.ThreeD.LightIntensity.Intensity
```

The intensity of the light.

Definition at line 32 of file Lights.cs.

The documentation for this struct was generated from the following file:

- VectSharp.ThreeD/Lights.cs

7.57 VectSharp.LinearEasing Class Reference

Describes a linear easing (i.e., no easing).

Inheritance diagram for VectSharp.LinearEasing:

Public Member Functions

- LinearEasing ()

  Creates a new LinearEasing.
- double Ease (double value)

  Applies the easing to the specified transition offset.

Parameters

value	*The transition offset (ranging from 0 to 1).*

Returns

The eased transition offset value.

7.57.1 Detailed Description

Describes a linear easing (i.e., no easing).

Definition at line 1365 of file Animation.cs.

7.57.2 Constructor & Destructor Documentation

7.57.2.1 LinearEasing()

```
VectSharp.LinearEasing.LinearEasing ( )
```

Creates a new LinearEasing.

Definition at line 1370 of file Animation.cs.

7.57.3 Member Function Documentation

7.57.3.1 Ease()

```
double VectSharp.LinearEasing.Ease (
            double value )
```

Applies the easing to the specified transition offset.

Parameters

value	The transition offset (ranging from 0 to 1).

Returns

The eased transition offset value.

Implements VectSharp.IEasing.

The documentation for this class was generated from the following file:

- VectSharp/Animation.cs

7.58 VectSharp.LinearGradientBrush Class Reference

Represents a brush painting with a linear gradient.

Inheritance diagram for VectSharp.LinearGradientBrush:

Public Member Functions

- LinearGradientBrush (Point startPoint, Point endPoint, IEnumerable< GradientStop > gradientStops)

  Creates a new LinearGradientBrush with the specified start point, end point and gradient stops.
- LinearGradientBrush (Point startPoint, Point endPoint, params GradientStop[ ] gradientStops)

  Creates a new LinearGradientBrush with the specified start point, end point and gradient stops.
- LinearGradientBrush RelativeTo (Graphics referenceGraphics)

  Returns a LinearGradientBrush with the same gradient stops as the current instance, whose start and end point correspond to the points of the current instance in the original reference frame of the referenceGraphics . This involves computing the current transform matrix of the referenceGraphics , inverting it, and applying the inverse matrix to the StartPoint and EndPoint of the current instance.
- override Brush MultiplyOpacity (double opacity)

  Returns a brush corresponding the current instance, with the specified opacity multiplication applied.

Parameters

opacity	*The value that will be used to multiply the opacity of the brush.*

Returns

A brush corresponding the current instance, with the specified opacity multiplication applied.

Properties

- Point StartPoint `[get]`

  The starting point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
- Point EndPoint `[get]`

  The end point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.

Additional Inherited Members

7.58.1 Detailed Description

Represents a brush painting with a linear gradient.

Definition at line 277 of file Brush.cs.

7.58.2 Constructor & Destructor Documentation

7.58.2.1 LinearGradientBrush() [1/2]

```
VectSharp.LinearGradientBrush.LinearGradientBrush (
            Point startPoint,
            Point endPoint,
            IEnumerable< GradientStop > gradientStops )
```

Creates a new LinearGradientBrush with the specified start point, end point and gradient stops.

Parameters

startPoint	The starting point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
endPoint	The ending point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
gradientStops	The colour stops in the gradient.

Definition at line 295 of file Brush.cs.

7.58.2.2 LinearGradientBrush() [2/2]

```
VectSharp.LinearGradientBrush.LinearGradientBrush (
            Point startPoint,
            Point endPoint,
            params GradientStop[] gradientStops )
```

Creates a new LinearGradientBrush with the specified start point, end point and gradient stops.

Parameters

startPoint	The starting point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
endPoint	The ending point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
gradientStops	The colour stops in the gradient.

Definition at line 309 of file Brush.cs.

7.58.3 Member Function Documentation

7.58.3.1 MultiplyOpacity()

```
override Brush VectSharp.LinearGradientBrush.MultiplyOpacity (
            double opacity )  [virtual]
```

Returns a brush corresponding the current instance, with the specified *opacity* multiplication applied.

Parameters

opacity	The value that will be used to multiply the opacity of the brush.

Returns

A brush corresponding the current instance, with the specified *opacity* multiplication applied.

Implements VectSharp.Brush.

Definition at line 391 of file Brush.cs.

7.58.3.2 RelativeTo()

```
LinearGradientBrush VectSharp.LinearGradientBrush.RelativeTo (
            Graphics referenceGraphics )
```

Returns a LinearGradientBrush with the same gradient stops as the current instance, whose start and end point correspond to the points of the current instance in the original reference frame of the *referenceGraphics* . This involves computing the current transform matrix of the *referenceGraphics* , inverting it, and applying the inverse matrix to the StartPoint and EndPoint of the current instance.

Parameters

referenceGraphics	The Graphics whose original reference frame is to be used.

Returns

A LinearGradientBrush with the same gradient stops as the current instance, whose start and end point correspond to the points of the current instance in the original reference frame of the *referenceGraphics* .

Definition at line 341 of file Brush.cs.

7.58.4 Property Documentation

7.58.4.1 EndPoint

```
Point VectSharp.LinearGradientBrush.EndPoint  [get]
```

The end point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.

Definition at line 287 of file Brush.cs.

7.58.4.2 StartPoint

```
Point VectSharp.LinearGradientBrush.StartPoint  [get]
```

The starting point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.

Definition at line 282 of file Brush.cs.

The documentation for this class was generated from the following file:

- VectSharp/Brush.cs

7.59 VectSharp.LineDash Struct Reference

Represents instructions on how to paint a dashed line.

Public Member Functions

- LineDash (double unitsOn, double unitsOff, double phase)

  Define a new line dash pattern.

Public Attributes

- double UnitsOn

  Length of the "on" (painted) segment.
- double UnitsOff

  Length of the "off" (not painted) segment.
- double Phase

  Position in the dash pattern at which the line starts.

Static Public Attributes

- static LineDash SolidLine = new LineDash(0, 0, 0)

  A solid (not dashed) line

7.59.1 Detailed Description

Represents instructions on how to paint a dashed line.

Definition at line 112 of file Enums.cs.

7.59.2 Constructor & Destructor Documentation

7.59.2.1 LineDash()

```
VectSharp.LineDash.LineDash (
            double unitsOn,
            double unitsOff,
            double phase )
```

Define a new line dash pattern.

Parameters

unitsOn	The length of the "on" (painted) segment.
unitsOff	The length of the "off" (not painted) segment.
phase	The position in the dash pattern at which the line starts.

Definition at line 140 of file Enums.cs.

7.59.3 Member Data Documentation

7.59.3.1 Phase

```
double VectSharp.LineDash.Phase
```

Position in the dash pattern at which the line starts.

Definition at line 132 of file Enums.cs.

7.59.3.2 SolidLine

```
LineDash VectSharp.LineDash.SolidLine = new LineDash(0, 0, 0)  [static]
```

A solid (not dashed) line

Definition at line 117 of file Enums.cs.

7.59.3.3 UnitsOff

```
double VectSharp.LineDash.UnitsOff
```

Length of the "off" (not painted) segment.

Definition at line 127 of file Enums.cs.

7.59.3.4 UnitsOn

```
double VectSharp.LineDash.UnitsOn
```

Length of the "on" (painted) segment.

Definition at line 122 of file Enums.cs.

The documentation for this struct was generated from the following file:

- VectSharp/Enums.cs

7.60 VectSharp.Markdown.Margins Class Reference

Represents the margins of a page.

Public Member Functions

- Margins (double left, double top, double right, double bottom)

  Creates a new Margins instance.

Properties

- double Left `[get]`

  The left margin.
- double Right `[get]`

  The right margin.
- double Top `[get]`

  The top margin.
- double Bottom `[get]`

  The bottom margin.

7.60.1 Detailed Description

Represents the margins of a page.

Definition at line 185 of file MarkdownContext.cs.

7.60.2 Constructor & Destructor Documentation

7.60.2.1 Margins()

```
VectSharp.Markdown.Margins.Margins (
            double left,
            double top,
            double right,
            double bottom )
```

Creates a new Margins instance.

Parameters

left	The left margin.
top	The top margin.
right	The right margin.
bottom	The bottom margin.

Definition at line 214 of file MarkdownContext.cs.

7.60.3 Property Documentation

7.60.3.1 Bottom

```
double VectSharp.Markdown.Margins.Bottom [get]
```

The bottom margin.

Definition at line 205 of file MarkdownContext.cs.

7.60.3.2 Left

```
double VectSharp.Markdown.Margins.Left [get]
```

The left margin.

Definition at line 190 of file MarkdownContext.cs.

7.60.3.3 Right

```
double VectSharp.Markdown.Margins.Right [get]
```

The right margin.

Definition at line 195 of file MarkdownContext.cs.

7.60.3.4 Top

```
double VectSharp.Markdown.Margins.Top [get]
```

The top margin.

Definition at line 200 of file MarkdownContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.Markdown/MarkdownContext.cs

7.61 VectSharp.MarkdownCanvas.MarkdownCanvasControl Class Reference

A control to display a Markdown document in an Avalonia application.

Inheritance diagram for VectSharp.MarkdownCanvas.MarkdownCanvasControl:

Public Member Functions

- MarkdownCanvasControl ()

  Initialises a new MarkdownCanvasControl.

Static Public Attributes

- static readonly StyledProperty< double > MaxRenderWidthProperty = AvaloniaProperty.Register<MarkdownCanvasControl, double>(nameof(MaxRenderWidth), double.PositiveInfinity)

  Defines the MaxRenderWidth property.
- static readonly StyledProperty< double > MinRenderWidthProperty = AvaloniaProperty.Register<MarkdownCanvasControl, double>(nameof(MinRenderWidth), 200)

  Defines the MinRenderWidth property.
- static readonly StyledProperty< double > MinVariationProperty = AvaloniaProperty.Register<MarkdownCanvasControl, double>(nameof(MinVariation), 10)

  Defines the MinVariation property.
- static readonly StyledProperty< string > DocumentSourceProperty = AvaloniaProperty.Register<MarkdownCanvasControl, string>(nameof(DocumentSource))

  Defines the DocumentSource property.
- static readonly StyledProperty< MarkdownDocument > DocumentProperty = AvaloniaProperty.↩ Register<MarkdownCanvasControl, MarkdownDocument>(nameof(Document))

  Defines the Document property.
- static readonly StyledProperty< AvaloniaContextInterpreter.TextOptions > TextConversionOptionsProperty = AvaloniaProperty.Register<MarkdownCanvasControl, AvaloniaContextInterpreter.TextOptions>(nameof(TextConversionOptior AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary)

  Defines the TextConversionOption property.

Properties

- double MaxRenderWidth `[get, set]`

  The maximum width for the rendered document. This will be used even if the control's client area is larger than this (the alignment of the document within the controll will depend on the control's ContentControl.HorizontalContent↩ Alignment).
- double MinRenderWidth `[get, set]`

  The minimum width for the rendered document. If the control's client area is smaller than this, the horizontal scroll bar will be activated.
- double MinVariation `[get, set]`

  The minimum width variation that triggers a document reflow. If the control is resized, but the width changes by less than this amount, the document is not re-drawn.
- string DocumentSource `[set]`

  Sets the currently displayed document from Markdown source.
- MarkdownDocument Document `[get, set]`

  Gets or sets the currently displayed MarkdownDocument.
- AvaloniaContextInterpreter.TextOptions TextConversionOption `[get, set]`

  Gets or sets the value that determines whether text items should be converted into paths when drawing. Setting this to AvaloniaContextInterpreter.TextOptions.NeverConvert will improve performance if you are using custom fonts, but may cause unexpected results unless the font families being used are of type ResourceFontFamily.
- MarkdownRenderer Renderer `[get]`

  The MarkdownRenderer used to render the Document. You can use the properties of this object to customise the rendering. Note that setting the Avalonia.Controls.Primitives.TemplatedControl.FontSize of the MarkdownCanvasControl will propagate to the Renderer's MarkdownRenderer.BaseFontSize.

7.61.1 Detailed Description

A control to display a Markdown document in an Avalonia application.

Definition at line 35 of file MarkdownCanvas.axaml.cs.

7.61.2 Constructor & Destructor Documentation

7.61.2.1 MarkdownCanvasControl()

```
VectSharp.MarkdownCanvas.MarkdownCanvasControl.MarkdownCanvasControl ( )
```

Initialises a new MarkdownCanvasControl.

Definition at line 133 of file MarkdownCanvas.axaml.cs.

7.61.3 Member Data Documentation

7.61.3.1 DocumentProperty

```
readonly StyledProperty<MarkdownDocument> VectSharp.MarkdownCanvas.MarkdownCanvasControl.↩
DocumentProperty = AvaloniaProperty.Register<MarkdownCanvasControl, MarkdownDocument>(nameof(Document))
[static]
```

Defines the Document property.

Definition at line 95 of file MarkdownCanvas.axaml.cs.

7.61.3.2 DocumentSourceProperty

```
readonly StyledProperty<string> VectSharp.MarkdownCanvas.MarkdownCanvasControl.Document↩
SourceProperty = AvaloniaProperty.Register<MarkdownCanvasControl, string>(nameof(DocumentSource))
[static]
```

Defines the DocumentSource property.

Definition at line 82 of file MarkdownCanvas.axaml.cs.

7.61.3.3 MaxRenderWidthProperty

```
readonly StyledProperty<double> VectSharp.MarkdownCanvas.MarkdownCanvasControl.MaxRender↩
WidthProperty = AvaloniaProperty.Register<MarkdownCanvasControl, double>(nameof(MaxRenderWidth),
double.PositiveInfinity)  [static]
```

Defines the MaxRenderWidth property.

Definition at line 40 of file MarkdownCanvas.axaml.cs.

7.61.3.4 MinRenderWidthProperty

```
readonly StyledProperty<double> VectSharp.MarkdownCanvas.MarkdownCanvasControl.MinRender↩
WidthProperty = AvaloniaProperty.Register<MarkdownCanvasControl, double>(nameof(MinRenderWidth),
200)  [static]
```

Defines the MinRenderWidth property.

Definition at line 54 of file MarkdownCanvas.axaml.cs.

7.61.3.5 MinVariationProperty

```
readonly StyledProperty<double> VectSharp.MarkdownCanvas.MarkdownCanvasControl.MinVariation↩
Property = AvaloniaProperty.Register<MarkdownCanvasControl, double>(nameof(MinVariation), 10)
[static]
```

Defines the MinVariation property.

Definition at line 68 of file MarkdownCanvas.axaml.cs.

7.61.3.6 TextConversionOptionsProperty

```
readonly StyledProperty<AvaloniaContextInterpreter.TextOptions> VectSharp.MarkdownCanvas.↩
MarkdownCanvasControl.TextConversionOptionsProperty = AvaloniaProperty.Register<MarkdownCanvasControl,
AvaloniaContextInterpreter.TextOptions>(nameof(TextConversionOption), AvaloniaContextInterpreter.↩
TextOptions.ConvertIfNecessary)  [static]
```

Defines the TextConversionOption property.

Definition at line 109 of file MarkdownCanvas.axaml.cs.

7.61.4 Property Documentation

7.61.4.1 Document

```
MarkdownDocument VectSharp.MarkdownCanvas.MarkdownCanvasControl.Document  [get], [set]
```

Gets or sets the currently displayed MarkdownDocument.

Definition at line 100 of file MarkdownCanvas.axaml.cs.

7.61.4.2 DocumentSource

```
string VectSharp.MarkdownCanvas.MarkdownCanvasControl.DocumentSource  [set]
```

Sets the currently displayed document from Markdown source.

Definition at line 87 of file MarkdownCanvas.axaml.cs.

7.61.4.3 MaxRenderWidth

```
double VectSharp.MarkdownCanvas.MarkdownCanvasControl.MaxRenderWidth [get], [set]
```

The maximum width for the rendered document. This will be used even if the control's client area is larger than this (the alignment of the document within the controll will depend on the control's ContentControl.HorizontalContent↩ Alignment).

Definition at line 45 of file MarkdownCanvas.axaml.cs.

7.61.4.4 MinRenderWidth

```
double VectSharp.MarkdownCanvas.MarkdownCanvasControl.MinRenderWidth [get], [set]
```

The minimum width for the rendered document. If the control's client area is smaller than this, the horizontal scroll bar will be activated.

Definition at line 59 of file MarkdownCanvas.axaml.cs.

7.61.4.5 MinVariation

```
double VectSharp.MarkdownCanvas.MarkdownCanvasControl.MinVariation [get], [set]
```

The minimum width variation that triggers a document reflow. If the control is resized, but the width changes by less than this amount, the document is not re-drawn.

Definition at line 73 of file MarkdownCanvas.axaml.cs.

7.61.4.6 Renderer

```
MarkdownRenderer VectSharp.MarkdownCanvas.MarkdownCanvasControl.Renderer [get]
```

The MarkdownRenderer used to render the Document. You can use the properties of this object to customise the rendering. Note that setting the Avalonia.Controls.Primitives.TemplatedControl.FontSize of the MarkdownCanvasControl will propagate to the Renderer's MarkdownRenderer.BaseFontSize.

Definition at line 124 of file MarkdownCanvas.axaml.cs.

7.61.4.7 TextConversionOption

```
AvaloniaContextInterpreter.TextOptions VectSharp.MarkdownCanvas.MarkdownCanvasControl.Text↩
ConversionOption [get], [set]
```

Gets or sets the value that determines whether text items should be converted into paths when drawing. Setting this to AvaloniaContextInterpreter.TextOptions.NeverConvert will improve performance if you are using custom fonts, but may cause unexpected results unless the font families being used are of type ResourceFontFamily.

Definition at line 115 of file MarkdownCanvas.axaml.cs.

The documentation for this class was generated from the following file:

- VectSharp.MarkdownCanvas/MarkdownCanvas.axaml.cs

7.62 VectSharp.Markdown.MarkdownRenderer Class Reference

Renders Markdown documents into VectSharp graphics objects.

Public Types

- enum VerticalAlignment

  Defines the options for the vertical alignment of table cells.

Public Member Functions

- Page RenderSinglePage (string markdownSource, double width, out Dictionary< string, string > link↩ Destinations)

  Parses the supplied markdownSource using all the supported extensions and renders the resulting document. Page breaks are disabled, and the document is rendered as a single page with the specified width . The page will be cropped at the appropriate height to contain the entire document.
- Page RenderSinglePage (MarkdownDocument markdownDocument, double width, out Dictionary< string, string > linkDestinations)

  Renders the supplied markdownDocument . Page breaks are disabled, and the document is rendered as a single page with the specified width . The page will be cropped at the appropriate height to contain the entire document.
- Document Render (string markdownSource, out Dictionary< string, string > linkDestinations)

  Parses the supplied markdownSource using all the supported extensions and renders the resulting document. The Document produced consists of one or more pages of the size specified in the PageSize of the current instance.
- Document Render (MarkdownDocument mardownDocument, out Dictionary< string, string > link↩ Destinations)

  Renders the supplied mardownDocument . The Document produced consists of one or more pages of the size specified in the PageSize of the current instance.

Properties

- double BaseFontSize = 9.71424 [get, set]

  The base font size to use when rendering the document. This will be the size of regular elements, and the size of header elements will be expressed as a multiple of this.
- double MathFontScalingFactor = 0.85 [get, set]

  Scaling factor for the font size to use when rendering math.
- double[ ] HeaderFontSizeMultipliers [get]

  The font size for elements at each header level. The values in this array will be multiplied by the BaseFontSize.
- double[ ] HeaderLineThicknesses = new double[ ] { 1, 1, 0, 0, 0, 0 } [get]

  The thickness of the separator line after a header of each level. A value of 0 disables the line after headers of that level.
- double ThematicBreakThickness = 2 [get, set]

  The thickness of thematic break lines.
- FontFamily RegularFontFamily = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica) [get, set]

  The font family for regular text.
- FontFamily BoldFontFamily = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica↩Bold) [get, set]

  The font family for bold text.
- FontFamily ItalicFontFamily = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica↩Oblique) [get, set]

  The font family for italic text.
- FontFamily BoldItalicFontFamily = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.↩HelveticaBoldOblique) [get, set]

  The font family for bold italic text.
- FontFamily CodeFont = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Courier) [get, set]

  The font family for code elements.
- FontFamily CodeFontBold = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBold) [get, set]

  The font family for bold code elements.
- FontFamily CodeFontItalic = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Courier↩Oblique) [get, set]

  The font family for italic code elements.
- FontFamily CodeFontBoldItalic = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Courier↩BoldOblique) [get, set]

  The font family for bold italic code elements.
- double UnderlineThickness = 0.075 [get, set]

  The thickness of underlines. This value will be multiplied by the font size of the element being underlined.
- double BoldUnderlineThickness = 0.15 [get, set]

  The thickness of underlines for bold text. This value will be multiplied by the font size of the element being underlined.
- Margins Margins = new Margins(55, 55, 55, 55) [get, set]

  The margins of the page.
- Margins TableCellMargins = new Margins(5, 0, 5, 0) [get, set]

  The margins for table cells.
- VerticalAlignment TableVAlign = VerticalAlignment.Middle [get, set]

  The vertical alignment of table cells.
- Size PageSize = new Size(595, 842) [get, set]

  The size of the page.
- double SpaceBeforeParagaph = 0 [get, set]

  The space before each text paragraph. This value will be multiplied by the BaseFontSize.

- double SpaceAfterParagraph = 0.75 [get, set]

  The space after each text paragraph. This value will be multiplied by the BaseFontSize.
- double SpaceAfterLine = 0.25 [get, set]

  The space after each line of text. This value will be multiplied by the BaseFontSize.
- double SpaceBeforeHeading = 0.25 [get, set]

  The space before each heading. This value will be multiplied by the font size of the heading.
- double SpaceAfterHeading = 0.25 [get, set]

  The space after each heading. This value will be multiplied by the font size of the heading.
- double CodeInlineMargin = 0.25 [get, set]

  The margin at the left and right of code inlines. This value will be multiplied by the current font size.
- double IndentWidth = 40 [get, set]

  The indentation width used for list items.
- double QuoteBlockIndentWidth = 30 [get, set]

  The indentation width used for block quotes.
- double QuoteBlockBarWidth = 5 [get, set]

  The thickness of the bar to the left of block quotes.
- double SubSuperscriptFontSize = 0.7 [get, set]

  The font size for subscripts and superscripts. This value will be multiplied by the current font size.
- double SuperscriptShift = 0.33 [get, set]

  The upwards shift in the baseline for superscript elements. This value will be multiplied by the current font size.
- double SubscriptShift = 0.14 [get, set]

  The downwards shift in the baseline for subscript elements. This value will be multiplied by the current font size.
- string BaseImageUri = "" [get, set]

  The base uri for resolving relative image addresses.
- Func< string, string,(string, bool)> ImageUriResolver = HTTPUtils.ResolveImageURI [get, set]

  A method used to resolve (possibly remote) image uris into local file paths. The first argument of the method should be the image uri and the second argument the base uri used to resolve relative links. The method should return a tuple containing the path of the local file and a boolean value indicating whether the file has been fetched from a remote location and should be deleted after the program has finished using it.
- Uri BaseLinkUri = new Uri("about:blank") [get, set]

  The base uri for resolving links.
- Func< string, string > LinkUriResolver = a => a [get, set]

  A method used to resolve link addresses. The argument of the method should be the absolute link, and the method should return the resolved address. This can be used to "redirect" links to a different target.
- Func< string, RasterImage > RasterImageLoader = null [get, set]

  A method used to a load raster image from a local file. The argument of the method should be the path of a local image file, and the method should return a RasterImage representing that file. For example, this can be achieved using the `RasterImageFile` class from the `VectSharp.MuPDFUtils` package. If this is `null`, only SVG images will be included in the document.
- double ImageUnitMultiplier = 0.60714 [get, set]

  The size of images (as defined in the image's width and height attributes) will be multiplied by this value to determine the actual size of the image on the page. This has no effect on images without a width or height attribute.
- double ImageMultiplier = 1 [get, set]

  The size of images will be multiplied by this value to determine the actual size of the image on the page. For images that have a width or height attribute, this will be applied in addition to the ImageUnitMultiplier. For images without width and height, only this multiplier will be applied.
- double ImageSideMargin = 10 [get, set]

  The margin on the right of left-aligned images and on the left of right-aligned images.
- double ImageMarginTolerance = 25 [get, set]

  Images will be allowed to extend into the page bottom margin area by this amount before triggering a page break. This should be smaller than the bottom margin, otherwise images risk being cut off by the page boundary.
- Func< string, string, List< List< FormattedString > > > SyntaxHighlighter = VectSharp.Markdown.SyntaxHighlighter.GetSynta
  [get, set]

A method used for syntax highlighting. The first argument should be the source code to highlight, while the second parameter is the name of the language to use for the highlight. The method should return a list of lists of FormattedStrings, with each list of FormattedStrings representing a line. For each code block, if the method returns `null`, no syntax highlighting is used.

- List< Action< Graphics, Colour > > Bullets `[get]`

  Bullet points used for unordered lists. Each element of this list corresponds to the bullet for each level of list indentation. If the list indentation is greater than the number of elements in this list, the bullet points will be reused cyclically. Each element of this list is a method taking two arguments: the first is the Graphics object on which the bullet point should be drawn, while the second is the colour in which it should be painted. The method should draw the bullet point centered around the origin. The size of the bullet point will be multiplied by the current font size.
- Colour ForegroundColour = Colours.Black `[get, set]`

  The foreground colour for text elements.
- Colour BackgroundColour = Colours.White `[get, set]`

  The background colour for the page.
- Colour HeaderLineColour = Colour.FromRgb(180, 180, 180) `[get, set]`

  The colour of the line below headers.
- Colour ThematicBreakLineColour = Colour.FromRgb(180, 180, 200) `[get, set]`

  The colour for thematic break lines.
- Colour LinkColour = Colour.FromRgb(25, 140, 191) `[get, set]`

  The colour for hypertext links-
- Colour CodeInlineBackgroundColour = Colour.FromRgb(240, 240, 240) `[get, set]`

  The background colour for code inlines.
- Colour CodeBlockBackgroundColour = Colour.FromRgb(240, 240, 245) `[get, set]`

  The background colour for code blocks.
- Colour QuoteBlockBarColour = Colour.FromRgb(75, 152, 220) `[get, set]`

  The colour for the bar to the left of block quotes.
- Colour QuoteBlockBackgroundColour = Colour.FromRgb(240, 240, 255) `[get, set]`

  The background colour for block quotes.
- Colour InsertedColour = Colour.FromRgb(0, 158, 115) `[get, set]`

  The colour for text that has been styled as "inserted".
- Colour MarkedColour = Colour.FromRgb(213, 94, 0) `[get, set]`

  The colour for text that has been styled as "marked".
- Colour TableHeaderRowSeparatorColour = Colours.Black `[get, set]`

  The colour for the line separating the table header row from normal rows.
- Colour TableRowSeparatorColour = Colour.FromRgb(180, 180, 180) `[get, set]`

  The colour for lines separating table rows from each other.
- double TableHeaderRowSeparatorThickness = 2 `[get, set]`

  The thickness of the line separating the table header row from normal rows.
- double TableHeaderSeparatorThickness = 1 `[get, set]`

  The thickness of lines separating table rows from each other.
- Graphics TaskListUncheckedBullet `[get, set]`

  The bullet used for unchecked task list items.
- Graphics TaskListCheckedBullet `[get, set]`

  The bullet used for checked task list items.
- bool AllowPageBreak = true `[get, set]`

  Determines whether page breaks should be treated as such in the source.

7.62.1 Detailed Description

Renders Markdown documents into VectSharp graphics objects.

Definition at line 37 of file MarkdownRenderer.cs.

7.62.2 Member Enumeration Documentation

7.62.2.1 VerticalAlignment

```
enum VectSharp.Markdown.MarkdownRenderer.VerticalAlignment
```

Defines the options for the vertical alignment of table cells.

Definition at line 130 of file MarkdownRenderer.cs.

7.62.3 Member Function Documentation

7.62.3.1 Render() [1/2]

```
Document VectSharp.Markdown.MarkdownRenderer.Render (
            MarkdownDocument mardownDocument,
            out Dictionary< string, string > linkDestinations )
```

Renders the supplied *mardownDocument* . The Document produced consists of one or more pages of the size specified in the PageSize of the current instance.

Parameters

mardownDocument	The markdown document to render.
linkDestinations	When this method returns, this value will contain a dictionary used to associate graphic action tags to hyperlinks. This can be used to enable such links when rendering the Document to a file.

Returns

A Document containing a rendering of the supplied markdown document, consisting of one or more pages of the size specified in the PageSize of the current instance.

Definition at line 504 of file MarkdownRenderer.cs.

7.62.3.2 Render() [2/2]

```
Document VectSharp.Markdown.MarkdownRenderer.Render (
            string markdownSource,
            out Dictionary< string, string > linkDestinations )
```

Parses the supplied *markdownSource* using all the supported extensions and renders the resulting document. The Document produced consists of one or more pages of the size specified in the PageSize of the current instance.

Parameters

markdownSource	The markdown source to parse.
linkDestinations	When this method returns, this value will contain a dictionary used to associate graphic action tags to hyperlinks. This can be used to enable such links when rendering the Document to a file.

Returns

A Document containing a rendering of the supplied markdown document, consisting of one or more pages of the size specified in the PageSize of the current instance.

Definition at line 491 of file MarkdownRenderer.cs.

7.62.3.3 RenderSinglePage() [1/2]

```
Page VectSharp.Markdown.MarkdownRenderer.RenderSinglePage (
            MarkdownDocument markdownDocument,
            double width,
            out Dictionary< string, string > linkDestinations )
```

Renders the supplied *markdownDocument* . Page breaks are disabled, and the document is rendered as a single page with the specified *width* . The page will be cropped at the appropriate height to contain the entire document.

Parameters

markdownDocument	The markdown document to render.
width	The width of the page.
linkDestinations	When this method returns, this value will contain a dictionary used to associate graphic action tags to hyperlinks. This can be used to enable such links when rendering the Page to a file.

Returns

A Page containing a rendering of the supplied markdown document.

Definition at line 432 of file MarkdownRenderer.cs.

7.62.3.4 RenderSinglePage() [2/2]

```
Page VectSharp.Markdown.MarkdownRenderer.RenderSinglePage (
            string markdownSource,
            double width,
            out Dictionary< string, string > linkDestinations )
```

Parses the supplied *markdownSource* using all the supported extensions and renders the resulting document. Page breaks are disabled, and the document is rendered as a single page with the specified *width* . The page will be cropped at the appropriate height to contain the entire document.

Parameters

markdownSource	The markdown source to parse.
width	The width of the page.
linkDestinations	When this method returns, this value will contain a dictionary used to associate graphic action tags to hyperlinks. This can be used to enable such links when rendering the Page to a file.

Returns

A Page containing a rendering of the supplied markdown document.

Definition at line 418 of file MarkdownRenderer.cs.

7.62.4 Property Documentation

7.62.4.1 AllowPageBreak

```
bool VectSharp.Markdown.MarkdownRenderer.AllowPageBreak = true  [get], [set]
```

Determines whether page breaks should be treated as such in the source.

Definition at line 402 of file MarkdownRenderer.cs.

7.62.4.2 BackgroundColour

```
Colour VectSharp.Markdown.MarkdownRenderer.BackgroundColour = Colours.White  [get], [set]
```

The background colour for the page.

Definition at line 298 of file MarkdownRenderer.cs.

7.62.4.3 BaseFontSize

```
double VectSharp.Markdown.MarkdownRenderer.BaseFontSize = 9.71424  [get], [set]
```

The base font size to use when rendering the document. This will be the size of regular elements, and the size of header elements will be expressed as a multiple of this.

Definition at line 42 of file MarkdownRenderer.cs.

7.62.4.4 BaseImageUri

```
string VectSharp.Markdown.MarkdownRenderer.BaseImageUri = ""  [get], [set]
```

The base uri for resolving relative image addresses.

Definition at line 221 of file MarkdownRenderer.cs.

7.62.4.5 BaseLinkUri

```
Uri VectSharp.Markdown.MarkdownRenderer.BaseLinkUri = new Uri("about:blank")  [get], [set]
```

The base uri for resolving links.

Definition at line 231 of file MarkdownRenderer.cs.

7.62.4.6 BoldFontFamily

```
FontFamily VectSharp.Markdown.MarkdownRenderer.BoldFontFamily = FontFamily.ResolveFontFamily(Font↩
Family.StandardFontFamilies.HelveticaBold)  [get], [set]
```

The font family for bold text.

Definition at line 75 of file MarkdownRenderer.cs.

7.62.4.7 BoldItalicFontFamily

```
FontFamily VectSharp.Markdown.MarkdownRenderer.BoldItalicFontFamily = FontFamily.ResolveFontFamily(Font↩
Family.StandardFontFamilies.HelveticaBoldOblique)  [get], [set]
```

The font family for bold italic text.

Definition at line 85 of file MarkdownRenderer.cs.

7.62.4.8 BoldUnderlineThickness

```
double VectSharp.Markdown.MarkdownRenderer.BoldUnderlineThickness = 0.15  [get], [set]
```

The thickness of underlines for bold text. This value will be multiplied by the font size of the element being underlined.

Definition at line 115 of file MarkdownRenderer.cs.

7.62.4.9 Bullets

```
List<Action<Graphics, Colour> > VectSharp.Markdown.MarkdownRenderer.Bullets  [get]
```

Initial value:

```
= new List<Action<Graphics, Colour»()
        {
            (graphics, colour) =>
            {
                graphics.FillPath(new GraphicsPath().Arc(-0.5, 0, 0.25, 0, 2 * Math.PI), colour);
            },
            (graphics, colour) =>
            {
                graphics.StrokePath(new GraphicsPath().Arc(-0.5, 0, 0.25, 0, 2 * Math.PI), colour, 0.1);
            },
            (graphics, colour) =>
            {
                graphics.StrokeRectangle(-0.75, -0.25, 0.5, 0.5, colour, 0.1);
            },
        }
```

Bullet points used for unordered lists. Each element of this list corresponds to the bullet for each level of list indentation. If the list indentation is greater than the number of elements in this list, the bullet points will be reused cyclically. Each element of this list is a method taking two arguments: the first is the Graphics object on which the bullet point should be drawn, while the second is the colour in which it should be painted. The method should draw the bullet point centered around the origin. The size of the bullet point will be multiplied by the current font size.

Definition at line 272 of file MarkdownRenderer.cs.

7.62.4.10 CodeBlockBackgroundColour

```
Colour VectSharp.Markdown.MarkdownRenderer.CodeBlockBackgroundColour = Colour.FromRgb(240,
240, 245)  [get], [set]
```

The background colour for code blocks.

Definition at line 323 of file MarkdownRenderer.cs.

7.62.4.11 CodeFont

```
FontFamily VectSharp.Markdown.MarkdownRenderer.CodeFont = FontFamily.ResolveFontFamily(Font↩
Family.StandardFontFamilies.Courier)  [get], [set]
```

The font family for code elements.

Definition at line 90 of file MarkdownRenderer.cs.

7.62.4.12 CodeFontBold

```
FontFamily VectSharp.Markdown.MarkdownRenderer.CodeFontBold = FontFamily.ResolveFontFamily(Font↩
Family.StandardFontFamilies.CourierBold)  [get], [set]
```

The font family for bold code elements.

Definition at line 95 of file MarkdownRenderer.cs.

7.62.4.13 CodeFontBoldItalic

```
FontFamily VectSharp.Markdown.MarkdownRenderer.CodeFontBoldItalic = FontFamily.ResolveFontFamily(Font↩
Family.StandardFontFamilies.CourierBoldOblique)  [get], [set]
```

The font family for bold italic code elements.

Definition at line 105 of file MarkdownRenderer.cs.

7.62.4.14 CodeFontItalic

```
FontFamily VectSharp.Markdown.MarkdownRenderer.CodeFontItalic = FontFamily.ResolveFontFamily(Font↩
Family.StandardFontFamilies.CourierOblique)  [get], [set]
```

The font family for italic code elements.

Definition at line 100 of file MarkdownRenderer.cs.

7.62.4.15 CodeInlineBackgroundColour

```
Colour VectSharp.Markdown.MarkdownRenderer.CodeInlineBackgroundColour = Colour.FromRgb(240,
240, 240)  [get], [set]
```

The background colour for code inlines.

Definition at line 318 of file MarkdownRenderer.cs.

7.62.4.16 CodeInlineMargin

```
double VectSharp.Markdown.MarkdownRenderer.CodeInlineMargin = 0.25  [get], [set]
```

The margin at the left and right of code inlines. This value will be multiplied by the current font size.

Definition at line 186 of file MarkdownRenderer.cs.

7.62.4.17 ForegroundColour

```
Colour VectSharp.Markdown.MarkdownRenderer.ForegroundColour = Colours.Black  [get], [set]
```

The foreground colour for text elements.

Definition at line 293 of file MarkdownRenderer.cs.

7.62.4.18 HeaderFontSizeMultipliers

```
double [] VectSharp.Markdown.MarkdownRenderer.HeaderFontSizeMultipliers  [get]
```

Initial value:

```
= new double[]
        {
            28 / 12.0, 22 / 12.0, 16 / 12.0, 14 / 12.0, 13 / 12.0, 12 / 12.0
        }
```

The font size for elements at each header level. The values in this array will be multiplied by the BaseFontSize.

Definition at line 52 of file MarkdownRenderer.cs.

7.62.4.19 HeaderLineColour

```
Colour VectSharp.Markdown.MarkdownRenderer.HeaderLineColour = Colour.FromRgb(180, 180, 180)
[get], [set]
```

The colour of the line below headers.

Definition at line 303 of file MarkdownRenderer.cs.

7.62.4.20 HeaderLineThicknesses

```
double [] VectSharp.Markdown.MarkdownRenderer.HeaderLineThicknesses = new double[] { 1, 1, 0,
0, 0, 0 }  [get]
```

The thickness of the separator line after a header of each level. A value of 0 disables the line after headers of that level.

Definition at line 60 of file MarkdownRenderer.cs.

7.62.4.21 ImageMarginTolerance

```
double VectSharp.Markdown.MarkdownRenderer.ImageMarginTolerance = 25  [get], [set]
```

Images will be allowed to extend into the page bottom margin area by this amount before triggering a page break. This should be smaller than the bottom margin, otherwise images risk being cut off by the page boundary.

Definition at line 261 of file MarkdownRenderer.cs.

7.62.4.22 ImageMultiplier

```
double VectSharp.Markdown.MarkdownRenderer.ImageMultiplier = 1  [get], [set]
```

The size of images will be multiplied by this value to determine the actual size of the image on the page. For images that have a width or height attribute, this will be applied in addition to the ImageUnitMultiplier. For images without width and height, only this multiplier will be applied.

Definition at line 251 of file MarkdownRenderer.cs.

7.62.4.23 ImageSideMargin

```
double VectSharp.Markdown.MarkdownRenderer.ImageSideMargin = 10  [get], [set]
```

The margin on the right of left-aligned images and on the left of right-aligned images.

Definition at line 256 of file MarkdownRenderer.cs.

7.62.4.24 ImageUnitMultiplier

```
double VectSharp.Markdown.MarkdownRenderer.ImageUnitMultiplier = 0.60714  [get], [set]
```

The size of images (as defined in the image's width and height attributes) will be multiplied by this value to determine the actual size of the image on the page. This has no effect on images without a width or height attribute.

Definition at line 246 of file MarkdownRenderer.cs.

7.62.4.25 ImageUriResolver

```
Func<string, string, (string, bool)> VectSharp.Markdown.MarkdownRenderer.ImageUriResolver =
HTTPUtils.ResolveImageURI  [get], [set]
```

A method used to resolve (possibly remote) image uris into local file paths. The first argument of the method should be the image uri and the second argument the base uri used to resolve relative links. The method should return a tuple containing the path of the local file and a boolean value indicating whether the file has been fetched from a remote location and should be deleted after the program has finished using it.

Definition at line 226 of file MarkdownRenderer.cs.

7.62.4.26 IndentWidth

```
double VectSharp.Markdown.MarkdownRenderer.IndentWidth = 40  [get], [set]
```

The indentation width used for list items.

Definition at line 191 of file MarkdownRenderer.cs.

7.62.4.27 InsertedColour

```
Colour VectSharp.Markdown.MarkdownRenderer.InsertedColour = Colour.FromRgb(0, 158, 115)  [get],
[set]
```

The colour for text that has been styled as "inserted".

Definition at line 338 of file MarkdownRenderer.cs.

7.62.4.28 ItalicFontFamily

```
FontFamily VectSharp.Markdown.MarkdownRenderer.ItalicFontFamily = FontFamily.ResolveFontFamily(Font↩
Family.StandardFontFamilies.HelveticaOblique)  [get], [set]
```

The font family for italic text.

Definition at line 80 of file MarkdownRenderer.cs.

7.62.4.29 LinkColour

```
Colour VectSharp.Markdown.MarkdownRenderer.LinkColour = Colour.FromRgb(25, 140, 191)  [get],
[set]
```

The colour for hypertext links-

Definition at line 313 of file MarkdownRenderer.cs.

7.62.4.30 LinkUriResolver

```
Func<string, string> VectSharp.Markdown.MarkdownRenderer.LinkUriResolver = a => a  [get],
[set]
```

A method used to resolve link addresses. The argument of the method should be the absolute link, and the method should return the resolved address. This can be used to "redirect" links to a different target.

Definition at line 236 of file MarkdownRenderer.cs.

7.62.4.31 Margins

```
Margins VectSharp.Markdown.MarkdownRenderer.Margins = new Margins(55, 55, 55, 55)  [get], [set]
```

The margins of the page.

Definition at line 120 of file MarkdownRenderer.cs.

7.62.4.32 MarkedColour

```
Colour VectSharp.Markdown.MarkdownRenderer.MarkedColour = Colour.FromRgb(213, 94, 0)  [get],
[set]
```

The colour for text that has been styled as "marked".

Definition at line 343 of file MarkdownRenderer.cs.

7.62.4.33 MathFontScalingFactor

```
double VectSharp.Markdown.MarkdownRenderer.MathFontScalingFactor = 0.85  [get], [set]
```

Scaling factor for the font size to use when rendering math.

Definition at line 47 of file MarkdownRenderer.cs.

7.62.4.34 PageSize

```
Size VectSharp.Markdown.MarkdownRenderer.PageSize = new Size(595, 842)  [get], [set]
```

The size of the page.

Definition at line 156 of file MarkdownRenderer.cs.

7.62.4.35 QuoteBlockBackgroundColour

```
Colour VectSharp.Markdown.MarkdownRenderer.QuoteBlockBackgroundColour = Colour.FromRgb(240,
240, 255)  [get], [set]
```

The background colour for block quotes.

Definition at line 333 of file MarkdownRenderer.cs.

7.62.4.36 QuoteBlockBarColour

```
Colour VectSharp.Markdown.MarkdownRenderer.QuoteBlockBarColour = Colour.FromRgb(75, 152, 220)
[get], [set]
```

The colour for the bar to the left of block quotes.

Definition at line 328 of file MarkdownRenderer.cs.

7.62.4.37 QuoteBlockBarWidth

```
double VectSharp.Markdown.MarkdownRenderer.QuoteBlockBarWidth = 5  [get], [set]
```

The thickness of the bar to the left of block quotes.

Definition at line 201 of file MarkdownRenderer.cs.

7.62.4.38 QuoteBlockIndentWidth

```
double VectSharp.Markdown.MarkdownRenderer.QuoteBlockIndentWidth = 30  [get], [set]
```

The indentation width used for block quotes.

Definition at line 196 of file MarkdownRenderer.cs.

7.62.4.39 RasterImageLoader

```
Func<string, RasterImage> VectSharp.Markdown.MarkdownRenderer.RasterImageLoader = null  [get],
[set]
```

A method used to a load raster image from a local file. The argument of the method should be the path of a local image file, and the method should return a RasterImage representing that file. For example, this can be achieved using the `RasterImageFile` class from the `VectSharp.MuPDFUtils` package. If this is `null`, only SVG images will be included in the document.

Definition at line 241 of file MarkdownRenderer.cs.

7.62.4.40 RegularFontFamily

```
FontFamily VectSharp.Markdown.MarkdownRenderer.RegularFontFamily = FontFamily.ResolveFontFamily(Font↩
Family.StandardFontFamilies.Helvetica)  [get], [set]
```

The font family for regular text.

Definition at line 70 of file MarkdownRenderer.cs.

7.62.4.41 SpaceAfterHeading

```
double VectSharp.Markdown.MarkdownRenderer.SpaceAfterHeading = 0.25  [get], [set]
```

The space after each heading. This value will be multiplied by the font size of the heading.

Definition at line 181 of file MarkdownRenderer.cs.

7.62.4.42 SpaceAfterLine

```
double VectSharp.Markdown.MarkdownRenderer.SpaceAfterLine = 0.25  [get], [set]
```

The space after each line of text. This value will be multiplied by the BaseFontSize.

Definition at line 171 of file MarkdownRenderer.cs.

7.62.4.43 SpaceAfterParagraph

```
double VectSharp.Markdown.MarkdownRenderer.SpaceAfterParagraph = 0.75  [get], [set]
```

The space after each text paragraph. This value will be multiplied by the BaseFontSize.

Definition at line 166 of file MarkdownRenderer.cs.

7.62.4.44 SpaceBeforeHeading

```
double VectSharp.Markdown.MarkdownRenderer.SpaceBeforeHeading = 0.25  [get], [set]
```

The space before each heading. This value will be multiplied by the font size of the heading.

Definition at line 176 of file MarkdownRenderer.cs.

7.62.4.45 SpaceBeforeParagaph

```
double VectSharp.Markdown.MarkdownRenderer.SpaceBeforeParagaph = 0  [get], [set]
```

The space before each text paragraph. This value will be multiplied by the BaseFontSize.

Definition at line 161 of file MarkdownRenderer.cs.

7.62.4.46 SubscriptShift

```
double VectSharp.Markdown.MarkdownRenderer.SubscriptShift = 0.14  [get], [set]
```

The downwards shift in the baseline for subscript elements. This value will be multiplied by the current font size.

Definition at line 216 of file MarkdownRenderer.cs.

7.62.4.47 SubSuperscriptFontSize

```
double VectSharp.Markdown.MarkdownRenderer.SubSuperscriptFontSize = 0.7  [get], [set]
```

The font size for subscripts and superscripts. This value will be multiplied by the current font size.

Definition at line 206 of file MarkdownRenderer.cs.

7.62.4.48 SuperscriptShift

```
double VectSharp.Markdown.MarkdownRenderer.SuperscriptShift = 0.33  [get], [set]
```

The upwards shift in the baseline for superscript elements. This value will be multiplied by the current font size.

Definition at line 211 of file MarkdownRenderer.cs.

7.62.4.49 SyntaxHighlighter

```
Func<string, string, List<List<FormattedString> > > VectSharp.Markdown.MarkdownRenderer.↩
SyntaxHighlighter = VectSharp.Markdown.SyntaxHighlighter.GetSyntaxHighlightedLines  [get],
[set]
```

A method used for syntax highlighting. The first argument should be the source code to highlight, while the second parameter is the name of the language to use for the highlight. The method should return a list of lists of FormattedStrings, with each list of FormattedStrings representing a line. For each code block, if the method returns `null`, no syntax highlighting is used.

Definition at line 266 of file MarkdownRenderer.cs.

7.62.4.50 TableCellMargins

```
Margins VectSharp.Markdown.MarkdownRenderer.TableCellMargins = new Margins(5, 0, 5, 0)  [get],
[set]
```

The margins for table cells.

Definition at line 125 of file MarkdownRenderer.cs.

7.62.4.51 TableHeaderRowSeparatorColour

```
Colour VectSharp.Markdown.MarkdownRenderer.TableHeaderRowSeparatorColour = Colours.Black  [get],
[set]
```

The colour for the line separating the table header row from normal rows.

Definition at line 348 of file MarkdownRenderer.cs.

7.62.4.52 TableHeaderRowSeparatorThickness

```
double VectSharp.Markdown.MarkdownRenderer.TableHeaderRowSeparatorThickness = 2  [get], [set]
```

The thickness of the line separating the table header row from normal rows.

Definition at line 358 of file MarkdownRenderer.cs.

7.62.4.53 TableHeaderSeparatorThickness

```
double VectSharp.Markdown.MarkdownRenderer.TableHeaderSeparatorThickness = 1  [get], [set]
```

The thickness of lines separating table rows from each other.

Definition at line 363 of file MarkdownRenderer.cs.

7.62.4.54 TableRowSeparatorColour

```
Colour VectSharp.Markdown.MarkdownRenderer.TableRowSeparatorColour = Colour.FromRgb(180, 180,
180)  [get], [set]
```

The colour for lines separating table rows from each other.

Definition at line 353 of file MarkdownRenderer.cs.

7.62.4.55 TableVAlign

```
VerticalAlignment VectSharp.Markdown.MarkdownRenderer.TableVAlign = VerticalAlignment.Middle
[get], [set]
```

The vertical alignment of table cells.

Definition at line 151 of file MarkdownRenderer.cs.

7.62.4.56 TaskListCheckedBullet

```
Graphics VectSharp.Markdown.MarkdownRenderer.TaskListCheckedBullet  [get], [set]
```

Initial value:

```
= new Func<Graphics>(() =>
        {
            Graphics tbr = new Graphics();
            GraphicsPath checkboxPath = new GraphicsPath().MoveTo(-0.7, -0.4).LineTo(-0.3, -0.4).Arc(-0.3,
        -0.2, 0.2, 3 * Math.PI / 2, 2 * Math.PI).LineTo(-0.1, 0.2).Arc(-0.3, 0.2, 0.2, 0, Math.PI /
        2).LineTo(-0.7, 0.4).Arc(-0.7, 0.2, 0.2, Math.PI / 2, Math.PI).LineTo(-0.9, -0.2).Arc(-0.7, -0.2,
        0.2, Math.PI, 3 * Math.PI / 2).Close();
            tbr.FillPath(checkboxPath, Colour.FromRgb(240, 246, 249));
            tbr.StrokePath(checkboxPath, Colour.FromRgb(0, 114, 178), 0.075);
            GraphicsPath tickpath = new GraphicsPath().MoveTo(-0.75, -0.1).LineTo(-0.5, 0.15).LineTo(-0.1,
        -0.4);
            tbr.StrokePath(new GraphicsPath().MoveTo(-0.5, 0.15).LineTo(-0.1, -0.4), Colour.FromRgb(240,
        246, 249), 0.3, LineCaps.Round);
            tbr.StrokePath(tickpath, Colour.FromRgb(0, 158, 115), 0.2, LineCaps.Round);
            return tbr;
        })()
```

The bullet used for checked task list items.

Definition at line 383 of file MarkdownRenderer.cs.

7.62.4.57 TaskListUncheckedBullet

```
Graphics VectSharp.Markdown.MarkdownRenderer.TaskListUncheckedBullet  [get], [set]
```

Initial value:

```
= new Func<Graphics>(() =>
        {
            Graphics tbr = new Graphics();
            GraphicsPath checkboxPath = new GraphicsPath().MoveTo(-0.7, -0.4).LineTo(-0.3, -0.4).Arc(-0.3,
        -0.2, 0.2, 3 * Math.PI / 2, 2 * Math.PI).LineTo(-0.1, 0.2).Arc(-0.3, 0.2, 0.2, 0, Math.PI /
        2).LineTo(-0.7, 0.4).Arc(-0.7, 0.2, 0.2, Math.PI / 2, Math.PI).LineTo(-0.9, -0.2).Arc(-0.7, -0.2,
        0.2, Math.PI, 3 * Math.PI / 2).Close();
            tbr.FillPath(checkboxPath, Colour.FromRgb(240, 246, 249));
            tbr.StrokePath(checkboxPath, Colour.FromRgb(0, 114, 178), 0.075);
            return tbr;
        })()
```

The bullet used for unchecked task list items.

Definition at line 368 of file MarkdownRenderer.cs.

7.62.4.58 ThematicBreakLineColour

```
Colour VectSharp.Markdown.MarkdownRenderer.ThematicBreakLineColour = Colour.FromRgb(180, 180,
200)  [get], [set]
```

The colour for thematic break lines.

Definition at line 308 of file MarkdownRenderer.cs.

7.62.4.59 ThematicBreakThickness

```
double VectSharp.Markdown.MarkdownRenderer.ThematicBreakThickness = 2  [get], [set]
```

The thickness of thematic break lines.

Definition at line 65 of file MarkdownRenderer.cs.

7.62.4.60 UnderlineThickness

```
double VectSharp.Markdown.MarkdownRenderer.UnderlineThickness = 0.075  [get], [set]
```

The thickness of underlines. This value will be multiplied by the font size of the element being underlined.

Definition at line 110 of file MarkdownRenderer.cs.

The documentation for this class was generated from the following file:

- VectSharp.Markdown/MarkdownRenderer.cs

7.63 VectSharp.ThreeD.MaskedLightSource Class Reference

Represents a point light source with a stencil in front of it.

Inheritance diagram for VectSharp.ThreeD.MaskedLightSource:

Public Member Functions

- MaskedLightSource (double intensity, Point3D position, NormalizedVector3D direction, double distance, GraphicsPath mask, double maskOrientation, double triangulationResolution)

  Creates a new MaskedLightSource by triangulating the specified GraphicsPath.
- MaskedLightSource (double intensity, Point3D position, NormalizedVector3D direction, double distance, IEnumerable< GraphicsPath > triangulatedMask, double maskOrientation)

  Creates a new MaskedLightSource using the specified triangulatedMask .
- LightIntensity GetLightAt (Point3D point)

  Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	*The Point3DElement at which the light intensity should be computed.*

Returns

- double GetObstruction (Point3D point, IEnumerable< Triangle3DElement > shadowingTriangles)

  Determines the amount of obstruction of the light that results at point due to the specified shadowingTriangles .

Parameters

point	*The Point3D at which the obstruction should be computed.*
shadowingTriangles	*A collection of Triangle3DElement casting shadows.*

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Properties

- bool CastsShadow = true `[get, set]`

  Determines whether the light casts a shadow or not.
- Point3D Position `[get]`

  The position of the light source.
- Point3D Origin `[get]`

  The projection of the Position on the mask plane along the light's Direction.
- NormalizedVector3D Direction `[get]`

  The direction of the light.
- double Distance `[get]`

  The distance between the light source and the mask plane.
- double Intensity `[get, set]`

  The base intensity of the light.
- double DistanceAttenuationExponent = 2 `[get, set]`

  An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.
- double AngleAttenuationExponent = 1 `[get, set]`

  An exponent determining how fast the light attenuates away from the light's axis. Set to 0 to disable angular attenuation.

7.63.1 Detailed Description

Represents a point light source with a stencil in front of it.

Definition at line 385 of file Lights.cs.

7.63.2 Constructor & Destructor Documentation

7.63.2.1 MaskedLightSource() [1/2]

```
VectSharp.ThreeD.MaskedLightSource.MaskedLightSource (
            double intensity,
            Point3D position,
            NormalizedVector3D direction,
            double distance,
            GraphicsPath mask,
            double maskOrientation,
            double triangulationResolution )
```

Creates a new MaskedLightSource by triangulating the specified GraphicsPath.

Parameters

intensity	The base intensity of the light.
position	The position of the light source.
direction	The direction of the light.
distance	The distance between the light source and the mask plane.
mask	A GraphicsPath representing the transparent part of the mask.
maskOrientation	An angle in radians determining the orientation of the 2D mask in the mask plane.
triangulationResolution	The resolution to use to triangulate the *mask* .

Definition at line 437 of file Lights.cs.

7.63.2.2 MaskedLightSource() [2/2]

```
VectSharp.ThreeD.MaskedLightSource.MaskedLightSource (
            double intensity,
            Point3D position,
            NormalizedVector3D direction,
            double distance,
            IEnumerable< GraphicsPath > triangulatedMask,
            double maskOrientation )
```

Creates a new MaskedLightSource using the specified *triangulatedMask* .

Parameters

intensity	The base intensity of the light.
position	The position of the light source.
direction	The direction of the light.
distance	The distance between the light source and the mask plane.
triangulatedMask	A collection of GraphicsPaths representing the transparent part of the mask. Each GraphicsPath should represent a single triangle.
maskOrientation	An angle in radians determining the orientation of the 2D mask in the mask plane.

Definition at line 451 of file Lights.cs.

7.63.3 Member Function Documentation

7.63.3.1 GetLightAt()

```
LightIntensity VectSharp.ThreeD.MaskedLightSource.GetLightAt (
            Point3D point )
```

Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	The Point3DElement at which the light intensity should be computed.

Returns

Implements VectSharp.ThreeD.ILightSource.

Definition at line 483 of file Lights.cs.

7.63.3.2 GetObstruction()

```
double VectSharp.ThreeD.MaskedLightSource.GetObstruction (
            Point3D point,
            IEnumerable< Triangle3DElement > shadowingTriangles )
```

Determines the amount of obstruction of the light that results at *point* due to the specified *shadowingTriangles* .

Parameters

point	The Point3D at which the obstruction should be computed.
shadowingTriangles	A collection of Triangle3DElement casting shadows.

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 556 of file Lights.cs.

7.63.4 Property Documentation

7.63.4.1 AngleAttenuationExponent

```
double VectSharp.ThreeD.MaskedLightSource.AngleAttenuationExponent = 1  [get], [set]
```

An exponent determining how fast the light attenuates away from the light's axis. Set to 0 to disable angular attenuation.

Definition at line 425 of file Lights.cs.

7.63.4.2 CastsShadow

```
bool VectSharp.ThreeD.MaskedLightSource.CastsShadow = true  [get], [set]
```

Determines whether the light casts a shadow or not.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 388 of file Lights.cs.

7.63.4.3 Direction

```
NormalizedVector3D VectSharp.ThreeD.MaskedLightSource.Direction  [get]
```

The direction of the light.

Definition at line 403 of file Lights.cs.

7.63.4.4 Distance

```
double VectSharp.ThreeD.MaskedLightSource.Distance  [get]
```

The distance between the light source and the mask plane.

Definition at line 408 of file Lights.cs.

7.63.4.5 DistanceAttenuationExponent

```
double VectSharp.ThreeD.MaskedLightSource.DistanceAttenuationExponent = 2  [get], [set]
```

An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.

Definition at line 420 of file Lights.cs.

7.63.4.6 Intensity

```
double VectSharp.ThreeD.MaskedLightSource.Intensity  [get], [set]
```

The base intensity of the light.

Definition at line 415 of file Lights.cs.

7.63.4.7 Origin

```
Point3D VectSharp.ThreeD.MaskedLightSource.Origin  [get]
```

The projection of the Position on the mask plane along the light's Direction.

Definition at line 398 of file Lights.cs.

7.63.4.8 Position

```
Point3D VectSharp.ThreeD.MaskedLightSource.Position  [get]
```

The position of the light source.

Definition at line 393 of file Lights.cs.

The documentation for this class was generated from the following file:

- VectSharp.ThreeD/Lights.cs

7.64 VectSharp.Filters.MaskFilter Class Reference

Represents a filter that uses the luminance of an image to mask another image.

Inheritance diagram for VectSharp.Filters.MaskFilter:

Public Member Functions

- MaskFilter (Graphics mask)

  Creates a new MaskFilter with the specified mask image.
- RasterImage Filter (RasterImage image, Rectangle bounds, double scale)

  Applies the filter to a RasterImage.

Parameters

image	*The RasterImage to which the filter will be applied.*
bounds	*The region on the graphics surface where the image will be drawn.*
scale	*The scale of the image with respect to the filter.*

Returns

A new RasterImage containing the filtered image. The source image is left unaltered.

Properties

- Point TopLeftMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.
- Point BottomRightMargin `[get]`

  Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.
- Graphics Mask `[get]`

  The image that is used to mask the input image.

7.64.1 Detailed Description

Represents a filter that uses the luminance of an image to mask another image.

Definition at line 26 of file MaskFilter.cs.

7.64.2 Constructor & Destructor Documentation

7.64.2.1 MaskFilter()

```
VectSharp.Filters.MaskFilter.MaskFilter (
            Graphics mask )
```

Creates a new MaskFilter with the specified mask image.

Parameters

mask	The image that is used to mask the input image.

Definition at line 42 of file MaskFilter.cs.

7.64.3 Member Function Documentation

7.64.3.1 Filter()

```
RasterImage VectSharp.Filters.MaskFilter.Filter (
            RasterImage image,
            Rectangle bounds,
            double scale )
```

Applies the filter to a RasterImage.

Parameters

image	The RasterImage to which the filter will be applied.
bounds	The region on the graphics surface where the image will be drawn.
scale	The scale of the image with respect to the filter.

Returns

A new RasterImage containing the filtered image. The source *image* is left unaltered.

Implements VectSharp.Filters.IFilterWithLocation.

Definition at line 45 of file MaskFilter.cs.

7.64.4 Property Documentation

7.64.4.1 BottomRightMargin

```
Point VectSharp.Filters.MaskFilter.BottomRightMargin [get]
```

Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 31 of file MaskFilter.cs.

7.64.4.2 Mask

```
Graphics VectSharp.Filters.MaskFilter.Mask [get]
```

The image that is used to mask the input image.

Definition at line 36 of file MaskFilter.cs.

7.64.4.3 TopLeftMargin

```
Point VectSharp.Filters.MaskFilter.TopLeftMargin [get]
```

Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.

Implements VectSharp.Filters.IFilter.

Definition at line 29 of file MaskFilter.cs.

The documentation for this class was generated from the following file:

- VectSharp/Filters/MaskFilter.cs

7.65 VectSharp.Fonts.Nimbus Class Reference

Contains an IFontLibrary providing access to the Nimbus family of standard fonts (used e.g. by MuPDF).

Properties

- static IFontLibrary Library [get]
  The font library.

7.65.1 Detailed Description

Contains an IFontLibrary providing access to the Nimbus family of standard fonts (used e.g. by MuPDF).

Definition at line 21 of file Nimbus.cs.

7.65.2 Property Documentation

7.65.2.1 Library

```
IFontLibrary VectSharp.Fonts.Nimbus.Library [static], [get]
```

The font library.

Definition at line 26 of file Nimbus.cs.

The documentation for this class was generated from the following file:

- VectSharp.Fonts.Nimbus/Nimbus.cs

7.66 VectSharp.ThreeD.ObjectFactory Class Reference

A static class containing methods to create complex 3D objects.

Static Public Member Functions

- static List< Element3D > CreateCube (Point3D center, double size, IEnumerable< IMaterial > fill, string tag=null, int zIndex=0)

  Creates a cube.
- static List< Element3D > CreateCuboid (Point3D center, double sizeX, double sizeY, double sizeZ, IEnumerable< IMaterial > fill, string tag=null, int zIndex=0)

  Creates a cuboid.
- static List< Element3D > CreateRectangle (Point3D point1, Point3D point2, Point3D point3, Point3D point4, IEnumerable< IMaterial > fill, string tag=null, int zIndex=0)

  Creates a quadrilater. All the vertices need not be coplanar.
- static List< Element3D > CreateRectangle (Point3D point1, Point3D point2, Point3D point3, Point3D point4, NormalizedVector3D point1Normal, NormalizedVector3D point2Normal, NormalizedVector3D point3Normal, NormalizedVector3D point4Normal, IEnumerable< IMaterial > fill, string tag=null, int zIndex=0)

  Creates a quadrilater, specifying the vertex normals at the four vertices. All the vertices need not be coplanar.
- static List< Element3D > CreateSphere (Point3D center, double radius, int steps, IEnumerable< IMaterial > fill, string tag=null, int zIndex=0)

  Creates a sphere.
- static List< Element3D > CreateTetrahedron (Point3D center, double radius, IEnumerable< IMaterial > fill, string tag=null, int zIndex=0)

  Creates a tetrahedron inscribed in a sphere.
- static List< Element3D > CreatePolygon (GraphicsPath polygon2D, double triangulationResolution, Point3D origin, NormalizedVector3D xAxis, NormalizedVector3D yAxis, bool reverseTriangles, IEnumerable< IMaterial > fill, string tag=null, int zIndex=0)

  Creates a flat polygon.
- static List< Element3D > CreatePrism (GraphicsPath polygonBase2D, double triangulationResolution, Point3D bottomOrigin, Point3D topOrigin, NormalizedVector3D baseXAxis, NormalizedVector3D baseYAxis, IEnumerable< IMaterial > fill, string tag=null, int zIndex=0)

  Creates a prism with the specified base.
- static List< Element3D > CreateWireframe (IEnumerable< Element3D > object3D, Colour colour, double thickness=1, LineCaps lineCap=LineCaps.Butt, LineDash? lineDash=null, string tag=null, int zIndex=0)

  Creates a wireframe from a collection of Element3Ds.
- static List< Element3D > CreatePoints (IEnumerable< Element3D > object3D, Colour colour, double diameter=1, string tag=null, int zIndex=0)

  Obtains a list of Point3DElement corresponding to the vertices of a list of Element3Ds.

7.66.1 Detailed Description

A static class containing methods to create complex 3D objects.

Definition at line 28 of file ObjectFactory.cs.

7.66.2 Member Function Documentation

7.66.2.1 CreateCube()

```
static List< Element3D > VectSharp.ThreeD.ObjectFactory.CreateCube (
            Point3D center,
            double size,
            IEnumerable< IMaterial > fill,
            string tag = null,
            int zIndex = 0 )  [static]
```

Creates a cube.

Parameters

center	The centre of the cube.
size	The length of each side of the cube.
fill	A collection of materials that will be applied to the Triangle3DElements returned by this method.
tag	A tag that will be applied to the Triangle3DElements returned by this method.
zIndex	A z-index that will be applied to the Triangle3DElements returned by this method.

Returns

A list of Triangle3DElements that constitute the cube.

Definition at line 39 of file ObjectFactory.cs.

7.66.2.2 CreateCuboid()

```
static List< Element3D > VectSharp.ThreeD.ObjectFactory.CreateCuboid (
            Point3D center,
            double sizeX,
            double sizeY,
            double sizeZ,
            IEnumerable< IMaterial > fill,
            string tag = null,
            int zIndex = 0 )  [static]
```

Creates a cuboid.

Parameters

center	The centre of the cube.
sizeX	The length of the sides of the cube parallel to the x axis.
sizeY	The length of the sides of the cube parallel to the y axis.
sizeZ	The length of the sides of the cube parallel to the z axis.
fill	A collection of materials that will be applied to the Triangle3DElements returned by this method.
tag	A tag that will be applied to the Triangle3DElements returned by this method.
zIndex	A z-index that will be applied to the Triangle3DElements returned by this method.

Returns

A list of Triangle3DElements that constitute the cuboid.

Definition at line 55 of file ObjectFactory.cs.

7.66.2.3 CreatePoints()

```
static List< Element3D > VectSharp.ThreeD.ObjectFactory.CreatePoints (
            IEnumerable< Element3D > object3D,
            Colour colour,
            double diameter = 1,
            string tag = null,
            int zIndex = 0 )  [static]
```

Obtains a list of Point3DElement corresponding to the vertices of a list of Element3Ds.

Parameters

object3D	The collection of Element3Ds. Point3DElements are ignored.
colour	The colour of the Point3DElements returned by this method.
diameter	The diameter of the Point3DElements returned by this method.
tag	A tag that will be applied to the Point3DElements returned by this method.
zIndex	A z-index that will be applied to the Point3DElements returned by this method.

Returns

A list of Point3DElements corresponding to the vertices of the Element3Ds.

Definition at line 412 of file ObjectFactory.cs.

7.66.2.4 CreatePolygon()

```
static List< Element3D > VectSharp.ThreeD.ObjectFactory.CreatePolygon (
            GraphicsPath polygon2D,
            double triangulationResolution,
            Point3D origin,
            NormalizedVector3D xAxis,
            NormalizedVector3D yAxis,
            bool reverseTriangles,
            IEnumerable< IMaterial > fill,
            string tag = null,
            int zIndex = 0 )  [static]
```

Creates a flat polygon.

Parameters

polygon2D	A 2D GraphicsPath representing the polygon.
triangulationResolution	The resolution that will be used to linearise curve segments in the GraphicsPath.
origin	A Point3D that will correspond to the origin of the 2D reference system.
xAxis	A NormalizedVector3D that will correspond to the x axis of the 2D reference system. This will be orthonormalised to the *yAxis* .
yAxis	A NormalizedVector3D that will correspond to the y axis of the 2D reference system.
reverseTriangles	Indicates whether the order of the points (and thus the normals) of all the triangles returned by this method should be reversed.
fill	A collection of materials that will be applied to the Triangle3DElements returned by this method.
tag	A tag that will be applied to the Triangle3DElements returned by this method.
zIndex	A z-index that will be applied to the Triangle3DElements returned by this method.

Returns

A list of Triangle3DElements that constitute the polygon.

Definition at line 273 of file ObjectFactory.cs.

7.66.2.5 CreatePrism()

```
static List< Element3D > VectSharp.ThreeD.ObjectFactory.CreatePrism (
            GraphicsPath polygonBase2D,
            double triangulationResolution,
            Point3D bottomOrigin,
            Point3D topOrigin,
            NormalizedVector3D baseXAxis,
            NormalizedVector3D baseYAxis,
            IEnumerable< IMaterial > fill,
            string tag = null,
            int zIndex = 0 )  [static]
```

Creates a prism with the specified base.

Parameters

polygonBase2D	A 2D GraphicsPath representing the base of the prism.
triangulationResolution	The resolution that will be used to linearise curve segments in the GraphicsPath.
bottomOrigin	A Point3D that will correspond to the origin of the 2D reference system of the bottom base.
topOrigin	A Point3D that will correspond to the origin of the 2D reference system of the top base.
baseXAxis	A NormalizedVector3D that will correspond to the x axis of the 2D reference system of the bases. This will be orthonormalised to the *baseYAxis* .
baseYAxis	A NormalizedVector3D that will correspond to the y axis of the 2D reference system of the bases.
fill	A collection of materials that will be applied to the Triangle3DElements returned by this method.
tag	A tag that will be applied to the Triangle3DElements returned by this method.
zIndex	A z-index that will be applied to the Triangle3DElements returned by this method.

Returns

A list of Triangle3DElements that constitute the prism.

Definition at line 314 of file ObjectFactory.cs.

7.66.2.6 CreateRectangle() [1/2]

```
static List< Element3D > VectSharp.ThreeD.ObjectFactory.CreateRectangle (
            Point3D point1,
            Point3D point2,
            Point3D point3,
            Point3D point4,
            IEnumerable< IMaterial > fill,
            string tag = null,
            int zIndex = 0 )  [static]
```

Creates a quadrilater. All the vertices need not be coplanar.

Parameters

point1	The first vertex of the quadrilater.
point2	The second vertex of the quadrilater.
point3	The third vertex of the quadrilater.
point4	The fourth vertex of the quadrilater.
fill	A collection of materials that will be applied to the Triangle3DElements returned by this method.
tag	A tag that will be applied to the Triangle3DElements returned by this method.
zIndex	A z-index that will be applied to the Triangle3DElements returned by this method.

Returns

A list containing two Triangle3DElements representing the quadrilater.

Definition at line 93 of file ObjectFactory.cs.

7.66.2.7 CreateRectangle() [2/2]

```
static List< Element3D > VectSharp.ThreeD.ObjectFactory.CreateRectangle (
            Point3D point1,
            Point3D point2,
            Point3D point3,
            Point3D point4,
            NormalizedVector3D point1Normal,
            NormalizedVector3D point2Normal,
            NormalizedVector3D point3Normal,
            NormalizedVector3D point4Normal,
            IEnumerable< IMaterial > fill,
            string tag = null,
            int zIndex = 0 )  [static]
```

Creates a quadrilater, specifying the vertex normals at the four vertices. All the vertices need not be coplanar.

Parameters

point1	The first vertex of the quadrilater.
point2	The second vertex of the quadrilater.
point3	The third vertex of the quadrilater.
point4	The fourth vertex of the quadrilater.
point1Normal	The vertex normal at the first vertex of the quadrilater.
point2Normal	The vertex normal at the second vertex of the quadrilater.
point3Normal	The vertex normal at the third vertex of the quadrilater.
point4Normal	The vertex normal at the fourth vertex of the quadrilater.
fill	A collection of materials that will be applied to the Triangle3DElements returned by this method.
tag	A tag that will be applied to the Triangle3DElements returned by this method.
zIndex	A z-index that will be applied to the Triangle3DElements returned by this method.

Returns

A list containing two Triangle3DElements representing the quadrilater.

Definition at line 123 of file ObjectFactory.cs.

7.66.2.8 CreateSphere()

```
static List< Element3D > VectSharp.ThreeD.ObjectFactory.CreateSphere (
            Point3D center,
            double radius,
            int steps,
            IEnumerable< IMaterial > fill,
            string tag = null,
            int zIndex = 0 )  [static]
```

Creates a sphere.

Parameters

center	The centre of the sphere.
radius	The radius of the sphere.
steps	The number of meridians and parallels to use when generating the sphere.
fill	A collection of materials that will be applied to the Triangle3DElements returned by this method.
tag	A tag that will be applied to the Triangle3DElements returned by this method.
zIndex	A z-index that will be applied to the Triangle3DElements returned by this method.

Returns

A list of Triangle3DElements that constitute the sphere.

Definition at line 148 of file ObjectFactory.cs.

7.66.2.9 CreateTetrahedron()

```
static List< Element3D > VectSharp.ThreeD.ObjectFactory.CreateTetrahedron (
            Point3D center,
            double radius,
            IEnumerable< IMaterial > fill,
            string tag = null,
            int zIndex = 0 )  [static]
```

Creates a tetrahedron inscribed in a sphere.

Parameters

center	The centre of the tetrahedron.
radius	The radius of the sphere in which the tetrahedron is inscribed.
fill	A collection of materials that will be applied to the Triangle3DElements returned by this method.
tag	A tag that will be applied to the Triangle3DElements returned by this method.
zIndex	A z-index that will be applied to the Triangle3DElements returned by this method.

Returns

A list of Triangle3DElements that constitute the sphere.

Definition at line 238 of file ObjectFactory.cs.

7.66.2.10 CreateWireframe()

```
static List< Element3D > VectSharp.ThreeD.ObjectFactory.CreateWireframe (
            IEnumerable< Element3D > object3D,
            Colour colour,
            double thickness = 1,
            LineCaps lineCap = LineCaps.Butt,
            LineDash?  lineDash = null,
            string tag = null,
            int zIndex = 0 )  [static]
```

Creates a wireframe from a collection of Element3Ds.

Parameters

object3D	The collection of Element3Ds. Line3DElements and Point3DElements are ignored.
colour	The colour of the Line3DElements returned by this method.
thickness	The thickness of the Line3DElements returned by this method.
lineCap	The line cap of the Line3DElements returned by this method.
lineDash	The line dash of the Line3DElements returned by this method.
tag	A tag that will be applied to the Line3DElements returned by this method.
zIndex	A z-index that will be applied to the Line3DElements returned by this method.

Returns

A list of Line3DElements that constitute the wireframe.

Definition at line 370 of file ObjectFactory.cs.

The documentation for this class was generated from the following file:

- VectSharp.ThreeD/ObjectFactory.cs

7.67 VectSharp.Page Class Reference

Represents a Graphics object with a width and height.

Public Member Functions

- Page (double width, double height)

  Create a new page.
- void Crop (Rectangle region, bool removeClippedGraphics=false, string tag=null)

  Translate and resize the Page so that it displays the specified region .
- void Crop (Point topLeft, Size size, bool removeClippedGraphics=false, string tag=null)

  Translate and resize the Page so that it displays the rectangle defined by topLeft and size .
- void Crop (string tag=null)

  Translate and resize the Page so that it displays the rectangle corresponding to the bounding box of its contents.

Properties

- double Width `[get, set]`

  Width of the page.
- double Height `[get, set]`

  Height of the page.
- Graphics Graphics `[get, set]`

  Graphics surface of the page.
- Colour Background = Colour.FromRgba(255, 255, 255, 0) `[get, set]`

  Background colour of the page.

7.67.1 Detailed Description

Represents a Graphics object with a width and height.

Definition at line 47 of file Document.cs.

7.67.2 Constructor & Destructor Documentation

7.67.2.1 Page()

```
VectSharp.Page.Page (
            double width,
            double height )
```

Create a new page.

Parameters

width	The width of the page.
height	The height of the page.

Definition at line 74 of file Document.cs.

7.67.3 Member Function Documentation

7.67.3.1 Crop() [1/3]

```
void VectSharp.Page.Crop (
            Point topLeft,
            Size size,
            bool removeClippedGraphics = false,
            string tag = null )
```

Translate and resize the Page so that it displays the rectangle defined by *topLeft* and *size* .

Parameters

topLeft	The top left corner of the area to include in the page.
size	The size of the area to include in the page.
removeClippedGraphics	If this is `true`, graphics actions that fall outside of the specified region are completely removed from the plot, otherwise they are just hidden.
tag	A tag to identify the transform.

Definition at line 101 of file Document.cs.

7.67.3.2 Crop() [2/3]

```
void VectSharp.Page.Crop (
            Rectangle region,
            bool removeClippedGraphics = false,
            string tag = null )
```

Translate and resize the Page so that it displays the specified *region* .

Parameters

region	The area to include in the page.
removeClippedGraphics	If this is `true`, graphics actions that fall outside of the specified region are completely removed from the plot, otherwise they are just hidden.
tag	A tag to identify the transform.

Definition at line 89 of file Document.cs.

7.67.3.3 Crop() [3/3]

```
void VectSharp.Page.Crop (
            string tag = null )
```

Translate and resize the Page so that it displays the rectangle corresponding to the bounding box of its contents.

Parameters

tag	A tag to identify the transform.

Definition at line 129 of file Document.cs.

7.67.4 Property Documentation

7.67.4.1 Background

```
Colour VectSharp.Page.Background = Colour.FromRgba(255, 255, 255, 0)  [get], [set]
```

Background colour of the page.

Definition at line 67 of file Document.cs.

7.67.4.2 Graphics

```
Graphics VectSharp.Page.Graphics  [get], [set]
```

Graphics surface of the page.

Definition at line 62 of file Document.cs.

7.67.4.3 Height

```
double VectSharp.Page.Height  [get], [set]
```

Height of the page.

Definition at line 57 of file Document.cs.

7.67.4.4 Width

```
double VectSharp.Page.Width [get], [set]
```

Width of the page.

Definition at line 52 of file Document.cs.

The documentation for this class was generated from the following file:

- VectSharp/Document.cs

7.68 VectSharp.TrueTypeFile.PairKerning Class Reference

Contains information describing how the position of two glyphs in a kerning pair should be altered.

Properties

- Point Glyph1Placement [get]

  This vector contains the displacement that should be applied to the first glyph of the pair.
- Point Glyph1Advance [get]

  This vector describes how the advance width of the first glyph should be altered.
- Point Glyph2Placement [get]

  This vector contains the displacement that should be applied to the second glyph of the pair.
- Point Glyph2Advance [get]

  This vector describes how the advance width of the second glyph should be altered.

7.68.1 Detailed Description

Contains information describing how the position of two glyphs in a kerning pair should be altered.

Definition at line 4258 of file TrueType.cs.

7.68.2 Property Documentation

7.68.2.1 Glyph1Advance

```
Point VectSharp.TrueTypeFile.PairKerning.Glyph1Advance [get]
```

This vector describes how the advance width of the first glyph should be altered.

Definition at line 4268 of file TrueType.cs.

7.68.2.2 Glyph1Placement

```
Point VectSharp.TrueTypeFile.PairKerning.Glyph1Placement [get]
```

This vector contains the displacement that should be applied to the first glyph of the pair.

Definition at line 4263 of file TrueType.cs.

7.68.2.3 Glyph2Advance

```
Point VectSharp.TrueTypeFile.PairKerning.Glyph2Advance [get]
```

This vector describes how the advance width of the second glyph should be altered.

Definition at line 4278 of file TrueType.cs.

7.68.2.4 Glyph2Placement

```
Point VectSharp.TrueTypeFile.PairKerning.Glyph2Placement [get]
```

This vector contains the displacement that should be applied to the second glyph of the pair.

Definition at line 4273 of file TrueType.cs.

The documentation for this class was generated from the following file:

- VectSharp/TrueType.cs

7.69 VectSharp.ThreeD.ParallelLightSource Class Reference

Represents a parallel light source.

Inheritance diagram for VectSharp.ThreeD.ParallelLightSource:

Public Member Functions

- ParallelLightSource (double intensity, NormalizedVector3D direction)

  Creates a new ParallelLightSource instance.
- LightIntensity GetLightAt (Point3D point)

  Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	*The Point3DElement at which the light intensity should be computed.*

Returns

- double GetObstruction (Point3D point, IEnumerable< Triangle3DElement > shadowingTriangles)

  Determines the amount of obstruction of the light that results at point due to the specified shadowingTriangles .

Parameters

point	*The Point3D at which the obstruction should be computed.*
shadowingTriangles	*A collection of Triangle3DElement casting shadows.*

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Properties

- double Intensity `[get, set]`

  The intensity of the light.
- NormalizedVector3D Direction `[get]`

  The direction along which the light travels.
- NormalizedVector3D ReverseDirection `[get]`

  The reverse of Direction.
- bool CastsShadow = true `[get, set]`

  Determines whether the light casts a shadow or not.

7.69.1 Detailed Description

Represents a parallel light source.

Definition at line 126 of file Lights.cs.

7.69.2 Constructor & Destructor Documentation

7.69.2.1 ParallelLightSource()

```
VectSharp.ThreeD.ParallelLightSource.ParallelLightSource (
            double intensity,
            NormalizedVector3D direction )
```

Creates a new ParallelLightSource instance.

Parameters

intensity	The intensity of the light.
direction	The direction along which the light travels.

Definition at line 151 of file Lights.cs.

7.69.3 Member Function Documentation

7.69.3.1 GetLightAt()

```
LightIntensity VectSharp.ThreeD.ParallelLightSource.GetLightAt (
            Point3D point )
```

Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	The Point3DElement at which the light intensity should be computed.

Returns

Implements VectSharp.ThreeD.ILightSource.

Definition at line 159 of file Lights.cs.

7.69.3.2 GetObstruction()

```
double VectSharp.ThreeD.ParallelLightSource.GetObstruction (
            Point3D point,
            IEnumerable< Triangle3DElement > shadowingTriangles )
```

Determines the amount of obstruction of the light that results at *point* due to the specified *shadowingTriangles* .

Parameters

point	The Point3D at which the obstruction should be computed.
shadowingTriangles	A collection of Triangle3DElement casting shadows.

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 165 of file Lights.cs.

7.69.4 Property Documentation

7.69.4.1 CastsShadow

```
bool VectSharp.ThreeD.ParallelLightSource.CastsShadow = true  [get], [set]
```

Determines whether the light casts a shadow or not.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 144 of file Lights.cs.

7.69.4.2 Direction

```
NormalizedVector3D VectSharp.ThreeD.ParallelLightSource.Direction  [get]
```

The direction along which the light travels.

Definition at line 136 of file Lights.cs.

7.69.4.3 Intensity

```
double VectSharp.ThreeD.ParallelLightSource.Intensity  [get], [set]
```

The intensity of the light.

Definition at line 131 of file Lights.cs.

7.69.4.4 ReverseDirection

```
NormalizedVector3D VectSharp.ThreeD.ParallelLightSource.ReverseDirection  [get]
```

The reverse of Direction.

Definition at line 141 of file Lights.cs.

The documentation for this class was generated from the following file:

- VectSharp.ThreeD/Lights.cs

7.70 VectSharp.SVG.Parser Class Reference

Contains methods to read an SVG image file.

Static Public Member Functions

- static Page ParseSVGURI (string uri, bool ignored=false)

  Parses an SVG image URI.
- static Page FromString (string svgSource)

  Parses SVG source into a Page containing the image represented by the code.
- static Page FromFile (string fileName)

  Parses an SVG image file into a Page containing the image.
- static Page FromStream (Stream svgSourceStream)

  Parses an stream containing SVG source code into a Page containing the image represented by the code.

Static Public Attributes

- static Func< string, bool, Page > ParseImageURI

  A function that takes as input an image URI and a boolean value indicating whether the image should be interpolated, and returns a Page object containing the image. By default, this is equal to ParseSVGURI, i.e. it is only able to parse SVG images. If you wish to enable the parsing of other formats, you should install the "VectSharp.MuPDFUtils" NuGet package and enable the parser in your program by doing something like:

7.70.1 Detailed Description

Contains methods to read an SVG image file.

Definition at line 33 of file SVGParser.cs.

7.70.2 Member Function Documentation

7.70.2.1 FromFile()

```
static Page VectSharp.SVG.Parser.FromFile (
            string fileName )  [static]
```

Parses an SVG image file into a Page containing the image.

Parameters

fileName	The path to the SVG image file.

Returns

A Page containing the image represented by the file.

Definition at line 180 of file SVGParser.cs.

7.70.2.2 FromStream()

```
static Page VectSharp.SVG.Parser.FromStream (
            Stream svgSourceStream )  [static]
```

Parses an stream containing SVG source code into a Page containing the image represented by the code.

Parameters

svgSourceStream	The stream containing SVG source code.

Returns

A Page containing the image represented by the *svgSourceStream* .

Definition at line 190 of file SVGParser.cs.

7.70.2.3 FromString()

```
static Page VectSharp.SVG.Parser.FromString (
            string svgSource )  [static]
```

Parses SVG source into a Page containing the image represented by the code.

Parameters

svgSource	The SVG source code.

Returns

A Page containing the image represented by the *svgSource* .

Definition at line 116 of file SVGParser.cs.

7.70.2.4 ParseSVGURI()

```
static Page VectSharp.SVG.Parser.ParseSVGURI (
            string uri,
            bool ignored = false )  [static]
```

Parses an SVG image URI.

Parameters

uri	The image URI to parse.
ignored	This value is ignored and is only needed for compatibility.

Returns

A Page containing the parsed SVG image, or null.

Definition at line 54 of file SVGParser.cs.

7.70.3 Member Data Documentation

7.70.3.1 ParseImageURI

```
Func<string, bool, Page> VectSharp.SVG.Parser.ParseImageURI  [static]
```

A function that takes as input an image URI and a boolean value indicating whether the image should be interpolated, and returns a Page object containing the image. By default, this is equal to ParseSVGURI, i.e. it is only able to parse SVG images. If you wish to enable the parsing of other formats, you should install the "VectSharp.Mu↩ PDFUtils" NuGet package and enable the parser in your program by doing something like:

```
VectSharp.SVG.Parser.ParseImageURI = VectSharp.MuPDFUtils.ImageURIParser.Parser(VectShar
```

Definition at line 46 of file SVGParser.cs.

The documentation for this class was generated from the following file:

- VectSharp.SVG/SVGParser.cs

7.71 VectSharp.PDF.PDFContextInterpreter Class Reference

Contains methods to render a Document as a PDF document.

Classes

- class FilterOption

  Determines how and whether image filters are rasterised.

Public Types

- enum TextOptions

  Defines whether the used fonts should be included in the file.

Static Public Member Functions

- static void SaveAsPDF (this Document document, string fileName, TextOptions textOption=TextOptions.↩ SubsetFonts, bool compressStreams=true, Dictionary< string, string > linkDestinations=null, FilterOption filterOption=default)

  Save the document to a PDF file.
- static void SaveAsPDF (this Document document, Stream stream, TextOptions textOption=TextOptions.↩ SubsetFonts, bool compressStreams=true, Dictionary< string, string > linkDestinations=null, FilterOption filterOption=default)

  Save the document to a PDF stream.

7.71.1 Detailed Description

Contains methods to render a Document as a PDF document.

Definition at line 829 of file PDFContext.cs.

7.71.2 Member Enumeration Documentation

7.71.2.1 TextOptions

```
enum VectSharp.PDF.PDFContextInterpreter.TextOptions
```

Defines whether the used fonts should be included in the file.

Definition at line 1124 of file PDFContext.cs.

7.71.3 Member Function Documentation

7.71.3.1 SaveAsPDF() [1/2]

```
static void VectSharp.PDF.PDFContextInterpreter.SaveAsPDF (
            this Document document,
            Stream stream,
            TextOptions textOption = TextOptions.SubsetFonts,
            bool compressStreams = true,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default )  [static]
```

Save the document to a PDF stream.

Parameters

document	The Document to save.
stream	The stream to which the PDF data will be written.
textOption	Defines whether the used fonts should be included in the file.
compressStreams	Indicates whether the streams in the PDF file should be compressed.
linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Definition at line 1208 of file PDFContext.cs.

7.71.3.2 SaveAsPDF() [2/2]

```
static void VectSharp.PDF.PDFContextInterpreter.SaveAsPDF (
            this Document document,
            string fileName,
            TextOptions textOption = TextOptions.SubsetFonts,
            bool compressStreams = true,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default )  [static]
```

Save the document to a PDF file.

Parameters

document	The Document to save.
fileName	The full path to the file to save. If it exists, it will be overwritten.
textOption	Defines whether the used fonts should be included in the file.
compressStreams	Indicates whether the streams in the PDF file should be compressed.
linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Definition at line 1113 of file PDFContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.PDF/PDFContext.cs

7.72 VectSharp.ThreeD.PhongMaterial Class Reference

Represents a material that uses a Phong reflection model to determine the colour of the material based on the light sources that hit it.

Inheritance diagram for VectSharp.ThreeD.PhongMaterial:

Public Member Functions

- PhongMaterial (Colour colour)

  Creates a new PhongMaterial instance.
- Colour GetColour (Point3D point, NormalizedVector3D surfaceNormal, Camera camera, IList< ILightSource > lights, IList< double > obstructions)

  Obtains the Colour at the specified point.

Parameters

point	*The point whose colour should be determined.*
surfaceNormal	*The normal to the surface at the specified point .*
camera	*The camera being used to render the scene.*
lights	*A list of light sources that are present in the scene.*
obstructions	*A list of values indicating how obstructed each light source is.*

Returns

The Colour of the specified point.

Properties

- Colour Colour `[get]`

  The base colour of the material.
- double AmbientReflectionCoefficient = 1 `[get, set]`

  A coefficient determining how much ambient light is reflected by the material.
- double DiffuseReflectionCoefficient = 1 `[get, set]`

  A coefficient determining how much directional light is reflected by the material.
- double SpecularReflectionCoefficient = 1 `[get, set]`

  A coefficient determining the intensity of specular highlights.
- double SpecularShininess = 1 `[get, set]`

  A coefficient determining the extent of specular highlights.

7.72.1 Detailed Description

Represents a material that uses a Phong reflection model to determine the colour of the material based on the light sources that hit it.

Definition at line 74 of file Materials.cs.

7.72.2 Constructor & Destructor Documentation

7.72.2.1 PhongMaterial()

```
VectSharp.ThreeD.PhongMaterial.PhongMaterial (
            Colour colour )
```

Creates a new PhongMaterial instance.

Parameters

colour	The base colour of the material.

Definition at line 111 of file Materials.cs.

7.72.3 Member Function Documentation

7.72.3.1 GetColour()

```
Colour VectSharp.ThreeD.PhongMaterial.GetColour (
            Point3D point,
            NormalizedVector3D surfaceNormal,
            Camera camera,
            IList< ILightSource > lights,
            IList< double > obstructions )
```

Obtains the Colour at the specified point.

Parameters

point	The point whose colour should be determined.
surfaceNormal	The normal to the surface at the specified *point* .
camera	The camera being used to render the scene.
lights	A list of light sources that are present in the scene.
obstructions	A list of values indicating how obstructed each light source is.

Returns

The Colour of the specified point.

Implements VectSharp.ThreeD.IMaterial.

Definition at line 142 of file Materials.cs.

7.72.4 Property Documentation

7.72.4.1 AmbientReflectionCoefficient

```
double VectSharp.ThreeD.PhongMaterial.AmbientReflectionCoefficient = 1  [get], [set]
```

A coefficient determining how much ambient light is reflected by the material.

Definition at line 90 of file Materials.cs.

7.72.4.2 Colour

```
Colour VectSharp.ThreeD.PhongMaterial.Colour  [get]
```

The base colour of the material.

Definition at line 79 of file Materials.cs.

7.72.4.3 DiffuseReflectionCoefficient

```
double VectSharp.ThreeD.PhongMaterial.DiffuseReflectionCoefficient = 1  [get], [set]
```

A coefficient determining how much directional light is reflected by the material.

Definition at line 95 of file Materials.cs.

7.72.4.4 SpecularReflectionCoefficient

```
double VectSharp.ThreeD.PhongMaterial.SpecularReflectionCoefficient = 1  [get], [set]
```

A coefficient determining the intensity of specular highlights.

Definition at line 100 of file Materials.cs.

7.72.4.5 SpecularShininess

```
double VectSharp.ThreeD.PhongMaterial.SpecularShininess = 1  [get], [set]
```

A coefficient determining the extent of specular highlights.

Definition at line 105 of file Materials.cs.

The documentation for this class was generated from the following file:

- VectSharp.ThreeD/Materials.cs

7.73 VectSharp.Point Struct Reference

Represents a point relative to an origin in the top-left corner.

Public Member Functions

- Point (double x, double y)

  Create a new Point.
- double Modulus ()

  Computes the modulus of the vector represented by the Point.
- Point Normalize ()

  Normalises a Point.
- bool IsEqual (Point p2, double tolerance)

  Checks whether this Point is equal to another Point, up to a specified tolerance.

Static Public Member Functions

- static Point Min (Point p1, Point p2)

  Computes the top-left corner of the Rectangle identified by two Points.
- static Point Max (Point p1, Point p2)

  Computes the bottom-right corner of the Rectangle identified by two Points.
- static Rectangle Bounds (IEnumerable< Point > points)

  Computes the smallest Rectangle that contains all the specified points.
- static Rectangle Bounds (params Point[] points)

  Computes the smallest Rectangle that contains all the specified points.

Public Attributes

- double X

  Horizontal (x) coordinate, measured to the right of the origin.
- double Y

  Vertical (y) coordinate, measured to the bottom of the origin.

7.73.1 Detailed Description

Represents a point relative to an origin in the top-left corner.

Definition at line 27 of file Point.cs.

7.73.2 Constructor & Destructor Documentation

7.73.2.1 Point()

```
VectSharp.Point.Point (
            double x,
            double y )
```

Create a new Point.

Parameters

x	The horizontal (x) coordinate.
y	The vertical (y) coordinate.

Definition at line 44 of file Point.cs.

7.73.3 Member Function Documentation

7.73.3.1 Bounds() [1/2]

```
static Rectangle VectSharp.Point.Bounds (
            IEnumerable< Point > points )  [static]
```

Computes the smallest Rectangle that contains all the specified points.

Parameters

points	The points whose bounds are being computed.

Returns

The smallest Rectangle that contains all the specified points.

Definition at line 107 of file Point.cs.

7.73.3.2 Bounds() [2/2]

```
static Rectangle VectSharp.Point.Bounds (
            params Point[] points )  [static]
```

Computes the smallest Rectangle that contains all the specified points.

Parameters

points	The points whose bounds are being computed.

Returns

The smallest Rectangle that contains all the specified points.

Definition at line 136 of file Point.cs.

7.73.3.3 IsEqual()

```
bool VectSharp.Point.IsEqual (
            Point p2,
            double tolerance )
```

Checks whether this Point is equal to another Point, up to a specified tolerance.

Parameters

p2	The Point to compare.
tolerance	The tolerance threshold.

Returns

`true` if both coordinates of the Points are closer than *tolerance* or if their relative difference (i.e. `(a - b) / (a + b) * 2`) is smaller than *tolerance* . `false` otherwise.

Definition at line 75 of file Point.cs.

7.73.3.4 Max()

```
static Point VectSharp.Point.Max (
            Point p1,
            Point p2 )  [static]
```

Computes the bottom-right corner of the Rectangle identified by two Points.

Parameters

p1	The first point.
p2	The second point.

Returns

A Point whose X coordinate is the largest between the one of *p1* and *p2* , and likewise for the Y coordinate.

Definition at line 97 of file Point.cs.

7.73.3.5 Min()

```
static Point VectSharp.Point.Min (
            Point p1,
            Point p2 )  [static]
```

Computes the top-left corner of the Rectangle identified by two Points.

Parameters

p1	The first point.
p2	The second point.

Returns

A Point whose X coordinate is the smallest between the one of *p1* and *p2* , and likewise for the Y coordinate.

Definition at line 86 of file Point.cs.

7.73.3.6 Modulus()

```
double VectSharp.Point.Modulus ( )
```

Computes the modulus of the vector represented by the Point.

Returns

The modulus of the vector represented by the Point.

Definition at line 54 of file Point.cs.

7.73.3.7 Normalize()

```
Point VectSharp.Point.Normalize ( )
```

Normalises a Point.

Returns

The normalised Point.

Definition at line 63 of file Point.cs.

7.73.4 Member Data Documentation

7.73.4.1 X

```
double VectSharp.Point.X
```

Horizontal (x) coordinate, measured to the right of the origin.

Definition at line 32 of file Point.cs.

7.73.4.2 Y

```
double VectSharp.Point.Y
```

Vertical (y) coordinate, measured to the bottom of the origin.

Definition at line 37 of file Point.cs.

The documentation for this struct was generated from the following file:

- VectSharp/Point.cs

7.74 VectSharp.ThreeD.PointLightSource Class Reference

Represents a point light source.

Inheritance diagram for VectSharp.ThreeD.PointLightSource:

Public Member Functions

- PointLightSource (double intensity, Point3D position)

  Creates a new PointLightSource instance.
- LightIntensity GetLightAt (Point3D point)

  Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	*The Point3DElement at which the light intensity should be computed.*

Returns

- double GetObstruction (Point3D point, IEnumerable< Triangle3DElement > shadowingTriangles)

  Determines the amount of obstruction of the light that results at point due to the specified shadowingTriangles .

Parameters

point	*The Point3D at which the obstruction should be computed.*
shadowingTriangles	*A collection of Triangle3DElement casting shadows.*

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Properties

- bool CastsShadow = true `[get, set]`

  Determines whether the light casts a shadow or not.
- Point3D Position `[get, set]`

  The position of the light source.
- double Intensity `[get, set]`

  The base intensity of the light.
- double DistanceAttenuationExponent = 2 `[get, set]`

  An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.

7.74.1 Detailed Description

Represents a point light source.

Definition at line 184 of file Lights.cs.

7.74.2 Constructor & Destructor Documentation

7.74.2.1 PointLightSource()

```
VectSharp.ThreeD.PointLightSource.PointLightSource (
            double intensity,
            Point3D position )
```

Creates a new PointLightSource instance.

Parameters

intensity	The intensity of the light.
position	The position of the light source.

Definition at line 209 of file Lights.cs.

7.74.3 Member Function Documentation

7.74.3.1 GetLightAt()

```
LightIntensity VectSharp.ThreeD.PointLightSource.GetLightAt (
            Point3D point )
```

Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	The Point3DElement at which the light intensity should be computed.

Returns

Implements VectSharp.ThreeD.ILightSource.

Definition at line 216 of file Lights.cs.

7.74.3.2 GetObstruction()

```
double VectSharp.ThreeD.PointLightSource.GetObstruction (
            Point3D point,
            IEnumerable< Triangle3DElement > shadowingTriangles )
```

Determines the amount of obstruction of the light that results at *point* due to the specified *shadowingTriangles* .

Parameters

point	The Point3D at which the obstruction should be computed.
shadowingTriangles	A collection of Triangle3DElement casting shadows.

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 233 of file Lights.cs.

7.74.4 Property Documentation

7.74.4.1 CastsShadow

```
bool VectSharp.ThreeD.PointLightSource.CastsShadow = true  [get], [set]
```

Determines whether the light casts a shadow or not.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 187 of file Lights.cs.

7.74.4.2 DistanceAttenuationExponent

```
double VectSharp.ThreeD.PointLightSource.DistanceAttenuationExponent = 2  [get], [set]
```

An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.

Definition at line 202 of file Lights.cs.

7.74.4.3 Intensity

```
double VectSharp.ThreeD.PointLightSource.Intensity  [get], [set]
```

The base intensity of the light.

Definition at line 197 of file Lights.cs.

7.74.4.4 Position

```
Point3D VectSharp.ThreeD.PointLightSource.Position  [get], [set]
```

The position of the light source.

Definition at line 192 of file Lights.cs.

The documentation for this class was generated from the following file:

- VectSharp.ThreeD/Lights.cs

7.75 VectSharp.RadialGradientBrush Class Reference

Represents a brush painting with a radial gradient.

Inheritance diagram for VectSharp.RadialGradientBrush:

Public Member Functions

- RadialGradientBrush (Point focalPoint, Point centre, double radius, params GradientStop[ ] gradientStops)

  Creates a new RadialGradientBrush with the specified focal point, centre, radius and gradient stops.
- RadialGradientBrush (Point focalPoint, Point centre, double radius, IEnumerable< GradientStop > gradient↩ Stops)

  Creates a new RadialGradientBrush with the specified focal point, centre, radius and gradient stops.
- override Brush MultiplyOpacity (double opacity)

  Returns a brush corresponding the current instance, with the specified opacity multiplication applied.

Parameters

opacity	*The value that will be used to multiply the opacity of the brush.*

Returns

A brush corresponding the current instance, with the specified opacity multiplication applied.

Properties

- Point FocalPoint [get]

  The focal point of the gradient (i.e. the point within the circle where the gradient starts).
- Point Centre [get]

  Represents the centre of the gradient.
- double Radius [get]

  The radius of the gradient.

Additional Inherited Members

7.75.1 Detailed Description

Represents a brush painting with a radial gradient.

Definition at line 400 of file Brush.cs.

7.75.2 Constructor & Destructor Documentation

7.75.2.1 RadialGradientBrush() [1/2]

```
VectSharp.RadialGradientBrush.RadialGradientBrush (
            Point focalPoint,
            Point centre,
            double radius,
            params GradientStop[] gradientStops )
```

Creates a new RadialGradientBrush with the specified focal point, centre, radius and gradient stops.

Parameters

focalPoint	The focal point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
centre	The centre of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
radius	The radius of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
gradientStops	The colour stops in the gradient.

Definition at line 424 of file Brush.cs.

7.75.2.2 RadialGradientBrush() [2/2]

```
VectSharp.RadialGradientBrush.RadialGradientBrush (
            Point focalPoint,
            Point centre,
            double radius,
            IEnumerable< GradientStop > gradientStops )
```

Creates a new RadialGradientBrush with the specified focal point, centre, radius and gradient stops.

Parameters

focalPoint	The focal point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
centre	The centre of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
radius	The radius of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
gradientStops	The colour stops in the gradient.

Definition at line 463 of file Brush.cs.

7.75.3 Member Function Documentation

7.75.3.1 MultiplyOpacity()

```
override Brush VectSharp.RadialGradientBrush.MultiplyOpacity (
            double opacity )  [virtual]
```

Returns a brush corresponding the current instance, with the specified *opacity* multiplication applied.

Parameters

opacity	The value that will be used to multiply the opacity of the brush.

Returns

A brush corresponding the current instance, with the specified *opacity* multiplication applied.

Implements VectSharp.Brush.

Definition at line 496 of file Brush.cs.

7.75.4 Property Documentation

7.75.4.1 Centre

```
Point VectSharp.RadialGradientBrush.Centre [get]
```

Represents the centre of the gradient.

Definition at line 410 of file Brush.cs.

7.75.4.2 FocalPoint

```
Point VectSharp.RadialGradientBrush.FocalPoint [get]
```

The focal point of the gradient (i.e. the point within the circle where the gradient starts).

Definition at line 405 of file Brush.cs.

7.75.4.3 Radius

```
double VectSharp.RadialGradientBrush.Radius [get]
```

The radius of the gradient.

Definition at line 415 of file Brush.cs.

The documentation for this class was generated from the following file:

- VectSharp/Brush.cs

7.76 VectSharp.TrueTypeFile.CoverageTable.RangeRecord Struct Reference

Public Member Functions

- RangeRecord (Stream fs)

Public Attributes

- ushort StartGlyphID
- ushort EndGlyphID
- ushort StartCoverageIndex

7.76.1 Detailed Description

Definition at line 3620 of file TrueType.cs.

7.76.2 Constructor & Destructor Documentation

7.76.2.1 RangeRecord()

```
VectSharp.TrueTypeFile.CoverageTable.RangeRecord.RangeRecord (
            Stream fs )
```

Definition at line 3626 of file TrueType.cs.

7.76.3 Member Data Documentation

7.76.3.1 EndGlyphID

```
ushort VectSharp.TrueTypeFile.CoverageTable.RangeRecord.EndGlyphID
```

Definition at line 3623 of file TrueType.cs.

7.76.3.2 StartCoverageIndex

```
ushort VectSharp.TrueTypeFile.CoverageTable.RangeRecord.StartCoverageIndex
```

Definition at line 3624 of file TrueType.cs.

7.76.3.3 StartGlyphID

```
ushort VectSharp.TrueTypeFile.CoverageTable.RangeRecord.StartGlyphID
```

Definition at line 3622 of file TrueType.cs.

The documentation for this struct was generated from the following file:

- VectSharp/TrueType.cs

7.77 VectSharp.Raster.Raster Class Reference

Contains methods to render a page to a PNG image.

Static Public Member Functions

- static void SaveAsPNG (this Page page, string fileName, double scale=1)

  Render the page to a PNG file.
- static void SaveAsPNG (this Page page, Stream stream, double scale=1)

  Render the page to a PNG stream.
- static RasterImage Rasterise (this Graphics graphics, Rectangle region, double scale, bool interpolate)

  Rasterise a region of a Graphics object.
- static DisposableIntPtr SaveAsRawBytes (this Page pag, out int width, out int height, out int totalSize, double scale=1)

  Render the page to raw pixel data, in 32bpp RGBA format.
- static void SaveAsAnimatedPNG (this Animation animation, Stream imageStream, double scale=1, double frameRate=60, double durationScaling=1, AnimatedPNG.InterframeCompression interframe↩ Compression=AnimatedPNG.InterframeCompression.First)

  Saves the animation to a stream in animated PNG format.
- static void SaveAsAnimatedPNG (this Animation animation, string fileName, double scale=1, double frameRate=60, double durationScaling=1, AnimatedPNG.InterframeCompression interframe↩ Compression=AnimatedPNG.InterframeCompression.First)

  Saves the animation to an animated PNG file.

7.77.1 Detailed Description

Contains methods to render a page to a PNG image.

Definition at line 30 of file Raster.cs.

7.77.2 Member Function Documentation

7.77.2.1 Rasterise()

```
static RasterImage VectSharp.Raster.Raster.Rasterise (
            this Graphics graphics,
            Rectangle region,
            double scale,
            bool interpolate )  [static]
```

Rasterise a region of a Graphics object.

Parameters

graphics	The Graphics object that will be rasterised.
region	The region of the *graphics* that will be rasterised.
scale	The scale at which the image will be rendered.
interpolate	Whether the resulting image should be interpolated or not when it is drawn on another Graphics surface.

Returns

A RasterImage containing the rasterised graphics.

Definition at line 91 of file Raster.cs.

7.77.2.2 SaveAsAnimatedPNG() [1/2]

```
static void VectSharp.Raster.Raster.SaveAsAnimatedPNG (
            this Animation animation,
            Stream imageStream,
            double scale = 1,
            double frameRate = 60,
            double durationScaling = 1,
            AnimatedPNG.InterframeCompression interframeCompression = AnimatedPNG.InterframeCompression.First
)  [static]
```

Saves the animation to a stream in animated PNG format.

Parameters

animation	The animation to export.
imageStream	The stream on which the animated PNG will be written.
scale	The scale at which the animation will be rendered.
frameRate	The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated PNG specification at 90 fps.
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
interframeCompression	The kind of compression that will be used to reduce file size. Note that if the animation has a transparent background, no compression can be performed, and the value of this parameter is ignored.

Definition at line 184 of file Raster.cs.

7.77.2.3 SaveAsAnimatedPNG() [2/2]

```
static void VectSharp.Raster.Raster.SaveAsAnimatedPNG (
            this Animation animation,
            string fileName,
            double scale = 1,
            double frameRate = 60,
            double durationScaling = 1,
            AnimatedPNG.InterframeCompression interframeCompression = AnimatedPNG.InterframeCompression.First
)  [static]
```

Saves the animation to an animated PNG file.

Parameters

animation	The animation to export.
fileName	The output file to create.
scale	The scale at which the animation will be rendered.
frameRate	The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated PNG specification at 90 fps.
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
interframeCompression	The kind of compression that will be used to reduce file size. Note that if the animation has a transparent background, no compression can be performed, and the value of this parameter is ignored.

Definition at line 267 of file Raster.cs.

7.77.2.4 SaveAsPNG() [1/2]

```
static void VectSharp.Raster.Raster.SaveAsPNG (
            this Page page,
            Stream stream,
            double scale = 1 )  [static]
```

Render the page to a PNG stream.

Parameters

page	The Page to render.
stream	The stream to which the PNG data will be written.
scale	The scale to be used when rasterising the page. This will determine the width and height of the image file.

Definition at line 64 of file Raster.cs.

7.77.2.5 SaveAsPNG() [2/2]

```
static void VectSharp.Raster.Raster.SaveAsPNG (
            this Page page,
            string fileName,
            double scale = 1 )  [static]
```

Render the page to a PNG file.

Parameters

page	The Page to render.
fileName	The full path to the file to save. If it exists, it will be overwritten.
scale	The scale to be used when rasterising the page. This will determine the width and height of the image file.

Definition at line 39 of file Raster.cs.

7.77.2.6 SaveAsRawBytes()

```
static DisposableIntPtr VectSharp.Raster.Raster.SaveAsRawBytes (
            this Page pag,
            out int width,
            out int height,
            out int totalSize,
            double scale = 1 )  [static]
```

Render the page to raw pixel data, in 32bpp RGBA format.

Parameters

pag	The Page to render.
scale	The scale to be used when rasterising the page. This will determine the width and height of the image.
width	The width of the rendered image.
height	The height of the rendered image.
totalSize	The size in bytes of the raw pixel data.

Returns

A DisposableIntPtr containing a pointer to the raw pixel data, stored in unmanaged memory. Dispose this object to release the unmanaged memory.

Definition at line 143 of file Raster.cs.

The documentation for this class was generated from the following file:

- VectSharp.Raster/Raster.cs

7.78 VectSharp.RasterImage Class Reference

Represents a raster image, created from raw pixel data. Consider using the derived classes included in the NuGet package "VectSharp.MuPDFUtils" if you need to load a raster image from a file or a Stream.

Inheritance diagram for VectSharp.RasterImage:

Public Member Functions

- RasterImage (IntPtr pixelData, int width, int height, bool hasAlpha, bool interpolate)

  Creates a new RasterImage instance from the specified pixel data in RGB or RGBA format.
- RasterImage (ref DisposableIntPtr pixelData, int width, int height, bool hasAlpha, bool interpolate)

  Creates a new RasterImage instance from the specified pixel data in RGB or RGBA format.
- RasterImage (byte[] data, int width, int height, PixelFormats pixelFormat, bool interpolate)

  Creates a new RasterImage instance copying the specified pixel data.
- void ClearPNGCache ()

  Disposes the PNGStream. Also useful if is is necessary to regenerate it, e.g. because the underlying image pixel data has changed.
- void Dispose ()

Properties

- IntPtr ImageDataAddress `[get]`

  The memory address of the image pixel data.
- IDisposable DataHolder `[get]`

  An IDisposable that will be disposed when the image is disposed.
- string Id `[get]`

  A univocal identifier for this image.
- bool HasAlpha `[get]`

  Determines whether the image has an alpha channel.
- int Width `[get]`

  The width in pixels of the image.
- int Height `[get]`

  The height in pixels of the image.
- bool Interpolate `[get]`

  Determines whether the image should be interpolated when it is resized.
- MemoryStream PNGStream `[get]`

  Contains a representation of the image in PNG format. Generated at the first access and cached until the image is disposed.

7.78.1 Detailed Description

Represents a raster image, created from raw pixel data. Consider using the derived classes included in the NuGet package "VectSharp.MuPDFUtils" if you need to load a raster image from a file or a Stream.

Definition at line 98 of file RasterImage.cs.

7.78.2 Constructor & Destructor Documentation

7.78.2.1 RasterImage() [1/3]

```
VectSharp.RasterImage.RasterImage (
            IntPtr pixelData,
            int width,
            int height,
            bool hasAlpha,
            bool interpolate )
```

Creates a new RasterImage instance from the specified pixel data in RGB or RGBA format.

Parameters

pixelData	The address of the image pixel data in RGB or RGBA format.
width	The width in pixels of the image.
height	The height in pixels of the image.
hasAlpha	true if the image is in RGBA format, false if it is in RGB format.
interpolate	Whether the image should be interpolated when it is resized.

Definition at line 170 of file RasterImage.cs.

7.78.2.2 RasterImage() [2/3]

```
VectSharp.RasterImage.RasterImage (
            ref DisposableIntPtr pixelData,
            int width,
            int height,
            bool hasAlpha,
            bool interpolate )
```

Creates a new RasterImage instance from the specified pixel data in RGB or RGBA format.

Parameters

pixelData	The address of the image pixel data in RGB or RGBA format wrapped in a DisposableIntPtr. The RasterImage will take ownership of this memory.

Parameters

width	The width in pixels of the image.
height	The height in pixels of the image.
hasAlpha	true if the image is in RGBA format, false if it is in RGB format.
interpolate	Whether the image should be interpolated when it is resized.

Definition at line 188 of file RasterImage.cs.

7.78.2.3 RasterImage() [3/3]

```
VectSharp.RasterImage.RasterImage (
            byte[] data,
            int width,
            int height,
            PixelFormats pixelFormat,
            bool interpolate )
```

Creates a new RasterImage instance copying the specified pixel data.

Parameters

data	The image pixel data that will be copied.
width	The width in pixels of the image.
height	The height in pixels of the image.
pixelFormat	The format of the pixel data.
interpolate	Whether the image should be interpolated when it is resized.

Definition at line 207 of file RasterImage.cs.

7.78.3 Member Function Documentation

7.78.3.1 ClearPNGCache()

```
void VectSharp.RasterImage.ClearPNGCache ( )
```

Disposes the PNGStream. Also useful if is is necessary to regenerate it, e.g. because the underlying image pixel data has changed.

Definition at line 261 of file RasterImage.cs.

7.78.3.2 Dispose()

```
void VectSharp.RasterImage.Dispose ( )
```

Definition at line 295 of file RasterImage.cs.

7.78.4 Property Documentation

7.78.4.1 DataHolder

```
IDisposable VectSharp.RasterImage.DataHolder [get]
```

An IDisposable that will be disposed when the image is disposed.

Definition at line 108 of file RasterImage.cs.

7.78.4.2 HasAlpha

```
bool VectSharp.RasterImage.HasAlpha [get]
```

Determines whether the image has an alpha channel.

Definition at line 118 of file RasterImage.cs.

7.78.4.3 Height

```
int VectSharp.RasterImage.Height [get]
```

The height in pixels of the image.

Definition at line 128 of file RasterImage.cs.

7.78.4.4 Id

```
string VectSharp.RasterImage.Id [get]
```

A univocal identifier for this image.

Definition at line 113 of file RasterImage.cs.

7.78.4.5 ImageDataAddress

```
IntPtr VectSharp.RasterImage.ImageDataAddress [get]
```

The memory address of the image pixel data.

Definition at line 103 of file RasterImage.cs.

7.78.4.6 Interpolate

```
bool VectSharp.RasterImage.Interpolate [get]
```

Determines whether the image should be interpolated when it is resized.

Definition at line 133 of file RasterImage.cs.

7.78.4.7 PNGStream

```
MemoryStream VectSharp.RasterImage.PNGStream [get]
```

Contains a representation of the image in PNG format. Generated at the first access and cached until the image is disposed.

Definition at line 140 of file RasterImage.cs.

7.78.4.8 Width

```
int VectSharp.RasterImage.Width [get]
```

The width in pixels of the image.

Definition at line 123 of file RasterImage.cs.

The documentation for this class was generated from the following file:

- VectSharp/RasterImage.cs

7.79 VectSharp.ImageSharpUtils.RasterImageFile Class Reference

A RasterImage created from a file.

Inheritance diagram for VectSharp.ImageSharpUtils.RasterImageFile:

Public Member Functions

- RasterImageFile (string fileName, bool alpha=true, bool interpolate=true)

  Creates a new RasterImage from the specified file.

Additional Inherited Members

7.79.1 Detailed Description

A RasterImage created from a file.

Definition at line 30 of file RasterImages.cs.

7.79.2 Constructor & Destructor Documentation

7.79.2.1 RasterImageFile()

```
VectSharp.ImageSharpUtils.RasterImageFile.RasterImageFile (
            string fileName,
            bool alpha = true,
            bool interpolate = true )
```

Creates a new RasterImage from the specified file.

Parameters

fileName	The path to the file containing the image.
alpha	A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.
interpolate	A boolean value indicating whether the image should be interpolated when it is resized or not.

Definition at line 38 of file RasterImages.cs.

The documentation for this class was generated from the following file:

- VectSharp.ImageSharpUtils/RasterImages.cs

7.80 VectSharp.MuPDFUtils.RasterImageFile Class Reference

A RasterImage created from a file.

Inheritance diagram for VectSharp.MuPDFUtils.RasterImageFile:

Public Member Functions

- RasterImageFile (string fileName, int pageNumber=0, double scale=1, bool alpha=true, bool interpolate=true)

  Creates a new RasterImage from the specified file.

Additional Inherited Members

7.80.1 Detailed Description

A RasterImage created from a file.

Definition at line 28 of file RasterImages.cs.

7.80.2 Constructor & Destructor Documentation

7.80.2.1 RasterImageFile()

```
VectSharp.MuPDFUtils.RasterImageFile.RasterImageFile (
            string fileName,
            int pageNumber = 0,
            double scale = 1,
            bool alpha = true,
            bool interpolate = true )
```

Creates a new RasterImage from the specified file.

Parameters

fileName	The path to the file containing the image.
pageNumber	The number of the page in the file from which the image should be created, starting at 0. Only useful for multi-page formats, such as PDF.
scale	The scale factor at which to render the image.
alpha	A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.
interpolate	A boolean value indicating whether the image should be interpolated when it is resized or not.

Definition at line 38 of file RasterImages.cs.

The documentation for this class was generated from the following file:

- VectSharp.MuPDFUtils/RasterImages.cs

7.81 VectSharp.ImageSharpUtils.RasterImageStream Class Reference

A RasterImage created from a stream.

Inheritance diagram for VectSharp.ImageSharpUtils.RasterImageStream:

Public Member Functions

- RasterImageStream (Stream imageStream, bool alpha=true, bool interpolate=true)

  Creates a new RasterImage from the specified stream.
- RasterImageStream (IntPtr imageAddress, int imageLength, bool alpha=true, bool interpolate=true)

  Creates a new RasterImage from the specified stream.

Additional Inherited Members

7.81.1 Detailed Description

A RasterImage created from a stream.

Definition at line 106 of file RasterImages.cs.

7.81.2 Constructor & Destructor Documentation

7.81.2.1 RasterImageStream() [1/2]

```
VectSharp.ImageSharpUtils.RasterImageStream.RasterImageStream (
            Stream imageStream,
            bool alpha = true,
            bool interpolate = true )
```

Creates a new RasterImage from the specified stream.

Parameters

imageStream	The stream containing the image data.
alpha	A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.
interpolate	A boolean value indicating whether the image should be interpolated when it is resized or not.

Definition at line 114 of file RasterImages.cs.

7.81.2.2 RasterImageStream() [2/2]

```
VectSharp.ImageSharpUtils.RasterImageStream.RasterImageStream (
            IntPtr imageAddress,
            int imageLength,
            bool alpha = true,
            bool interpolate = true )
```

Creates a new RasterImage from the specified stream.

Parameters

imageAddress	A pointer to the address where the image data is contained.
imageLength	The length in bytes of the image data.
alpha	A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.
interpolate	A boolean value indicating whether the image should be interpolated when it is resized or not.

Definition at line 185 of file RasterImages.cs.

The documentation for this class was generated from the following file:

- VectSharp.ImageSharpUtils/RasterImages.cs

7.82 VectSharp.MuPDFUtils.RasterImageStream Class Reference

A RasterImage created from a stream.

Inheritance diagram for VectSharp.MuPDFUtils.RasterImageStream:

Public Member Functions

- RasterImageStream (Stream imageStream, InputFileTypes fileType, int pageNumber=0, double scale=1, bool alpha=true, bool interpolate=true)

  Creates a new RasterImage from the specified stream.
- RasterImageStream (IntPtr imageAddress, long imageLength, InputFileTypes fileType, int pageNumber=0, double scale=1, bool alpha=true, bool interpolate=true)

  Creates a new RasterImage from the specified stream.

Additional Inherited Members

7.82.1 Detailed Description

A RasterImage created from a stream.

Definition at line 69 of file RasterImages.cs.

7.82.2 Constructor & Destructor Documentation

7.82.2.1 RasterImageStream() [1/2]

```
VectSharp.MuPDFUtils.RasterImageStream.RasterImageStream (
            Stream imageStream,
            InputFileTypes fileType,
            int pageNumber = 0,
            double scale = 1,
            bool alpha = true,
            bool interpolate = true )
```

Creates a new RasterImage from the specified stream.

Parameters

imageStream	The stream containing the image data.
fileType	The type of the image contained in the stream.
pageNumber	The number of the page in the file from which the image should be created, starting at 0. Only useful for multi-page formats, such as PDF.
scale	The scale factor at which to render the image.
alpha	A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.
interpolate	A boolean value indicating whether the image should be interpolated when it is resized or not.

Definition at line 80 of file RasterImages.cs.

7.82.2.2 RasterImageStream() [2/2]

```
VectSharp.MuPDFUtils.RasterImageStream.RasterImageStream (
            IntPtr imageAddress,
            long imageLength,
            InputFileTypes fileType,
            int pageNumber = 0,
            double scale = 1,
            bool alpha = true,
            bool interpolate = true )
```

Creates a new RasterImage from the specified stream.

Parameters

imageAddress	A pointer to the address where the image data is contained.
imageLength	The length in bytes of the image data.
fileType	The type of the image contained in the stream.
pageNumber	The number of the page in the file from which the image should be created, starting at 0. Only useful for multi-page formats, such as PDF.
scale	The scale factor at which to render the image.
alpha	A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.
interpolate	A boolean value indicating whether the image should be interpolated when it is resized or not.

Definition at line 148 of file RasterImages.cs.

The documentation for this class was generated from the following file:

- VectSharp.MuPDFUtils/RasterImages.cs

7.83 VectSharp.Rectangle Struct Reference

Represents a rectangle.

Public Member Functions

- bool IsNaN ()

  Checks whether the rectangle is equivalent to Rectangle.NaN.
- Rectangle (Point location, Size size)

  Create a new Rectangle given its top-left corner and its size.
- Rectangle (double x, double y, double width, double height)

  Create a new Rectangle given its top-left corner and its size.
- Rectangle (Point topLeft, Point bottomRight)

  Create a new Rectangle given its top-left corner and its bottom-right corner.

Static Public Member Functions

- static Rectangle Union (Rectangle rectangle1, Rectangle rectangle2)

  Computes the rectangular bounds of the union of two Rectangles.
- static Rectangle Intersection (Rectangle rectangle1, Rectangle rectangle2)

  Computes the intersection of two Rectangles.
- static Rectangle Union (IEnumerable< Rectangle > rectangles)

  Computes the rectangular bounds of the union of multiple Rectangles.
- static Rectangle Union (params Rectangle[ ] rectangles)

  Computes the rectangular bounds of the union of multiple Rectangles.

Public Attributes

- Point Location

  The top-left corner of the rectangle.
- Size Size

  The size of the rectangle.

Static Public Attributes

- static readonly Rectangle NaN = new Rectangle(double.NaN, double.NaN, double.NaN, double.NaN)

  A rectangle whose dimensions are all double.NaN.

Properties

- Point Centre `[get]`

  The centre of the rectangle.

7.83.1 Detailed Description

Represents a rectangle.

Definition at line 172 of file Point.cs.

7.83.2 Constructor & Destructor Documentation

7.83.2.1 Rectangle() [1/3]

```
VectSharp.Rectangle.Rectangle (
            Point location,
            Size size )
```

Create a new Rectangle given its top-left corner and its size.

Parameters

location	The top-left corner of the rectangle.
size	The size of the rectangle.

Definition at line 208 of file Point.cs.

7.83.2.2 Rectangle() [2/3]

```
VectSharp.Rectangle.Rectangle (
            double x,
            double y,
            double width,
            double height )
```

Create a new Rectangle given its top-left corner and its size.

Parameters

x	The horizontal coordinate of the top-left corner of the rectangle.
y	The vertical coordinate of the top-left corner of the rectangle.
width	The width of the rectangle.
height	The height of the rectangle.

Definition at line 221 of file Point.cs.

7.83.2.3 Rectangle() [3/3]

```
VectSharp.Rectangle.Rectangle (
            Point topLeft,
            Point bottomRight )
```

Create a new Rectangle given its top-left corner and its bottom-right corner.

Parameters

topLeft	The top-left corner of the rectangle.
bottomRight	The bottom-right corner of the rectangle.

Definition at line 232 of file Point.cs.

7.83.3 Member Function Documentation

7.83.3.1 Intersection()

```
static Rectangle VectSharp.Rectangle.Intersection (
            Rectangle rectangle1,
            Rectangle rectangle2 )  [static]
```

Computes the intersection of two Rectangles.

Parameters

rectangle1	The first Rectangle.
rectangle2	The second Rectangle.

Returns

The rectangle corresponding to the intersection of *rectangle1* and *rectangle2* , or Rectangle.NaN if the intersection is empty.

Definition at line 277 of file Point.cs.

7.83.3.2 IsNaN()

```
bool VectSharp.Rectangle.IsNaN ( )
```

Checks whether the rectangle is equivalent to Rectangle.NaN.

Returns

`true` if all the dimensions of the rectangle are double.NaN, `false` otherwise.

Definition at line 183 of file Point.cs.

7.83.3.3 Union() [1/3]

```
static Rectangle VectSharp.Rectangle.Union (
            IEnumerable< Rectangle > rectangles )  [static]
```

Computes the rectangular bounds of the union of multiple Rectangles.

Parameters

rectangles	The Rectangles whose union will be computed.

Returns

The smallest Rectangle containing all the *rectangles* .

Definition at line 301 of file Point.cs.

7.83.3.4 Union() [2/3]

```
static Rectangle VectSharp.Rectangle.Union (
            params Rectangle[] rectangles )  [static]
```

Computes the rectangular bounds of the union of multiple Rectangles.

Parameters

rectangles	The Rectangles whose union will be computed.

Returns

The smallest Rectangle containing all the *rectangles* .

Definition at line 334 of file Point.cs.

7.83.3.5 Union() [3/3]

```
static Rectangle VectSharp.Rectangle.Union (
            Rectangle rectangle1,
            Rectangle rectangle2 )  [static]
```

Computes the rectangular bounds of the union of two Rectangles.

Parameters

rectangle1	The first Rectangle.
rectangle2	The second Rectangle.

Returns

The smallest Rectangle containing both *rectangle1* and *rectangle2* .

Definition at line 244 of file Point.cs.

7.83.4 Member Data Documentation

7.83.4.1 Location

```
Point VectSharp.Rectangle.Location
```

The top-left corner of the rectangle.

Definition at line 191 of file Point.cs.

7.83.4.2 NaN

```
readonly Rectangle VectSharp.Rectangle.NaN = new Rectangle(double.NaN, double.NaN, double.NaN,
double.NaN)  [static]
```

A rectangle whose dimensions are all double.NaN.

Definition at line 177 of file Point.cs.

7.83.4.3 Size

```
Size VectSharp.Rectangle.Size
```

The size of the rectangle.

Definition at line 196 of file Point.cs.

7.83.5 Property Documentation

7.83.5.1 Centre

```
Point VectSharp.Rectangle.Centre  [get]
```

The centre of the rectangle.

Definition at line 201 of file Point.cs.

The documentation for this struct was generated from the following file:

- VectSharp/Point.cs

7.84 VectSharp.Canvas.RenderAction Class Reference

Represents a light-weight rendering action.

Public Types

- enum ActionTypes

  Types of rendering actions.

Public Member Functions

- void BringToFront ()

  Brings the render action to the front of the rendering queue. This method can only be invoked after the output has been fully initialised.
- void SendToBack ()

  Brings the render action to the back of the rendering queue. This method can only be invoked after the output has been fully initialised.

Static Public Member Functions

- static RenderAction PathAction (Geometry geometry, Pen stroke, IBrush fill, Avalonia.Matrix transform, Geometry clippingPath, string tag=null)

  Creates a new RenderAction representing a path.
- static RenderAction TextAction (Avalonia.Media.FormattedText text, IBrush fill, Avalonia.Matrix transform, Geometry clippingPath, string tag=null)

  Creates a new RenderAction representing text.
- static RenderAction ImageAction (string imageId, Avalonia.Rect sourceRect, Avalonia.Rect destinationRect, Avalonia.Matrix transform, Geometry clippingPath, string tag=null)

  Creates a new RenderAction representing an image.

Properties

- ActionTypes ActionType `[get]`

  Type of the rendering action.
- Geometry Geometry `[get, set]`

  Geometry that needs to be rendered (null if the action type is ActionTypes.Text). If you change this, you need to invalidate the Parent's visual.
- Avalonia.Media.FormattedText Text `[get, set]`

  Text that needs to be rendered (null if the action type is ActionTypes.Path). If you change this, you need to invalidate the Parent's visual.
- Pen Stroke `[get, set]`

  Rendering stroke (null if the action type is ActionTypes.Text or if the rendered action only has a Fill). If you change this, you need to invalidate the Parent's visual.
- IBrush Fill `[get, set]`

  Rendering fill (null if the rendered action only has a Stroke). If you change this, you need to invalidate the Parent's visual.
- string ImageId `[get, set]`

  Univocal identifier of the image that needs to be drawn.
- Avalonia.? Rect ImageSource `[get, set]`

  The source rectangle of the image.
- Avalonia.? Rect ImageDestination `[get, set]`

  The destination rectangle of the image.
- Geometry ClippingPath `[get, set]`

  The current clipping path.

- Avalonia.Matrix InverseTransform = Avalonia.Matrix.Identity `[get]`

  Inverse transformation matrix.
- Avalonia.Matrix Transform `[get, set]`

  Rendering transformation matrix. If you change this, you need to invalidate the Parent's visual.
- string Tag `[get, set]`

  A tag to access the RenderAction.
- Avalonia.Controls.Canvas Parent `[get]`

  The container of this RenderAction.

Events

- EventHandler< Avalonia.Input.PointerEventArgs > PointerEnter

  Raised when the pointer enters the area covered by the RenderAction.
- EventHandler< Avalonia.Input.PointerEventArgs > PointerLeave

  Raised when the pointer leaves the area covered by the RenderAction.
- EventHandler< Avalonia.Input.PointerPressedEventArgs > PointerPressed

  Raised when the pointer is pressed while over the area covered by the RenderAction.
- EventHandler< Avalonia.Input.PointerReleasedEventArgs > PointerReleased

  Raised when the pointer is released after a PointerPressed event.

7.84.1 Detailed Description

Represents a light-weight rendering action.

Definition at line 1290 of file AvaloniaContext.cs.

7.84.2 Member Enumeration Documentation

7.84.2.1 ActionTypes

```
enum VectSharp.Canvas.RenderAction.ActionTypes
```

Types of rendering actions.

Definition at line 1295 of file AvaloniaContext.cs.

7.84.3 Member Function Documentation

7.84.3.1 BringToFront()

```
void VectSharp.Canvas.RenderAction.BringToFront ( )
```

Brings the render action to the front of the rendering queue. This method can only be invoked after the output has been fully initialised.

Definition at line 1516 of file AvaloniaContext.cs.

7.84.3.2 ImageAction()

```
static RenderAction VectSharp.Canvas.RenderAction.ImageAction (
            string imageId,
            Avalonia.Rect sourceRect,
            Avalonia.Rect destinationRect,
            Avalonia.Matrix transform,
            Geometry clippingPath,
            string tag = null )  [static]
```

Creates a new RenderAction representing an image.

Parameters

imageId	The univocal identifier of the image to draw.
sourceRect	The source rectangle of the image.
destinationRect	The destination rectangle of the image.
transform	The transform that will be applied to the image.
clippingPath	The clipping path.
tag	A tag to access the RenderAction. If this is null this RenderAction is not visible in the hit test.

Returns

A new RenderAction representing an image.

Definition at line 1499 of file AvaloniaContext.cs.

7.84.3.3 PathAction()

```
static RenderAction VectSharp.Canvas.RenderAction.PathAction (
            Geometry geometry,
            Pen stroke,
            IBrush fill,
            Avalonia.Matrix transform,
            Geometry clippingPath,
            string tag = null )  [static]
```

Creates a new RenderAction representing a path.

Parameters

geometry	The geometry to be rendered.
stroke	The stroke of the path (can be null).
fill	The fill of the path (can be null).
transform	The transform that will be applied to the path.
clippingPath	The clipping path.
tag	A tag to access the RenderAction. If this is null this RenderAction is not visible in the hit test.

Returns

A new RenderAction representing a path.

Definition at line 1452 of file AvaloniaContext.cs.

7.84.3.4 SendToBack()

```
void VectSharp.Canvas.RenderAction.SendToBack ( )
```

Brings the render action to the back of the rendering queue. This method can only be invoked after the output has been fully initialised.

Definition at line 1524 of file AvaloniaContext.cs.

7.84.3.5 TextAction()

```
static RenderAction VectSharp.Canvas.RenderAction.TextAction (
            Avalonia.Media.FormattedText text,
            IBrush fill,
            Avalonia.Matrix transform,
            Geometry clippingPath,
            string tag = null )  [static]
```

Creates a new RenderAction representing text.

Parameters

text	The text to be rendered.
fill	The fill of the text (can be null).
transform	The transform that will be applied to the text.
clippingPath	The clipping path.
tag	A tag to access the RenderAction. If this is null this RenderAction is not visible in the hit test.

Returns

A new RenderAction representing text.

Definition at line 1475 of file AvaloniaContext.cs.

7.84.4 Property Documentation

7.84.4.1 ActionType

```
ActionTypes VectSharp.Canvas.RenderAction.ActionType [get]
```

Type of the rendering action.

Definition at line 1316 of file AvaloniaContext.cs.

7.84.4.2 ClippingPath

```
Geometry VectSharp.Canvas.RenderAction.ClippingPath [get], [set]
```

The current clipping path.

Definition at line 1356 of file AvaloniaContext.cs.

7.84.4.3 Fill

```
IBrush VectSharp.Canvas.RenderAction.Fill [get], [set]
```

Rendering fill (null if the rendered action only has a Stroke). If you change this, you need to invalidate the Parent's visual.

Definition at line 1336 of file AvaloniaContext.cs.

7.84.4.4 Geometry

```
Geometry VectSharp.Canvas.RenderAction.Geometry [get], [set]
```

Geometry that needs to be rendered (null if the action type is ActionTypes.Text). If you change this, you need to invalidate the Parent's visual.

Definition at line 1321 of file AvaloniaContext.cs.

7.84.4.5 ImageDestination

```
Avalonia.?  Rect VectSharp.Canvas.RenderAction.ImageDestination  [get], [set]
```

The destination rectangle of the image.

Definition at line 1351 of file AvaloniaContext.cs.

7.84.4.6 ImageId

```
string VectSharp.Canvas.RenderAction.ImageId  [get], [set]
```

Univocal identifier of the image that needs to be drawn.

Definition at line 1341 of file AvaloniaContext.cs.

7.84.4.7 ImageSource

```
Avalonia.?  Rect VectSharp.Canvas.RenderAction.ImageSource  [get], [set]
```

The source rectangle of the image.

Definition at line 1346 of file AvaloniaContext.cs.

7.84.4.8 InverseTransform

```
Avalonia.Matrix VectSharp.Canvas.RenderAction.InverseTransform = Avalonia.Matrix.Identity
[get]
```

Inverse transformation matrix.

Definition at line 1363 of file AvaloniaContext.cs.

7.84.4.9 Parent

```
Avalonia.Controls.Canvas VectSharp.Canvas.RenderAction.Parent  [get]
```

The container of this RenderAction.

Definition at line 1388 of file AvaloniaContext.cs.

7.84.4.10 Stroke

```
Pen VectSharp.Canvas.RenderAction.Stroke  [get], [set]
```

Rendering stroke (null if the action type is ActionTypes.Text or if the rendered action only has a Fill). If you change this, you need to invalidate the Parent's visual.

Definition at line 1331 of file AvaloniaContext.cs.

7.84.4.11 Tag

```
string VectSharp.Canvas.RenderAction.Tag  [get], [set]
```

A tag to access the RenderAction.

Definition at line 1381 of file AvaloniaContext.cs.

7.84.4.12 Text

```
Avalonia.Media.FormattedText VectSharp.Canvas.RenderAction.Text  [get], [set]
```

Text that needs to be rendered (null if the action type is ActionTypes.Path). If you change this, you need to invalidate the Parent's visual.

Definition at line 1326 of file AvaloniaContext.cs.

7.84.4.13 Transform

```
Avalonia.Matrix VectSharp.Canvas.RenderAction.Transform  [get], [set]
```

Rendering transformation matrix. If you change this, you need to invalidate the Parent's visual.

Definition at line 1368 of file AvaloniaContext.cs.

7.84.5 Event Documentation

7.84.5.1 PointerEnter

```
EventHandler<Avalonia.Input.PointerEventArgs> VectSharp.Canvas.RenderAction.PointerEnter
```

Raised when the pointer enters the area covered by the RenderAction.

Definition at line 1399 of file AvaloniaContext.cs.

7.84.5.2 PointerLeave

```
EventHandler<Avalonia.Input.PointerEventArgs> VectSharp.Canvas.RenderAction.PointerLeave
```

Raised when the pointer leaves the area covered by the RenderAction.

Definition at line 1404 of file AvaloniaContext.cs.

7.84.5.3 PointerPressed

```
EventHandler<Avalonia.Input.PointerPressedEventArgs> VectSharp.Canvas.RenderAction.Pointer↩
Pressed
```

Raised when the pointer is pressed while over the area covered by the RenderAction.

Definition at line 1409 of file AvaloniaContext.cs.

7.84.5.4 PointerReleased

```
EventHandler<Avalonia.Input.PointerReleasedEventArgs> VectSharp.Canvas.RenderAction.Pointer↩
Released
```

Raised when the pointer is released after a PointerPressed event.

Definition at line 1414 of file AvaloniaContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.Canvas/AvaloniaContext.cs

7.85 VectSharp.ResourceFontFamily Class Reference

Represents a FontFamily created from a resource stream.

Inheritance diagram for VectSharp.ResourceFontFamily:

Public Member Functions

- ResourceFontFamily (System.IO.Stream resourceStream, string resourceName)

  Create a new ResourceFontFamily from the specified resourceStream containing a TTF file, passing the specified resourceName to the

Public Attributes

- string ResourceName

  The name of the embedded resource, which will be parsed using

Additional Inherited Members

7.85.1 Detailed Description

Represents a FontFamily created from a resource stream.

Definition at line 694 of file Font.cs.

7.85.2 Constructor & Destructor Documentation

7.85.2.1 ResourceFontFamily()

```
VectSharp.ResourceFontFamily.ResourceFontFamily (
            System.IO.Stream resourceStream,
            string resourceName )
```

Create a new ResourceFontFamily from the specified *resourceStream* containing a TTF file, passing the specified *resourceName* to the

```
Avalonia.Media.FontFamily.Parse(string, Uri)"</code> method.  </summary>
<param name="resourceStream">A resource stream containing a TTF file.</param>
<param name="resourceName">The name of the embedded resource, which will be
parsed using Avalonia.Media.FontFamily.Parse(string, Uri).
```

Definition at line 706 of file Font.cs.

7.85.3 Member Data Documentation

7.85.3.1 ResourceName

```
string VectSharp.ResourceFontFamily.ResourceName
```

The name of the embedded resource, which will be parsed using

`Avalonia.Media.FontFamily.Parse(string, Uri)`.

Definition at line 699 of file Font.cs.

The documentation for this class was generated from the following file:

- VectSharp/Font.cs

7.86 VectSharp.ThreeD.Scene Class Reference

Represents a 3D scene.

Inheritance diagram for VectSharp.ThreeD.Scene:

Public Member Functions

- Scene ()

  Creates a new Scene.
- void AddElement (Element3D element)

  Adds the specified element to the scene.

Parameters

element	*The Element3D to add.*

- void AddRange (IEnumerable< Element3D > elements)

  Adds the specified elements to the scene.

Parameters

elements	*A collection of Element3Ds to add.*

- void Replace (Func< Element3D, Element3D > replacementFunction)

  Replaces each element in the scene with the element returned by the replacementFunction .

Parameters

replacementFunction	*A function replacing each Element3D in the scene with another Element3D.*

- void Replace (Func< Element3D, IEnumerable< Element3D > > replacementFunction)

  Replaces each element in the scene with the element(s) returned by the replacementFunction .

Parameters

replacementFunction	*A function replacing each Element3D in the scene with 0 or more Element3Ds.*

Properties

- object SceneLock `[get]`

  An object used to synchronise multithreaded rendering of the same scene.
- IEnumerable< Element3D > SceneElements `[get]`

  The Element3Ds constituting the scene.

7.86.1 Detailed Description

Represents a 3D scene.

Definition at line 66 of file Scene.cs.

7.86.2 Constructor & Destructor Documentation

7.86.2.1 Scene()

```
VectSharp.ThreeD.Scene.Scene ( )
```

Creates a new Scene.

Definition at line 79 of file Scene.cs.

7.86.3 Member Function Documentation

7.86.3.1 AddElement()

```
void VectSharp.ThreeD.Scene.AddElement (
            Element3D element )
```

Adds the specified *element* to the scene.

Parameters

element	The Element3D to add.

Implements VectSharp.ThreeD.IScene.

Definition at line 86 of file Scene.cs.

7.86.3.2 AddRange()

```
void VectSharp.ThreeD.Scene.AddRange (
            IEnumerable< Element3D > elements )
```

Adds the specified *elements* to the scene.

Parameters

elements	A collection of Element3Ds to add.

Implements VectSharp.ThreeD.IScene.

Definition at line 92 of file Scene.cs.

7.86.3.3 Replace() [1/2]

```
void VectSharp.ThreeD.Scene.Replace (
            Func< Element3D, Element3D > replacementFunction )
```

Replaces each element in the scene with the element returned by the *replacementFunction* .

Parameters

replacementFunction	A function replacing each Element3D in the scene with another Element3D.

Implements VectSharp.ThreeD.IScene.

Definition at line 98 of file Scene.cs.

7.86.3.4 Replace() [2/2]

```
void VectSharp.ThreeD.Scene.Replace (
            Func< Element3D, IEnumerable< Element3D > > replacementFunction )
```

Replaces each element in the scene with the element(s) returned by the *replacementFunction* .

Parameters

replacementFunction	A function replacing each Element3D in the scene with 0 or more Element3Ds.

Implements VectSharp.ThreeD.IScene.

Definition at line 107 of file Scene.cs.

7.86.4 Property Documentation

7.86.4.1 SceneElements

```
IEnumerable<Element3D> VectSharp.ThreeD.Scene.SceneElements  [get]
```

The Element3Ds constituting the scene.

Implements VectSharp.ThreeD.IScene.

Definition at line 74 of file Scene.cs.

7.86.4.2 SceneLock

```
object VectSharp.ThreeD.Scene.SceneLock  [get]
```

An object used to synchronise multithreaded rendering of the same scene.

Implements VectSharp.ThreeD.IScene.

Definition at line 69 of file Scene.cs.

The documentation for this class was generated from the following file:

- VectSharp.ThreeD/Scene.cs

7.87 VectSharp.Segment Class Reference

Represents a segment as part of a GraphicsPath.

Public Member Functions

- abstract Segment Clone ()

  Creates a copy of the Segment.
- abstract double Measure (Point previousPoint)

  Computes the length of the Segment.
- abstract Point GetPointAt (Point previousPoint, double position)

  Gets the point on the Segment at the specified (relative) position).
- abstract Point GetTangentAt (Point previousPoint, double position)

  Gets the tangent to the Segment at the specified (relative) position).
- abstract IEnumerable< Segment > Linearise (Point? previousPoint, double resolution)

  Transform the segment into a series of linear segments. Segments that are already linear are not changed.
- abstract IEnumerable< Point > GetLinearisationTangents (Point? previousPoint, double resolution)

  Gets the tanget at the points at which the segment would be linearised.
- abstract IEnumerable< Segment > Transform (Func< Point, Point > transformationFunction)

  Applies an arbitrary transformation to all of the points of the Segment.

Properties

- abstract SegmentType Type `[get]`

  The type of the Segment.
- Point[] Points `[get]`

  The points used to define the Segment.
- virtual Point Point `[get]`

  The end point of the Segment.

7.87.1 Detailed Description

Represents a segment as part of a GraphicsPath.

Definition at line 28 of file Segment.cs.

7.87.2 Member Function Documentation

7.87.2.1 Clone()

```
abstract Segment VectSharp.Segment.Clone ( ) [pure virtual]
```

Creates a copy of the Segment.

Returns

A copy of the Segment.

7.87.2.2 GetLinearisationTangents()

```
abstract IEnumerable< Point > VectSharp.Segment.GetLinearisationTangents (
            Point? previousPoint,
            double resolution ) [pure virtual]
```

Gets the tanget at the points at which the segment would be linearised.

Parameters

previousPoint	The point from which the Segment starts (i.e. the endpoint of the previous Segment).
resolution	The absolute length between successive samples in curve segments.

Returns

A collection of tangents at the points in which the segment would be linearised.

7.87.2.3 GetPointAt()

```
abstract Point VectSharp.Segment.GetPointAt (
            Point previousPoint,
            double position )  [pure virtual]
```

Gets the point on the Segment at the specified (relative) *position*).

Parameters

previousPoint	The point from which the Segment starts (i.e. the endpoint of the previous Segment).
position	The relative position on the Segment (0 is the start of the Segment, 1 is the end of the Segment).

Returns

The point at the specified position.

7.87.2.4 GetTangentAt()

```
abstract Point VectSharp.Segment.GetTangentAt (
            Point previousPoint,
            double position )  [pure virtual]
```

Gets the tangent to the Segment at the specified (relative) *position*).

Parameters

previousPoint	The point from which the Segment starts (i.e. the endpoint of the previous Segment).
position	The relative position on the Segment (0 is the start of the Segment, 1 is the end of the Segment).

Returns

The tangent to the point at the specified position.

7.87.2.5 Linearise()

```
abstract IEnumerable< Segment > VectSharp.Segment.Linearise (
            Point?  previousPoint,
            double resolution )  [pure virtual]
```

Transform the segment into a series of linear segments. Segments that are already linear are not changed.

Parameters

previousPoint	The point from which the Segment starts (i.e. the endpoint of the previous Segment).
resolution	The absolute length between successive samples in curve segments.

Returns

A collection of linear segments that approximate the current segment.

7.87.2.6 Measure()

```
abstract double VectSharp.Segment.Measure (
            Point previousPoint )  [pure virtual]
```

Computes the length of the Segment.

Parameters

previousPoint	The point from which the Segment starts (i.e. the endpoint of the previous Segment).

Returns

The length of the segment.

7.87.2.7 Transform()

```
abstract IEnumerable< Segment > VectSharp.Segment.Transform (
            Func< Point, Point > transformationFunction )  [pure virtual]
```

Applies an arbitrary transformation to all of the points of the Segment.

Parameters

transformationFunction	An arbitrary transformation function.

Returns

A collection of Segments that have been transformed according to the *transformationFunction* .

7.87.3 Property Documentation

7.87.3.1 Point

```
virtual Point VectSharp.Segment.Point  [get]
```

The end point of the Segment.

Definition at line 44 of file Segment.cs.

7.87.3.2 Points

```
Point [] VectSharp.Segment.Points  [get]
```

The points used to define the Segment.

Definition at line 39 of file Segment.cs.

7.87.3.3 Type

```
abstract SegmentType VectSharp.Segment.Type  [get]
```

The type of the Segment.

Definition at line 34 of file Segment.cs.

The documentation for this class was generated from the following file:

- VectSharp/Segment.cs

7.88 VectSharp.SimpleFontLibrary Class Reference

A font library that can be used to cache and resolve font family names.

Inheritance diagram for VectSharp.SimpleFontLibrary:

Public Member Functions

- SimpleFontLibrary (IFontLibrary standardFontLibrary)

  Create a new SimpleFontLibrary instance.
- SimpleFontLibrary ()

  Create a new SimpleFontLibrary instance, using the default font library to resolve the standard font families.
- SimpleFontLibrary (FontFamily timesRoman, FontFamily timesBold, FontFamily timesItalic, FontFamily timesBoldItalic, FontFamily helvetica, FontFamily helveticaBold, FontFamily helveticaOblique, FontFamily helveticaBoldOblique, FontFamily courier, FontFamily courierBold, FontFamily courierOblique, FontFamily courierBoldOblique, FontFamily symbol, FontFamily zapfdingbats)

  Create a new SimpleFontLibrary instance, with the specified replacements for the standard font families.
- SimpleFontLibrary (string timesRoman, string timesBold, string timesItalic, string timesBoldItalic, string helvetica, string helveticaBold, string helveticaOblique, string helveticaBoldOblique, string courier, string courierBold, string courierOblique, string courierBoldOblique, string symbol, string zapfdingbats)

  Create a new SimpleFontLibrary instance, with the specified replacements for the standard font families.
- void Add (string fontFamilyName, FontFamily fontFamily)

  Add the specified font family to the library with the specified name.
- void Add (FontFamily fontFamily)

  Add the specified font family to the library.
- void Add (string fileName)

  Add the font family contained in the specified True Type Font file to the library.
- void Add (string fontFamily, string fileName)

  Add the font family contained in the specified True Type Font file to the library, with the specified name. The font family is not loaded until it is requested for the first time.
- override FontFamily ResolveFontFamily (FontFamily.StandardFontFamilies standardFontFamily)

  Create a new font family from the specified standard font family name.

Parameters

standardFontFamily	*The standard name of the font family.*

Returns

A FontFamily object corresponding to the specified font family.

- override FontFamily ResolveFontFamily (string fontFamily)

  Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised.

Parameters

fontFamily	*The name of the font family to create, or the path to a TTF file.*

Returns

If the font family name or the true type file is valid, a FontFamily object corresponding to the specified font family.

7.88.1 Detailed Description

A font library that can be used to cache and resolve font family names.

Definition at line 186 of file FontLibrary.cs.

7.88.2 Constructor & Destructor Documentation

7.88.2.1 SimpleFontLibrary() [1/4]

```
VectSharp.SimpleFontLibrary.SimpleFontLibrary (
            IFontLibrary standardFontLibrary )
```

Create a new SimpleFontLibrary instance.

Parameters

standardFontLibrary	An existing font library that will be used to resolve the standard font families.

Definition at line 198 of file FontLibrary.cs.

7.88.2.2 SimpleFontLibrary() [2/4]

```
VectSharp.SimpleFontLibrary.SimpleFontLibrary ( )
```

Create a new SimpleFontLibrary instance, using the default font library to resolve the standard font families.

Definition at line 218 of file FontLibrary.cs.

7.88.2.3 SimpleFontLibrary() [3/4]

```
VectSharp.SimpleFontLibrary.SimpleFontLibrary (
            FontFamily timesRoman,
            FontFamily timesBold,
            FontFamily timesItalic,
            FontFamily timesBoldItalic,
            FontFamily helvetica,
            FontFamily helveticaBold,
            FontFamily helveticaOblique,
            FontFamily helveticaBoldOblique,
            FontFamily courier,
            FontFamily courierBold,
            FontFamily courierOblique,
            FontFamily courierBoldOblique,
            FontFamily symbol,
            FontFamily zapfdingbats )
```

Create a new SimpleFontLibrary instance, with the specified replacements for the standard font families.

Parameters

timesRoman	The font family to use for the Times-Roman standard font.
timesBold	The font family to use for the Times-Bold standard font.
timesItalic	The font family to use for the Times-Italic standard font.
timesBoldItalic	The font family to use for the Times-BoldItalic standard font.
helvetica	The font family to use for the Helvetica standard font.
helveticaBold	The font family to use for the Helvetica-Bold standard font.
helveticaOblique	The font family to use for the Helvetica-Oblique standard font.
helveticaBoldOblique	The font family to use for the Helvetica-BoldOblique standard font.
courier	The font family to use for the Courier standard font.
courierBold	The font family to use for the Courier-Bold standard font.
courierOblique	The font family to use for the Courier-Oblique standard font.
courierBoldOblique	The font family to use for the Courier-BoldOblique standard font.
symbol	The font family to use for the Symbol standard font.
zapfdingbats	The font family to use for the Zapfdingbats standard font.

Definition at line 240 of file FontLibrary.cs.

7.88.2.4 SimpleFontLibrary() [4/4]

```
VectSharp.SimpleFontLibrary.SimpleFontLibrary (
            string timesRoman,
            string timesBold,
            string timesItalic,
            string timesBoldItalic,
            string helvetica,
            string helveticaBold,
            string helveticaOblique,
            string helveticaBoldOblique,
```

```
string courier,
string courierBold,
string courierOblique,
string courierBoldOblique,
string symbol,
string zapfdingbats )
```

Create a new SimpleFontLibrary instance, with the specified replacements for the standard font families.

Parameters

timesRoman	The font family to use for the Times-Roman standard font.
timesBold	The font family to use for the Times-Bold standard font.
timesItalic	The font family to use for the Times-Italic standard font.
timesBoldItalic	The font family to use for the Times-BoldItalic standard font.
helvetica	The font family to use for the Helvetica standard font.
helveticaBold	The font family to use for the Helvetica-Bold standard font.
helveticaOblique	The font family to use for the Helvetica-Oblique standard font.
helveticaBoldOblique	The font family to use for the Helvetica-BoldOblique standard font.
courier	The font family to use for the Courier standard font.
courierBold	The font family to use for the Courier-Bold standard font.
courierOblique	The font family to use for the Courier-Oblique standard font.
courierBoldOblique	The font family to use for the Courier-BoldOblique standard font.
symbol	The font family to use for the Symbol standard font.
zapfdingbats	The font family to use for the Zapfdingbats standard font.

Definition at line 298 of file FontLibrary.cs.

7.88.3 Member Function Documentation

7.88.3.1 Add() [1/4]

```
void VectSharp.SimpleFontLibrary.Add (
            FontFamily fontFamily )
```

Add the specified font family to the library.

Parameters

fontFamily	The font family to add.

Definition at line 361 of file FontLibrary.cs.

7.88.3.2 Add() [2/4]

```
void VectSharp.SimpleFontLibrary.Add (
            string fileName )
```

Add the font family contained in the specified True Type Font file to the library.

Parameters

fileName	The path to the TTF file containing the font family.

Definition at line 373 of file FontLibrary.cs.

7.88.3.3 Add() [3/4]

```
void VectSharp.SimpleFontLibrary.Add (
            string fontFamily,
            string fileName )
```

Add the font family contained in the specified True Type Font file to the library, with the specified name. The font family is not loaded until it is requested for the first time.

Parameters

fontFamily	The name of the font family.
fileName	The path to the TTF file containing the font family.

Definition at line 388 of file FontLibrary.cs.

7.88.3.4 Add() [4/4]

```
void VectSharp.SimpleFontLibrary.Add (
            string fontFamilyName,
            FontFamily fontFamily )
```

Add the specified font family to the library with the specified name.

Parameters

fontFamilyName	The name of the font family.
fontFamily	The font family to add.

Definition at line 344 of file FontLibrary.cs.

7.88.3.5 ResolveFontFamily() [1/2]

```
override FontFamily VectSharp.SimpleFontLibrary.ResolveFontFamily (
            FontFamily.StandardFontFamilies standardFontFamily )  [virtual]
```

Create a new font family from the specified standard font family name.

Parameters

standardFontFamily	The standard name of the font family.

Returns

A FontFamily object corresponding to the specified font family.

Implements VectSharp.FontLibrary.

Definition at line 395 of file FontLibrary.cs.

7.88.3.6 ResolveFontFamily() [2/2]

```
override FontFamily VectSharp.SimpleFontLibrary.ResolveFontFamily (
            string fontFamily )  [virtual]
```

Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised.

Parameters

fontFamily	The name of the font family to create, or the path to a TTF file.

Returns

If the font family name or the true type file is valid, a FontFamily object corresponding to the specified font family.

Implements VectSharp.FontLibrary.

Definition at line 401 of file FontLibrary.cs.

The documentation for this class was generated from the following file:

- VectSharp/FontLibrary.cs

7.89 VectSharp.Size Struct Reference

Represents the size of an object.

Public Member Functions

- Size (double width, double height)

  Create a new Size.

Public Attributes

- double Width

  Width of the object.
- double Height

  Height of the object.

7.89.1 Detailed Description

Represents the size of an object.

Definition at line 145 of file Point.cs.

7.89.2 Constructor & Destructor Documentation

7.89.2.1 Size()

```
VectSharp.Size.Size (
            double width,
            double height )
```

Create a new Size.

Parameters

width	The width of the object.
height	The height of the object.

Definition at line 162 of file Point.cs.

7.89.3 Member Data Documentation

7.89.3.1 Height

```
double VectSharp.Size.Height
```

Height of the object.

Definition at line 155 of file Point.cs.

7.89.3.2 Width

```
double VectSharp.Size.Width
```

Width of the object.

Definition at line 150 of file Point.cs.

The documentation for this struct was generated from the following file:

- VectSharp/Point.cs

7.90 VectSharp.Canvas.SKMultiLayerRenderCanvas Class Reference

Represents a multi-threaded, triple-buffered canvas on which the image is drawn using SkiaSharp.

Inheritance diagram for VectSharp.Canvas.SKMultiLayerRenderCanvas:

Public Member Functions

- SKMultiLayerRenderCanvas (Document document, Colour backgroundColour, double width, double height, List< SKRenderAction > layerTransforms=null)

  Create a new SKMultiLayerRenderCanvas from a Document, where each page represents a layer.
- SKMultiLayerRenderCanvas (IEnumerable< Page > layers, Colour backgroundColour, double width, double height, List< SKRenderAction > layerTransforms=null)

  Create a new SKMultiLayerRenderCanvas from a collection of Pages, each representing a layer.
- SKMultiLayerRenderCanvas (List< SKRenderContext > contents, List< SKRenderAction > content↩ Transforms, Colour backgroundColour, double width, double height)

  Create a new SKMultiLayerRenderCanvas from a list of SKRenderContexts, each representing a layer.

- void UpdateWith (List< SKRenderContext > contents, List< SKRenderAction > contentTransforms, Colour backgroundColour, double width, double height)

  Replace the contents of the SKMultiLayerRenderCanvas with the specified layers.
- void UpdateLayer (int layer, SKRenderContext newContent, SKRenderAction newTransform)

  Replace a single layer with the specified content.
- void AddLayer (SKRenderContext newContent, SKRenderAction newTransform)

  Add a new layer to the image.
- void InsertLayer (int index, SKRenderContext newContent, SKRenderAction newTransform)

  Insert a new layer at the specified index.
- void RemoveLayer (int layer)

  Remove the specified layer from the image.
- void SwitchLayers (int layer1, int layer2)

  Switch the position of the two specified layers.
- void MoveLayer (int oldIndex, int newIndex)

  Move the specified layer to the specified position, shifting all other layers as necessary.
- RenderTargetBitmap RenderAtResolution (int width, int height, SKColor? background=null)

  Render the image at to a bitmap at the specified resolution.
- void InvalidateDirty ()

  Invalidate the contents of the canvas, forcing it to redraw itself.
- void InvalidateZIndex ()

  Invalidate the contents of the canvas, specifying that the order of the layers has changed.
- override void Render (DrawingContext context)
- void Dispose ()

Public Attributes

- List< List< SKRenderAction > > RenderActions

  The list of render actions, each element in this list is itself a list, containing the actions that correspond to a layer in the image.
- List< SKRenderAction > LayerTransforms

  The list of transforms associated with each layer.
- object RenderLock = new object()

  An lock for the rendering loop. The public methods of this class already lock on this, but you may need it if you want to directly manipulate the contents of the canvas.

Properties

- double PageWidth `[get, set]`

  The width of the page that is rendered on this canvas.
- double PageHeight `[get, set]`

  The height of the page that is rendered on this canvas.
- Func< long, Bitmap > Spinner = null `[get, set]`

  If the image to draw is not already cached, this method is called with an argument containing the number of milliseconds since the image was last rendered. The method can return a Bitmap that will be drawn on the canvas in order to let users know that the image is being rendered in background.

7.90.1 Detailed Description

Represents a multi-threaded, triple-buffered canvas on which the image is drawn using SkiaSharp.

Definition at line 43 of file SKMultiLayerRenderCanvas.cs.

7.90.2 Constructor & Destructor Documentation

7.90.2.1 SKMultiLayerRenderCanvas() [1/3]

```
VectSharp.Canvas.SKMultiLayerRenderCanvas.SKMultiLayerRenderCanvas (
            Document document,
            Colour backgroundColour,
            double width,
            double height,
            List< SKRenderAction > layerTransforms = null )
```

Create a new SKMultiLayerRenderCanvas from a Document, where each page represents a layer.

Parameters

document	The document containing the layers as Pages.
layerTransforms	A list of transforms associated with each layer. This list should contain the same number of elements as the number of pages in *document* . This is useful to manipulate the position of each layer individually. If this is null, an identity transform is applied to each layer.
backgroundColour	The background colour of the canvas.
width	The width of the canvas and the pages it contains.
height	The height of the canvas and the pages it contains.

Definition at line 98 of file SKMultiLayerRenderCanvas.cs.

7.90.2.2 SKMultiLayerRenderCanvas() [2/3]

```
VectSharp.Canvas.SKMultiLayerRenderCanvas.SKMultiLayerRenderCanvas (
            IEnumerable< Page > layers,
            Colour backgroundColour,
            double width,
            double height,
            List< SKRenderAction > layerTransforms = null )
```

Create a new SKMultiLayerRenderCanvas from a collection of Pages, each representing a layer.

Parameters

layers	The contents of the canvas. Each element in this list represents a layer.
layerTransforms	A list of transforms associated with each layer. This list should contain the same number of elements as *layers* . This is useful to manipulate the position of each layer individually. If this is null, an identity transform is applied to each layer.
backgroundColour	The background colour of the canvas.
width	The width of the canvas and the pages it contains.
height	The height of the canvas and the pages it contains.

Definition at line 108 of file SKMultiLayerRenderCanvas.cs.

7.90.2.3 SKMultiLayerRenderCanvas() [3/3]

```
VectSharp.Canvas.SKMultiLayerRenderCanvas.SKMultiLayerRenderCanvas (
            List< SKRenderContext > contents,
            List< SKRenderAction > contentTransforms,
            Colour backgroundColour,
            double width,
            double height )
```

Create a new SKMultiLayerRenderCanvas from a list of SKRenderContexts, each representing a layer.

Parameters

contents	The contents of the canvas. Each element in this list represents a layer. A Page can be converded to a SKRenderContext through the CopyToSKRenderContext method.
contentTransforms	A list of transforms associated with each layer. This list should contain the same number of elements as *contents* . This is useful to manipulate the position of each layer individually.
backgroundColour	The background colour of the canvas.
width	The width of the canvas and the page it contains.
height	The height of the canvas and the page it contains.

Definition at line 143 of file SKMultiLayerRenderCanvas.cs.

7.90.3 Member Function Documentation

7.90.3.1 AddLayer()

```
void VectSharp.Canvas.SKMultiLayerRenderCanvas.AddLayer (
            SKRenderContext newContent,
            SKRenderAction newTransform )
```

Add a new layer to the image.

Parameters

newContent	The contents of the new layer. A Page can be converded to a SKRenderContext through the CopyToSKRenderContext method.
newTransform	The transform for the new layer.

Definition at line 276 of file SKMultiLayerRenderCanvas.cs.

7.90.3.2 Dispose()

```
void VectSharp.Canvas.SKMultiLayerRenderCanvas.Dispose ( )
```

Definition at line 1165 of file SKMultiLayerRenderCanvas.cs.

7.90.3.3 InsertLayer()

```
void VectSharp.Canvas.SKMultiLayerRenderCanvas.InsertLayer (
            int index,
            SKRenderContext newContent,
            SKRenderAction newTransform )
```

Insert a new layer at the specified index.

Parameters

index	The position at which the new layer will be inserted.
newContent	The contents of the new layer.
newTransform	The transform for the new layer.

Definition at line 309 of file SKMultiLayerRenderCanvas.cs.

7.90.3.4 InvalidateDirty()

```
void VectSharp.Canvas.SKMultiLayerRenderCanvas.InvalidateDirty ( )
```

Invalidate the contents of the canvas, forcing it to redraw itself.

Definition at line 988 of file SKMultiLayerRenderCanvas.cs.

7.90.3.5 InvalidateZIndex()

```
void VectSharp.Canvas.SKMultiLayerRenderCanvas.InvalidateZIndex ( )
```

Invalidate the contents of the canvas, specifying that the order of the layers has changed.

Definition at line 997 of file SKMultiLayerRenderCanvas.cs.

7.90.3.6 MoveLayer()

```
void VectSharp.Canvas.SKMultiLayerRenderCanvas.MoveLayer (
            int oldIndex,
            int newIndex )
```

Move the specified layer to the specified position, shifting all other layers as necessary.

Parameters

oldIndex	The current index of the layer to move.
newIndex	The final index of the layer. Layers after this will be shifted by 1 in order to accommodate the moved layer.

Definition at line 389 of file SKMultiLayerRenderCanvas.cs.

7.90.3.7 RemoveLayer()

```
void VectSharp.Canvas.SKMultiLayerRenderCanvas.RemoveLayer (
            int layer )
```

Remove the specified layer from the image.

Parameters

layer	The index of the layer to remove.

Definition at line 340 of file SKMultiLayerRenderCanvas.cs.

7.90.3.8 Render()

```
override void VectSharp.Canvas.SKMultiLayerRenderCanvas.Render (
            DrawingContext context )
```

Definition at line 1024 of file SKMultiLayerRenderCanvas.cs.

7.90.3.9 RenderAtResolution()

```
RenderTargetBitmap VectSharp.Canvas.SKMultiLayerRenderCanvas.RenderAtResolution (
            int width,
            int height,
            SKColor?  background = null )
```

Render the image at to a bitmap at the specified resolution.

Parameters

width	The width of the rendered image. Note that the actual width of the returned image might be lower than this, depending on the aspect ratio of the image.
height	The height of the rendered image. Note that the actual height of the returned image might be lower than this, depending on the aspect ratio of the image.
background	The background colour for the image. If this is `null`, the current background colour is used.

Returns

A RenderTargetBitmap containing the image rendered at the specified resolution.

Definition at line 735 of file SKMultiLayerRenderCanvas.cs.

7.90.3.10 SwitchLayers()

```
void VectSharp.Canvas.SKMultiLayerRenderCanvas.SwitchLayers (
            int layer1,
            int layer2 )
```

Switch the position of the two specified layers.

Parameters

layer1	The index of the first layer to switch.
layer2	The index of the second layer to switch.

Definition at line 364 of file SKMultiLayerRenderCanvas.cs.

7.90.3.11 UpdateLayer()

```
void VectSharp.Canvas.SKMultiLayerRenderCanvas.UpdateLayer (
            int layer,
            SKRenderContext newContent,
            SKRenderAction newTransform )
```

Replace a single layer with the specified content.

Parameters

layer	The index of the layer to replace.
newContent	The new contents of the layer. A Page can be converded to a SKRenderContext through the CopyToSKRenderContext method.
newTransform	The new transform for the layer.

Definition at line 244 of file SKMultiLayerRenderCanvas.cs.

7.90.3.12 UpdateWith()

```
void VectSharp.Canvas.SKMultiLayerRenderCanvas.UpdateWith (
            List< SKRenderContext > contents,
```

```
List< SKRenderAction > contentTransforms,
Colour backgroundColour,
double width,
double height )
```

Replace the contents of the SKMultiLayerRenderCanvas with the specified layers.

Parameters

contents	The contents of the canvas. Each element in this list represents a layer. A Page can be converded to a SKRenderContext through the CopyToSKRenderContext method.
contentTransforms	A list of transforms associated with each layer. This list should contain the same number of elements as *contents* . This is useful to manipulate the position of each layer individually.
backgroundColour	The background colour of the canvas.
width	The width of the canvas and the page it contains.
height	The height of the canvas and the page it contains.

Definition at line 166 of file SKMultiLayerRenderCanvas.cs.

7.90.4 Member Data Documentation

7.90.4.1 LayerTransforms

```
List<SKRenderAction> VectSharp.Canvas.SKMultiLayerRenderCanvas.LayerTransforms
```

The list of transforms associated with each layer.

Definition at line 64 of file SKMultiLayerRenderCanvas.cs.

7.90.4.2 RenderActions

```
List<List<SKRenderAction> > VectSharp.Canvas.SKMultiLayerRenderCanvas.RenderActions
```

The list of render actions, each element in this list is itself a list, containing the actions that correspond to a layer in the image.

Definition at line 59 of file SKMultiLayerRenderCanvas.cs.

7.90.4.3 RenderLock

```
object VectSharp.Canvas.SKMultiLayerRenderCanvas.RenderLock = new object()
```

An lock for the rendering loop. The public methods of this class already lock on this, but you may need it if you want to directly manipulate the contents of the canvas.

Definition at line 726 of file SKMultiLayerRenderCanvas.cs.

7.90.5 Property Documentation

7.90.5.1 PageHeight

```
double VectSharp.Canvas.SKMultiLayerRenderCanvas.PageHeight  [get], [set]
```

The height of the page that is rendered on this canvas.

Definition at line 53 of file SKMultiLayerRenderCanvas.cs.

7.90.5.2 PageWidth

```
double VectSharp.Canvas.SKMultiLayerRenderCanvas.PageWidth  [get], [set]
```

The width of the page that is rendered on this canvas.

Definition at line 48 of file SKMultiLayerRenderCanvas.cs.

7.90.5.3 Spinner

```
Func<long, Bitmap> VectSharp.Canvas.SKMultiLayerRenderCanvas.Spinner = null  [get], [set]
```

If the image to draw is not already cached, this method is called with an argument containing the number of milliseconds since the image was last rendered. The method can return a Bitmap that will be drawn on the canvas in order to let users know that the image is being rendered in background.

Definition at line 70 of file SKMultiLayerRenderCanvas.cs.

The documentation for this class was generated from the following file:

- VectSharp.Canvas/SKMultiLayerRenderCanvas.cs

7.91 VectSharp.Canvas.SKRenderAction Class Reference

Represents a light-weight rendering action.

Inheritance diagram for VectSharp.Canvas.SKRenderAction:

Public Types

- enum ActionTypes

  Types of rendering actions.

Public Member Functions

- void InvalidateHitTestPath ()
- void InvalidateVisual ()
- void InvalidateZIndex ()
- void InvalidateAll ()
- void Dispose ()

Static Public Member Functions

- static SKRenderAction PathAction (SKPath path, SKPaint paint, string tag=null)

  Creates a new SKRenderAction representing a path.
- static SKRenderAction ClipAction (SKPath clippingPath, string tag=null)

  Creates a new SKRenderAction representing a clipping action.
- static SKRenderAction TextAction (string text, float x, float y, SKFont font, SKPaint paint, string tag=null)

  Creates a new SKRenderAction representing text.
- static SKRenderAction ImageAction (string imageId, SKRect sourceRect, SKRect destinationRect, string tag=null)

  Creates a new SKRenderAction representing an image.
- static SKRenderAction TransformAction (SKMatrix transform, string tag=null)

  Creates a new SKRenderAction representing a transform.
- static SKRenderAction SaveAction (string tag=null)

  Creates a new SKRenderAction that saves the current graphics state.
- static SKRenderAction RestoreAction (string tag=null)

  Creates a new SKRenderAction that saves the current graphics state.
- static SKRenderAction DrawFilteredGraphicsAction (SKRenderContext graphics, IFilter filter, string tag=null)

  Create a new SKRenderAction that draws some graphics with a filter.

Properties

- bool Disposed `[get]`

  Returns a boolean value indicating whether the current instance has been disposed.
- ActionTypes ActionType `[get]`

  Type of the rendering action.
- SKPath Path `[get, set]`

  Path that needs to be rendered (null if the action type is not ActionTypes.Path). If you change this, you probably want to call this object's InvalidateHitTestPath method.
- string Text `[get, set]`

  Text that needs to be rendered (null if the action type is not ActionTypes.Text). If you change this, you probably want to call this object's InvalidateHitTestPath method.
- SKFont Font `[get, set]`

  The font that will be used to render the text (null if the action type is not ActionTypes.Text). If you change this, you probably want to call this object's InvalidateHitTestPath method.
- float TextX `[get, set]`

The X coordainate at which the text will be drawn (null if the action type is not ActionTypes.Text). If you change this, you probably want to call this object's InvalidateHitTestPath method.

- float TextY [get, set]

  The Y coordainate at which the text will be drawn (null if the action type is not ActionTypes.Text). If you change this, you probably want to call this object's InvalidateHitTestPath method.

- SKPaint Paint [get, set]

  Paint used to render the text or path (null if the action type is neither ActionTypes.Text nor ActionTypes.Path). If you change this, you probably want to call this object's InvalidateHitTestPath method.

- string ImageId [get, set]

  Univocal identifier of the image that needs to be drawn.

- SKRect? ImageSource [get, set]

  The source rectangle of the image (null if the action type is not ActionTypes.RasterImage). If you change this, you probably want to call this object's InvalidateVisual method.

- SKRect? ImageDestination [get, set]

  The destination rectangle of the image (null if the action type is not ActionTypes.RasterImage). If you change this, you probably want to call this object's InvalidateHitTestPath method.

- SKMatrix? Transform = null [get, set]

  The transformation matrix that will be applied to the current coordinate system (null if the action type is not Action↩ Types.Transform). If you change this, you probably want to call this object's InvalidateVisual method.

- string Tag [get]

  A tag to access the SKRenderAction.

- uint ZIndex = 0 [get, set]

  The Z-index of the rendering action (an action with a higher Z-index will always appear above an action with a lower Z-index). The more different values there are for the Z-index, the slower the rendering, so keep use of this property to a minimum. If you change this, you probably want to call this object's InvalidateZIndex method.

- SKRenderContext Graphics = null [get, set]

  The graphics that will be drawn with the specified Filter. If you change this, you probably want to call this object's InvalidateVisual method.

- IFilter Filter [get, set]

  The filter with which the Graphics is drawn. If you change this, you probably want to call this object's InvalidateVisual method.

- object Payload [get, set]

  An arbitrary object associated with the RenderAction.

- Avalonia.Controls.Canvas Parent [get]

  The container of this SKRenderAction.

Events

- EventHandler< Avalonia.Input.PointerEventArgs > PointerEnter

  Raised when the pointer enters the area covered by the SKRenderAction.

- EventHandler< Avalonia.Input.PointerEventArgs > PointerLeave

  Raised when the pointer leaves the area covered by the SKRenderAction.

- EventHandler< Avalonia.Input.PointerPressedEventArgs > PointerPressed

  Raised when the pointer is pressed while over the area covered by the SKRenderAction.

- EventHandler< Avalonia.Input.PointerReleasedEventArgs > PointerReleased

  Raised when the pointer is released after a PointerPressed event.

7.91.1 Detailed Description

Represents a light-weight rendering action.

Definition at line 30 of file SKRenderContext.cs.

7.91.2 Member Enumeration Documentation

7.91.2.1 ActionTypes

```
enum VectSharp.Canvas.SKRenderAction.ActionTypes
```

Types of rendering actions.

Definition at line 42 of file SKRenderContext.cs.

7.91.3 Member Function Documentation

7.91.3.1 ClipAction()

```
static SKRenderAction VectSharp.Canvas.SKRenderAction.ClipAction (
            SKPath clippingPath,
            string tag = null )  [static]
```

Creates a new SKRenderAction representing a clipping action.

Parameters

clippingPath	The path to be used for clipping.
tag	A tag to access the SKRenderAction.

Returns

A new SKRenderAction representing a clipping action.

Definition at line 345 of file SKRenderContext.cs.

7.91.3.2 Dispose()

```
void VectSharp.Canvas.SKRenderAction.Dispose ( )
```

Definition at line 494 of file SKRenderContext.cs.

7.91.3.3 DrawFilteredGraphicsAction()

```
static SKRenderAction VectSharp.Canvas.SKRenderAction.DrawFilteredGraphicsAction (
            SKRenderContext graphics,
            IFilter filter,
            string tag = null )  [static]
```

Create a new SKRenderAction that draws some graphics with a filter.

Returns

A new SKRenderAction that draws some graphics with a filter.

Definition at line 464 of file SKRenderContext.cs.

7.91.3.4 ImageAction()

```
static SKRenderAction VectSharp.Canvas.SKRenderAction.ImageAction (
            string imageId,
            SKRect sourceRect,
            SKRect destinationRect,
            string tag = null )  [static]
```

Creates a new SKRenderAction representing an image.

Parameters

imageId	The univocal identifier of the image to draw.
sourceRect	The source rectangle of the image.
destinationRect	The destination rectangle of the image.
tag	A tag to access the SKRenderAction. If this is null this SKRenderAction is not visible in the hit test.

Returns

A new SKRenderAction representing an image.

Definition at line 397 of file SKRenderContext.cs.

7.91.3.5 InvalidateAll()

```
void VectSharp.Canvas.SKRenderAction.InvalidateAll ( )
```

This methods signals to the Parent that the Z-index, shape and visual properties (e.g. the colour) of this object have changed and triggers a redraw.

If you make changes to more than one SKRenderAction contained in the same Canvas, you only need to invalidate the last one.

This method should only be called after the output has been fully initialized.

Definition at line 303 of file SKRenderContext.cs.

7.91.3.6 InvalidateHitTestPath()

```
void VectSharp.Canvas.SKRenderAction.InvalidateHitTestPath ( )
```

Signals to this object that its shape has changed a new path needs to be computed for the purpose of hit-testing. Also signals to the Parent that the visual properties of this object have changed and triggers a redraw.

This method should be called whenever the "shape" of the object represented by the SKRenderAction changes. If only the visual properties of this object have changed (e.g. the colour), call the InvalidateVisual method instead.

If you make changes to more than one SKRenderAction contained in the same Canvas, you only need to invalidate the last one.

This method should only be called after the output has been fully initialized.

Definition at line 232 of file SKRenderContext.cs.

7.91.3.7 InvalidateVisual()

```
void VectSharp.Canvas.SKRenderAction.InvalidateVisual ( )
```

This methods signals to the Parent that the visual properties (e.g. the colour) of this object have changed and triggers a redraw.

If the "shape" of the object has changed as well, call the InvalidateHitTestPath method instead. If the Z-index of the object has changed, call the InvalidateZIndex method instead. If both the "shape" and the Z-index of the object have changed, call the InvalidateAll method.

If you make changes to more than one SKRenderAction contained in the same Canvas, you only need to invalidate the last one.

This method should only be called after the output has been fully initialized.

Definition at line 282 of file SKRenderContext.cs.

7.91.3.8 InvalidateZIndex()

```
void VectSharp.Canvas.SKRenderAction.InvalidateZIndex ( )
```

This methods signals to the Parent that the Z-index and visual properties (e.g. the colour) of this object have changed and triggers a redraw.

If the "shape" of the object has changed as well, call the InvalidateAll method instead.

If you make changes to more than one SKRenderAction contained in the same Canvas, you only need to invalidate the last one.

This method should only be called after the output has been fully initialized.

Definition at line 293 of file SKRenderContext.cs.

7.91.3.9 PathAction()

```
static SKRenderAction VectSharp.Canvas.SKRenderAction.PathAction (
            SKPath path,
            SKPaint paint,
            string tag = null )  [static]
```

Creates a new SKRenderAction representing a path.

Parameters

path	The geometry to be rendered.
paint	The paint used to fill or stroke the path.
tag	A tag to access the SKRenderAction. If this is null this SKRenderAction is not visible in the hit test.

Returns

A new SKRenderAction representing a path.

Definition at line 321 of file SKRenderContext.cs.

7.91.3.10 RestoreAction()

```
static SKRenderAction VectSharp.Canvas.SKRenderAction.RestoreAction (
            string tag = null )  [static]
```

Creates a new SKRenderAction that saves the current graphics state.

Parameters

tag	A tag to access the SKRenderAction.

Returns

A new SKRenderAction that restores the last saved graphics state.

Definition at line 451 of file SKRenderContext.cs.

7.91.3.11 SaveAction()

```
static SKRenderAction VectSharp.Canvas.SKRenderAction.SaveAction (
            string tag = null )  [static]
```

Creates a new SKRenderAction that saves the current graphics state.

Parameters

tag	A tag to access the SKRenderAction.

Returns

A new SKRenderAction that saves the current graphics state.

Definition at line 437 of file SKRenderContext.cs.

7.91.3.12 TextAction()

```
static SKRenderAction VectSharp.Canvas.SKRenderAction.TextAction (
            string text,
            float x,
            float y,
            SKFont font,
            SKPaint paint,
            string tag = null )  [static]
```

Creates a new SKRenderAction representing text.

Parameters

text	The text to be rendered.
x	The X coordinate at which the text will be drawn.
y	The Y coordinate at which the text will be drawn.
font	The font to be used to render the text.
paint	The paint to be used to fill or stroke the text.
tag	A tag to access the SKRenderAction. If this is null this SKRenderAction is not visible in the hit test.

Returns

A new SKRenderAction representing text.

Definition at line 365 of file SKRenderContext.cs.

7.91.3.13 TransformAction()

```
static SKRenderAction VectSharp.Canvas.SKRenderAction.TransformAction (
            SKMatrix transform,
            string tag = null )  [static]
```

Creates a new SKRenderAction representing a transform.

Parameters

transform	The transform to apply.
tag	A tag to access the SKRenderAction.

Returns

A new SKRenderAction representing a transform.

Definition at line 422 of file SKRenderContext.cs.

7.91.4 Property Documentation

7.91.4.1 ActionType

```
ActionTypes VectSharp.Canvas.SKRenderAction.ActionType  [get]
```

Type of the rendering action.

Definition at line 88 of file SKRenderContext.cs.

7.91.4.2 Disposed

```
bool VectSharp.Canvas.SKRenderAction.Disposed  [get]
```

Returns a boolean value indicating whether the current instance has been disposed.

Definition at line 35 of file SKRenderContext.cs.

7.91.4.3 Filter

```
IFilter VectSharp.Canvas.SKRenderAction.Filter  [get], [set]
```

The filter with which the Graphics is drawn. If you change this, you probably want to call this object's InvalidateVisual method.

Definition at line 163 of file SKRenderContext.cs.

7.91.4.4 Font

```
SKFont VectSharp.Canvas.SKRenderAction.Font  [get], [set]
```

The font that will be used to render the text (null if the action type is not ActionTypes.Text). If you change this, you probably want to call this object's InvalidateHitTestPath method.

Definition at line 106 of file SKRenderContext.cs.

7.91.4.5 Graphics

```
SKRenderContext VectSharp.Canvas.SKRenderAction.Graphics = null  [get], [set]
```

The graphics that will be drawn with the specified Filter. If you change this, you probably want to call this object's InvalidateVisual method.

Definition at line 158 of file SKRenderContext.cs.

7.91.4.6 ImageDestination

```
SKRect?  VectSharp.Canvas.SKRenderAction.ImageDestination  [get], [set]
```

The destination rectangle of the image (`null` if the action type is not ActionTypes.RasterImage). If you change this, you probably want to call this object's InvalidateHitTestPath method.

Definition at line 136 of file SKRenderContext.cs.

7.91.4.7 ImageId

```
string VectSharp.Canvas.SKRenderAction.ImageId  [get], [set]
```

Univocal identifier of the image that needs to be drawn.

Definition at line 126 of file SKRenderContext.cs.

7.91.4.8 ImageSource

```
SKRect?  VectSharp.Canvas.SKRenderAction.ImageSource  [get], [set]
```

The source rectangle of the image (`null` if the action type is not ActionTypes.RasterImage). If you change this, you probably want to call this object's InvalidateVisual method.

Definition at line 131 of file SKRenderContext.cs.

7.91.4.9 Paint

```
SKPaint VectSharp.Canvas.SKRenderAction.Paint  [get], [set]
```

Paint used to render the text or path (`null` if the action type is neither ActionTypes.Text nor ActionTypes.Path). If you change this, you probably want to call this object's InvalidateHitTestPath method.

Definition at line 121 of file SKRenderContext.cs.

7.91.4.10 Parent

```
Avalonia.Controls.Canvas VectSharp.Canvas.SKRenderAction.Parent  [get]
```

The container of this SKRenderAction.

Definition at line 175 of file SKRenderContext.cs.

7.91.4.11 Path

```
SKPath VectSharp.Canvas.SKRenderAction.Path  [get], [set]
```

Path that needs to be rendered (null if the action type is not ActionTypes.Path). If you change this, you probably want to call this object's InvalidateHitTestPath method.

Definition at line 93 of file SKRenderContext.cs.

7.91.4.12 Payload

```
object VectSharp.Canvas.SKRenderAction.Payload  [get], [set]
```

An arbitrary object associated with the RenderAction.

Definition at line 168 of file SKRenderContext.cs.

7.91.4.13 Tag

```
string VectSharp.Canvas.SKRenderAction.Tag  [get]
```

A tag to access the SKRenderAction.

Definition at line 146 of file SKRenderContext.cs.

7.91.4.14 Text

```
string VectSharp.Canvas.SKRenderAction.Text  [get], [set]
```

Text that needs to be rendered (null if the action type is not ActionTypes.Text). If you change this, you probably want to call this object's InvalidateHitTestPath method.

Definition at line 101 of file SKRenderContext.cs.

7.91.4.15 TextX

```
float VectSharp.Canvas.SKRenderAction.TextX  [get], [set]
```

The X coordainate at which the text will be drawn (null if the action type is not ActionTypes.Text). If you change this, you probably want to call this object's InvalidateHitTestPath method.

Definition at line 111 of file SKRenderContext.cs.

7.91.4.16 TextY

```
float VectSharp.Canvas.SKRenderAction.TextY  [get], [set]
```

The Y coordainate at which the text will be drawn (null if the action type is not ActionTypes.Text). If you change this, you probably want to call this object's InvalidateHitTestPath method.

Definition at line 116 of file SKRenderContext.cs.

7.91.4.17 Transform

```
SKMatrix?  VectSharp.Canvas.SKRenderAction.Transform = null  [get], [set]
```

The transformation matrix that will be applied to the current coordinate system (`null` if the action type is not ActionTypes.Transform). If you change this, you probably want to call this object's InvalidateVisual method.

Definition at line 141 of file SKRenderContext.cs.

7.91.4.18 ZIndex

```
uint VectSharp.Canvas.SKRenderAction.ZIndex = 0  [get], [set]
```

The Z-index of the rendering action (an action with a higher Z-index will always appear above an action with a lower Z-index). The more different values there are for the Z-index, the slower the rendering, so keep use of this property to a minimum. If you change this, you probably want to call this object's InvalidateZIndex method.

Definition at line 153 of file SKRenderContext.cs.

7.91.5 Event Documentation

7.91.5.1 PointerEnter

```
EventHandler<Avalonia.Input.PointerEventArgs> VectSharp.Canvas.SKRenderAction.PointerEnter
```

Raised when the pointer enters the area covered by the SKRenderAction.

Definition at line 186 of file SKRenderContext.cs.

7.91.5.2 PointerLeave

```
EventHandler<Avalonia.Input.PointerEventArgs> VectSharp.Canvas.SKRenderAction.PointerLeave
```

Raised when the pointer leaves the area covered by the SKRenderAction.

Definition at line 191 of file SKRenderContext.cs.

7.91.5.3 PointerPressed

```
EventHandler<Avalonia.Input.PointerPressedEventArgs> VectSharp.Canvas.SKRenderAction.Pointer↩
Pressed
```

Raised when the pointer is pressed while over the area covered by the SKRenderAction.

Definition at line 196 of file SKRenderContext.cs.

7.91.5.4 PointerReleased

```
EventHandler<Avalonia.Input.PointerReleasedEventArgs> VectSharp.Canvas.SKRenderAction.Pointer↩
Released
```

Raised when the pointer is released after a PointerPressed event.

Definition at line 201 of file SKRenderContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.Canvas/SKRenderContext.cs

7.92 VectSharp.Canvas.SKRenderContext Class Reference

Represents a page that has been prepared for fast rendering using the SkiaSharp renderer.

7.92.1 Detailed Description

Represents a page that has been prepared for fast rendering using the SkiaSharp renderer.

Definition at line 546 of file SKRenderContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.Canvas/SKRenderContext.cs

7.93 VectSharp.Canvas.SKRenderContextInterpreter Class Reference

Contains methods to render a Page to an Avalonia.Controls.Canvas using the SkiaSharp renderer.

Static Public Member Functions

- static SKMultiLayerRenderCanvas PaintToSKCanvas (this Document document, double? width=null, double? height=null, Colour? background=null, AvaloniaContextInterpreter.TextOptions textOption=Avalonia↩ ContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption=default)

  Render a Document to an Avalonia.Controls.Canvas using the SkiaSharp renderer. Each page corresponds to a layer in the image.
- static SKMultiLayerRenderCanvas PaintToSKCanvas (this Document document, Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > > taggedActions, bool removeTagged↩ ActionsAfterExecution=true, double? width=null, double? height=null, Colour? background=null, AvaloniaContextInterpreter.TextOptions textOption=AvaloniaContextInterpreter.TextOptions.ConvertIf↩ Necessary, FilterOption filterOption=default)

  Render a Document to an Avalonia.Controls.Canvas using the SkiaSharp renderer. Each page corresponds to a layer in the image.

- static SKMultiLayerRenderCanvas PaintToSKCanvas (this Document document, Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > > taggedActions, Dictionary< string,(SKBitmap, bool)> images, bool removeTaggedActionsAfterExecution=true, double? width=null, double? height=null, Colour? background=null, AvaloniaContextInterpreter.TextOptions textOption=AvaloniaContextInterpreter.↩ TextOptions.ConvertIfNecessary, FilterOption filterOption=default)

  Render a Document to an Avalonia.Controls.Canvas using the SkiaSharp renderer. Each page corresponds to a layer in the image.
- static SKMultiLayerRenderCanvas PaintToSKCanvas (this Page page, AvaloniaContextInterpreter.TextOptions textOption=AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption=default)

  Render a Page to an Avalonia.Controls.Canvas using the SkiaSharp renderer.
- static SKMultiLayerRenderCanvas PaintToSKCanvas (this Page page, Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > > taggedActions, bool removeTaggedActions↩ AfterExecution=true, AvaloniaContextInterpreter.TextOptions textOption=AvaloniaContextInterpreter.Text↩ Options.ConvertIfNecessary, FilterOption filterOption=default)

  Render a Page to an Avalonia.Controls.Canvas using the SkiaSharpRenderer.
- static SKMultiLayerRenderCanvas PaintToSKCanvas (this Page page, Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > > taggedActions, Dictionary< string,(SKBitmap, bool)> images, bool removeTaggedActionsAfterExecution=true, AvaloniaContextInterpreter.TextOptions textOption=AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption=default)

  Render a Page to an Avalonia.Controls.Canvas using the SkiaSharpRenderer.
- static SKRenderContext CopyToSKRenderContext (this Page page, AvaloniaContextInterpreter.TextOptions textOption=AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption=default)

  Render a Page to a SKRenderContext. This can be drawn using the SkiaSharpRenderer by adding it to a SKMultiLayerRenderCanvas.
- static SKRenderContext CopyToSKRenderContext (this Page page, Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > > taggedActions, bool removeTaggedActions↩ AfterExecution=true, AvaloniaContextInterpreter.TextOptions textOption=AvaloniaContextInterpreter.Text↩ Options.ConvertIfNecessary, FilterOption filterOption=default)

  Render a Page to a SKRenderContext. This can be drawn using the SkiaSharpRenderer by adding it to a SKMultiLayerRenderCanvas.
- static SKRenderContext CopyToSKRenderContext (this Page page, Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > > taggedActions, Dictionary< string,(SKBitmap, bool)> images, bool removeTaggedActionsAfterExecution=true, AvaloniaContextInterpreter.TextOptions textOption=AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption=default)

  Render a Page to a SKRenderContext. This can be drawn using the SkiaSharpRenderer by adding it to a SKMultiLayerRenderCanvas.
- static RasterImage Rasterise (this Graphics graphics, Rectangle region, double scale, bool interpolate)

  Rasterise a region of a Graphics object.

7.93.1 Detailed Description

Contains methods to render a Page to an Avalonia.Controls.Canvas using the SkiaSharp renderer.

Definition at line 1534 of file SKRenderContext.cs.

7.93.2 Member Function Documentation

7.93.2.1 CopyToSKRenderContext() [1/3]

```
static SKRenderContext VectSharp.Canvas.SKRenderContextInterpreter.CopyToSKRenderContext (
            this Page page,
            AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.Conver
            FilterOption filterOption = default )  [static]
```

Render a Page to a SKRenderContext. This can be drawn using the SkiaSharpRenderer by adding it to a SKMultiLayerRenderCanvas.

Parameters

page	The Page to render.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

A SKRenderContext containing the rendered graphics objects.

Definition at line 1655 of file SKRenderContext.cs.

7.93.2.2 CopyToSKRenderContext() [2/3]

```
static SKRenderContext VectSharp.Canvas.SKRenderContextInterpreter.CopyToSKRenderContext (
            this Page page,
            Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > >
taggedActions,
            bool removeTaggedActionsAfterExecution = true,
            AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.Conver
            FilterOption filterOption = default )  [static]
```

Render a Page to a SKRenderContext. This can be drawn using the SkiaSharpRenderer by adding it to a SKMultiLayerRenderCanvas.

Parameters

page	The Page to render.
taggedActions	A Dictionary containing the actions that will be performed on items with the corresponding tag. These should be functions that accept one parameter of type SKRenderAction and return an IEnumerable<SKRenderAction> of the render actions that will actually be added to the plot.
removeTaggedActionsAfterExecution	Whether the actions should be removed from *taggedActions* after their execution. Set to false if the same action should be performed on multiple items with the same tag.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

A SKRenderContext containing the rendered graphics objects.

Definition at line 1670 of file SKRenderContext.cs.

7.93.2.3 CopyToSKRenderContext() [3/3]

```
static SKRenderContext VectSharp.Canvas.SKRenderContextInterpreter.CopyToSKRenderContext (
            this Page page,
            Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > >
taggedActions,
            Dictionary< string,(SKBitmap, bool)> images,
            bool removeTaggedActionsAfterExecution = true,
            AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.Conver
            FilterOption filterOption = default )  [static]
```

Render a Page to a SKRenderContext. This can be drawn using the SkiaSharpRenderer by adding it to a SKMultiLayerRenderCanvas.

Parameters

page	The Page to render.
taggedActions	A Dictionary containing the actions that will be performed on items with the corresponding tag. These should be functions that accept one parameter of type SKRenderAction and return an IEnumerable<SKRenderAction> of the render actions that will actually be added to the plot.
images	A dictionary that associates to each raster image path (or data URL) the image rendered as a SKBitmap and a boolean value indicating whether it should be drawn as "pixelated" or not. This will be populated automatically as the page is rendered. If you are rendering multiple Pages (or you are rendering the same page multiple times), it will be beneficial to keep a reference to this dictionary and pass it again on further rendering requests; otherwise, you can just pass an empty dictionary.
removeTaggedActionsAfterExecution	Whether the actions should be removed from *taggedActions* after their execution. Set to false if the same action should be performed on multiple items with the same tag.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

A SKRenderContext containing the rendered graphics objects.

Definition at line 1687 of file SKRenderContext.cs.

7.93.2.4 PaintToSKCanvas() [1/6]

```
static SKMultiLayerRenderCanvas VectSharp.Canvas.SKRenderContextInterpreter.PaintToSKCanvas (
            this Document document,
            Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > >
taggedActions,
            bool removeTaggedActionsAfterExecution = true,
            double?  width = null,
            double?  height = null,
            Colour?  background = null,
            AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.Conver
            FilterOption filterOption = default )  [static]
```

Render a Document to an Avalonia.Controls.Canvas using the SkiaSharp renderer. Each page corresponds to a layer in the image.

Parameters

document	The Document to render.
taggedActions	A Dictionary containing the actions that will be performed on items with the corresponding tag. These should be functions that accept one parameter of type SKRenderAction and return an IEnumerable<SKRenderAction> of the render actions that will actually be added to the plot.
removeTaggedActionsAfterExecution	Whether the actions should be removed from *taggedActions* after their execution. Set to false if the same action should be performed on multiple items with the same tag.
width	The width of the document. If this is `null`, the width of the largest page is used.
height	The height of the document. If this is `null`, the height of the largest page is used.
background	The background colour of the document. If this is `null`, a transparent background is used.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered graphics objects.

Definition at line 1579 of file SKRenderContext.cs.

7.93.2.5 PaintToSKCanvas() [2/6]

```
static SKMultiLayerRenderCanvas VectSharp.Canvas.SKRenderContextInterpreter.PaintToSKCanvas (
            this Document document,
            Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > >
taggedActions,
```

```
Dictionary< string,(SKBitmap, bool)> images,
bool removeTaggedActionsAfterExecution = true,
double?  width = null,
double?  height = null,
Colour?  background = null,
AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.Conver
FilterOption filterOption = default )  [static]
```

Render a Document to an Avalonia.Controls.Canvas using the SkiaSharp renderer. Each page corresponds to a layer in the image.

Parameters

document	The Document to render.
taggedActions	A Dictionary containing the actions that will be performed on items with the corresponding tag. These should be functions that accept one parameter of type SKRenderAction and return an IEnumerable<SKRenderAction> of the render actions that will actually be added to the plot.
images	A dictionary that associates to each raster image path (or data URL) the image rendered as a SKBitmap and a boolean value indicating whether it should be drawn as "pixelated" or not. This will be populated automatically as the page is rendered. If you are rendering multiple Pages (or you are rendering the same page multiple times), it will be beneficial to keep a reference to this dictionary and pass it again on further rendering requests; otherwise, you can just pass an empty dictionary.
removeTaggedActionsAfterExecution	Whether the actions should be removed from *taggedActions* after their execution. Set to false if the same action should be performed on multiple items with the same tag.
width	The width of the document. If this is `null`, the width of the largest page is used.
height	The height of the document. If this is `null`, the height of the largest page is used.
background	The background colour of the document. If this is `null`, a transparent background is used.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered graphics objects.

Definition at line 1599 of file SKRenderContext.cs.

7.93.2.6 PaintToSKCanvas() [3/6]

```
static SKMultiLayerRenderCanvas VectSharp.Canvas.SKRenderContextInterpreter.PaintToSKCanvas (
            this Document document,
```

```
            double?  width = null,
            double?  height = null,
            Colour?  background = null,
            AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.Conver
            FilterOption filterOption = default )  [static]
```

Render a Document to an Avalonia.Controls.Canvas using the SkiaSharp renderer. Each page corresponds to a layer in the image.

Parameters

document	The Document to render.
width	The width of the document. If this is `null`, the width of the largest page is used.
height	The height of the document. If this is `null`, the height of the largest page is used.
background	The background colour of the document. If this is `null`, a transparent background is used.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered graphics objects.

Definition at line 1561 of file SKRenderContext.cs.

7.93.2.7 PaintToSKCanvas() [4/6]

```
static SKMultiLayerRenderCanvas VectSharp.Canvas.SKRenderContextInterpreter.PaintToSKCanvas (
            this Page page,
            AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.Conver
            FilterOption filterOption = default )  [static]
```

Render a Page to an Avalonia.Controls.Canvas using the SkiaSharp renderer.

Parameters

page	The Page to render.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered graphics objects.

Definition at line 1611 of file SKRenderContext.cs.

7.93.2.8 PaintToSKCanvas() [5/6]

```
static SKMultiLayerRenderCanvas VectSharp.Canvas.SKRenderContextInterpreter.PaintToSKCanvas (
            this Page page,
            Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > >
taggedActions,
            bool removeTaggedActionsAfterExecution = true,
            AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.Conver
            FilterOption filterOption = default )  [static]
```

Render a Page to an Avalonia.Controls.Canvas using the SkiaSharpRenderer.

Parameters

page	The Page to render.
taggedActions	A Dictionary containing the actions that will be performed on items with the corresponding tag. These should be functions that accept one parameter of type SKRenderAction and return an IEnumerable<SKRenderAction> of the render actions that will actually be added to the plot.
removeTaggedActionsAfterExecution	Whether the actions should be removed from *taggedActions* after their execution. Set to false if the same action should be performed on multiple items with the same tag.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered graphics objects.

Definition at line 1626 of file SKRenderContext.cs.

7.93.2.9 PaintToSKCanvas() [6/6]

```
static SKMultiLayerRenderCanvas VectSharp.Canvas.SKRenderContextInterpreter.PaintToSKCanvas (
            this Page page,
            Dictionary< string, Func< SKRenderAction, IEnumerable< SKRenderAction > > >
taggedActions,
            Dictionary< string,(SKBitmap, bool)> images,
            bool removeTaggedActionsAfterExecution = true,
            AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.Conver
            FilterOption filterOption = default )  [static]
```

Render a Page to an Avalonia.Controls.Canvas using the SkiaSharpRenderer.

Parameters

page	The Page to render.

Parameters

taggedActions	A Dictionary containing the actions that will be performed on items with the corresponding tag. These should be functions that accept one parameter of type SKRenderAction and return an IEnumerable<SKRenderAction> of the render actions that will actually be added to the plot.
images	A dictionary that associates to each raster image path (or data URL) the image rendered as a SKBitmap and a boolean value indicating whether it should be drawn as "pixelated" or not. This will be populated automatically as the page is rendered. If you are rendering multiple Pages (or you are rendering the same page multiple times), it will be beneficial to keep a reference to this dictionary and pass it again on further rendering requests; otherwise, you can just pass an empty dictionary.
removeTaggedActionsAfterExecution	Whether the actions should be removed from *taggedActions* after their execution. Set to false if the same action should be performed on multiple items with the same tag.
textOption	Defines whether text items should be converted into paths when drawing.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An Avalonia.Controls.Canvas containing the rendered graphics objects.

Definition at line 1643 of file SKRenderContext.cs.

7.93.2.10 Rasterise()

```
static RasterImage VectSharp.Canvas.SKRenderContextInterpreter.Rasterise (
            this Graphics graphics,
            Rectangle region,
            double scale,
            bool interpolate )  [static]
```

Rasterise a region of a Graphics object.

Parameters

graphics	The Graphics object that will be rasterised.
region	The region of the *graphics* that will be rasterised.
scale	The scale at which the image will be rendered.
interpolate	Whether the resulting image should be interpolated or not when it is drawn on another Graphics surface.

Returns

A RasterImage containing the rasterised graphics.

Definition at line 1712 of file SKRenderContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.Canvas/SKRenderContext.cs

7.94 VectSharp.SolidColourBrush Class Reference

Represents a brush painting with a single solid colour.

Inheritance diagram for VectSharp.SolidColourBrush:

Public Member Functions

- SolidColourBrush (Colour colour)

  Creates a new SolidColourBrush with the specified colour .
- override Brush MultiplyOpacity (double opacity)

  Returns a brush corresponding the current instance, with the specified opacity multiplication applied.

Parameters

opacity	*The value that will be used to multiply the opacity of the brush.*

Returns

A brush corresponding the current instance, with the specified opacity multiplication applied.

Static Public Member Functions

- static implicit operator SolidColourBrush (Colour colour)

  Implicitly converts a Colour into a SolidColourBrush.

Properties

- Colour Colour [get]

  The colour of the brush.
- double R [get]

  Red component of the colour. Range: [0, 1].
- double G [get]

  Green component of the colour. Range: [0, 1].
- double B [get]

  Blue component of the colour. Range: [0, 1].
- double A [get]

  Alpha component of the colour. Range: [0, 1].

7.94.1 Detailed Description

Represents a brush painting with a single solid colour.

Definition at line 54 of file Brush.cs.

7.94.2 Constructor & Destructor Documentation

7.94.2.1 SolidColourBrush()

```
VectSharp.SolidColourBrush.SolidColourBrush (
            Colour colour )
```

Creates a new SolidColourBrush with the specified *colour* .

Parameters

colour	The Colour to use for the brush.

Definition at line 85 of file Brush.cs.

7.94.3 Member Function Documentation

7.94.3.1 MultiplyOpacity()

```
override Brush VectSharp.SolidColourBrush.MultiplyOpacity (
            double opacity )  [virtual]
```

Returns a brush corresponding the current instance, with the specified *opacity* multiplication applied.

Parameters

opacity	The value that will be used to multiply the opacity of the brush.

Returns

A brush corresponding the current instance, with the specified *opacity* multiplication applied.

Implements VectSharp.Brush.

Definition at line 91 of file Brush.cs.

7.94.3.2 operator SolidColourBrush()

```
static implicit VectSharp.SolidColourBrush.operator SolidColourBrush (
            Colour colour )  [static]
```

Implicitly converts a Colour into a SolidColourBrush.

Parameters

colour	The Colour to use for the brush.

Definition at line 100 of file Brush.cs.

7.94.4 Property Documentation

7.94.4.1 A

```
double VectSharp.SolidColourBrush.A  [get]
```

Alpha component of the colour. Range: [0, 1].

Definition at line 79 of file Brush.cs.

7.94.4.2 B

```
double VectSharp.SolidColourBrush.B  [get]
```

Blue component of the colour. Range: [0, 1].

Definition at line 74 of file Brush.cs.

7.94.4.3 Colour

```
Colour VectSharp.SolidColourBrush.Colour [get]
```

The colour of the brush.

Definition at line 59 of file Brush.cs.

7.94.4.4 G

```
double VectSharp.SolidColourBrush.G [get]
```

Green component of the colour. Range: [0, 1].

Definition at line 69 of file Brush.cs.

7.94.4.5 R

```
double VectSharp.SolidColourBrush.R [get]
```

Red component of the colour. Range: [0, 1].

Definition at line 64 of file Brush.cs.

The documentation for this class was generated from the following file:

- VectSharp/Brush.cs

7.95 VectSharp.SplineEasing Class Reference

Describes an easing defined by a Cubic Bezier curve.

Inheritance diagram for VectSharp.SplineEasing:

Public Member Functions

- SplineEasing (Point controlPoint1, Point controlPoint2)

  Creates a new SplineEasing with the specified control points. The start point is always (0, 0) and the end point is always (1, 1).
- double Ease (double value)

  Applies the easing to the specified transition offset.

Parameters

value	*The transition offset (ranging from 0 to 1).*

Returns

The eased transition offset value.

Properties

- Point ControlPoint1 `[get]`

  The first control point of the curve.
- Point ControlPoint2 `[get]`

  The second control point of the curve.

7.95.1 Detailed Description

Describes an easing defined by a Cubic Bezier curve.

Definition at line 1274 of file Animation.cs.

7.95.2 Constructor & Destructor Documentation

7.95.2.1 SplineEasing()

```
VectSharp.SplineEasing.SplineEasing (
            Point controlPoint1,
            Point controlPoint2 )
```

Creates a new SplineEasing with the specified control points. The start point is always (0, 0) and the end point is always (1, 1).

Parameters

controlPoint1	The first control point of the curve. Both X and Y must be between 0 and 1, inclusive.
controlPoint2	The second control point of the curve. Both X and Y must be between 0 and 1, inclusive.

Exceptions

ArgumentException	This exception is thrown if any coordinate of the control points is < 0 or > 1.

Definition at line 1293 of file Animation.cs.

7.95.3 Member Function Documentation

7.95.3.1 Ease()

```
double VectSharp.SplineEasing.Ease (
            double value )
```

Applies the easing to the specified transition offset.

Parameters

value	The transition offset (ranging from 0 to 1).

Returns

The eased transition offset value.

Implements VectSharp.IEasing.

Definition at line 1342 of file Animation.cs.

7.95.4 Property Documentation

7.95.4.1 ControlPoint1

```
Point VectSharp.SplineEasing.ControlPoint1  [get]
```

The first control point of the curve.

Definition at line 1279 of file Animation.cs.

7.95.4.2 ControlPoint2

```
Point VectSharp.SplineEasing.ControlPoint2  [get]
```

The second control point of the curve.

Definition at line 1284 of file Animation.cs.

The documentation for this class was generated from the following file:

- VectSharp/Animation.cs

7.96 VectSharp.ThreeD.SpotlightLightSource Class Reference

Represents a conic spotlight.

Inheritance diagram for VectSharp.ThreeD.SpotlightLightSource:

Public Member Functions

- SpotlightLightSource (double intensity, Point3D position, NormalizedVector3D direction, double beamWidth↩ Angle, double cutoffAngle)

  Creates a new SpotlightLightSource instance.
- LightIntensity GetLightAt (Point3D point)

  Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	*The Point3DElement at which the light intensity should be computed.*

Returns

- double GetObstruction (Point3D point, IEnumerable< Triangle3DElement > shadowingTriangles)

  Determines the amount of obstruction of the light that results at point due to the specified shadowingTriangles .

Parameters

point	*The Point3D at which the obstruction should be computed.*
shadowingTriangles	*A collection of Triangle3DElement casting shadows.*

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Properties

- bool CastsShadow = true `[get, set]`

  Determines whether the light casts a shadow or not.
- Point3D Position `[get, set]`

The position of the light source.

- NormalizedVector3D Direction [get, set]

  The direction of the cone axis.

- double Intensity [get, set]

  The base intensity of the light.

- double BeamWidthAngle [get, set]

  The angular size of the light cone, in radians.

- double CutoffAngle [get, set]

  The angular size of the cutoff cone, in radians.

- double DistanceAttenuationExponent = 2 [get, set]

  An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.

- double AngleAttenuationExponent = 1 [get, set]

  An exponent determining how fast the light attenuates between the main light cone and the cutoff cone.

7.96.1 Detailed Description

Represents a conic spotlight.

Definition at line 256 of file Lights.cs.

7.96.2 Constructor & Destructor Documentation

7.96.2.1 SpotlightLightSource()

```
VectSharp.ThreeD.SpotlightLightSource.SpotlightLightSource (
            double intensity,
            Point3D position,
            NormalizedVector3D direction,
            double beamWidthAngle,
            double cutoffAngle )
```

Creates a new SpotlightLightSource instance.

Parameters

intensity	The intensity of the light.
position	The position of the light source.
direction	The direction of the cone's axis.
beamWidthAngle	The angular size of the light cone, in radians.
cutoffAngle	The angular size of the cutoff cone, in radians.

Definition at line 304 of file Lights.cs.

7.96.3 Member Function Documentation

7.96.3.1 GetLightAt()

```
LightIntensity VectSharp.ThreeD.SpotlightLightSource.GetLightAt (
            Point3D point )
```

Computes the light intensity at the specified point, without taking into account any obstructions.

Parameters

point	The Point3DElement at which the light intensity should be computed.

Returns

Implements VectSharp.ThreeD.ILightSource.

Definition at line 314 of file Lights.cs.

7.96.3.2 GetObstruction()

```
double VectSharp.ThreeD.SpotlightLightSource.GetObstruction (
            Point3D point,
            IEnumerable< Triangle3DElement > shadowingTriangles )
```

Determines the amount of obstruction of the light that results at *point* due to the specified *shadowingTriangles* .

Parameters

point	The Point3D at which the obstruction should be computed.
shadowingTriangles	A collection of Triangle3DElement casting shadows.

Returns

1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 362 of file Lights.cs.

7.96.4 Property Documentation

7.96.4.1 AngleAttenuationExponent

```
double VectSharp.ThreeD.SpotlightLightSource.AngleAttenuationExponent = 1  [get], [set]
```

An exponent determining how fast the light attenuates between the main light cone and the cutoff cone.

Definition at line 294 of file Lights.cs.

7.96.4.2 BeamWidthAngle

```
double VectSharp.ThreeD.SpotlightLightSource.BeamWidthAngle  [get], [set]
```

The angular size of the light cone, in radians.

Definition at line 279 of file Lights.cs.

7.96.4.3 CastsShadow

```
bool VectSharp.ThreeD.SpotlightLightSource.CastsShadow = true  [get], [set]
```

Determines whether the light casts a shadow or not.

Implements VectSharp.ThreeD.ILightSource.

Definition at line 259 of file Lights.cs.

7.96.4.4 CutoffAngle

```
double VectSharp.ThreeD.SpotlightLightSource.CutoffAngle  [get], [set]
```

The angular size of the cutoff cone, in radians.

Definition at line 284 of file Lights.cs.

7.96.4.5 Direction

```
NormalizedVector3D VectSharp.ThreeD.SpotlightLightSource.Direction  [get], [set]
```

The direction of the cone axis.

Definition at line 269 of file Lights.cs.

7.96.4.6 DistanceAttenuationExponent

```
double VectSharp.ThreeD.SpotlightLightSource.DistanceAttenuationExponent = 2  [get], [set]
```

An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.

Definition at line 289 of file Lights.cs.

7.96.4.7 Intensity

```
double VectSharp.ThreeD.SpotlightLightSource.Intensity  [get], [set]
```

The base intensity of the light.

Definition at line 274 of file Lights.cs.

7.96.4.8 Position

```
Point3D VectSharp.ThreeD.SpotlightLightSource.Position  [get], [set]
```

The position of the light source.

Definition at line 264 of file Lights.cs.

The documentation for this class was generated from the following file:

- VectSharp.ThreeD/Lights.cs

7.97 VectSharp.SVG.SVGContextInterpreter Class Reference

Contains methods to render a Page as an SVG file.

Classes

- class FilterOption

  Determines how and whether image filters are rasterised.

Public Types

- enum TextOptions

  Defines whether the used fonts should be included in the file.

Static Public Member Functions

- static void SaveAsSVG (this Page page, string fileName, TextOptions textOption=TextOptions.SubsetFonts, Dictionary< string, string > linkDestinations=null, FilterOption filterOption=default, bool useStyles=false)

  Render the page to an SVG file.
- static void SaveAsSVG (this Page page, Stream stream, TextOptions textOption=TextOptions.SubsetFonts, Dictionary< string, string > linkDestinations=null, FilterOption filterOption=default, bool useStyles=false)

  Render the page to an SVG stream.
- static XmlDocument SaveAsSVG (this Page page, TextOptions textOption=TextOptions.SubsetFonts, Dictionary< string, string > linkDestinations=null, FilterOption filterOption=default, bool useStyles=false)

  Render the page to an SVG document.
- static XmlDocument SaveAsAnimatedSVG (this Animation animation, bool includeControls=false, double durationScaling=1, TextOptions textOption=TextOptions.SubsetFonts, Dictionary< string, string > link↩ Destinations=null, FilterOption filterOption=default)

  Render the animation to an SVG document, using SVG animations.
- static void SaveAsAnimatedSVG (this Animation animation, Stream stream, bool includeControls=false, double durationScaling=1, TextOptions textOption=TextOptions.SubsetFonts, Dictionary< string, string > link↩ Destinations=null, FilterOption filterOption=default)

  Render the animation to an SVG stream, using SVG animations.
- static void SaveAsAnimatedSVG (this Animation animation, string fileName, bool includeControls=false, double durationScaling=1, TextOptions textOption=TextOptions.SubsetFonts, Dictionary< string, string > link↩ Destinations=null, FilterOption filterOption=default)

  Render the animation to an SVG file, using SVG animations.
- static XmlDocument SaveAsAnimatedSVGWithFrames (this Animation animation, bool includeControls=false, double frameRate=60, double durationScaling=1, TextOptions textOption=TextOptions.SubsetFonts, Dictionary< string, string > linkDestinations=null, FilterOption filterOption=default)

  Render the animation to an SVG document, encoding discrete frames.
- static void SaveAsAnimatedSVGWithFrames (this Animation animation, Stream stream, bool include↩ Controls=false, double frameRate=60, double durationScaling=1, TextOptions textOption=TextOptions.↩ SubsetFonts, Dictionary< string, string > linkDestinations=null, FilterOption filterOption=default)

  Render the animation to an SVG stream, encoding discrete frames.
- static void SaveAsAnimatedSVGWithFrames (this Animation animation, string fileName, bool include↩ Controls=false, double frameRate=60, double durationScaling=1, TextOptions textOption=TextOptions.↩ SubsetFonts, Dictionary< string, string > linkDestinations=null, FilterOption filterOption=default)

  Render the animation to an SVG file, encoding discrete frames.

7.97.1 Detailed Description

Contains methods to render a Page as an SVG file.

Definition at line 2093 of file SVGContext.cs.

7.97.2 Member Enumeration Documentation

7.97.2.1 TextOptions

```
enum VectSharp.SVG.SVGContextInterpreter.TextOptions
```

Defines whether the used fonts should be included in the file.

Definition at line 2116 of file SVGContext.cs.

7.97.3 Member Function Documentation

7.97.3.1 SaveAsAnimatedSVG() [1/3]

```
static XmlDocument VectSharp.SVG.SVGContextInterpreter.SaveAsAnimatedSVG (
            this Animation animation,
            bool includeControls = false,
            double durationScaling = 1,
            TextOptions textOption = TextOptions.SubsetFonts,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default )  [static]
```

Render the animation to an SVG document, using SVG animations.

Parameters

animation	The Animation to render.
includeControls	If this is `true`, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
textOption	Defines whether the used fonts should be included in the file.
linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An XmlDocument containing the animated SVG image.

Definition at line 2527 of file SVGContext.cs.

7.97.3.2 SaveAsAnimatedSVG() [2/3]

```
static void VectSharp.SVG.SVGContextInterpreter.SaveAsAnimatedSVG (
            this Animation animation,
            Stream stream,
            bool includeControls = false,
            double durationScaling = 1,
            TextOptions textOption = TextOptions.SubsetFonts,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default )  [static]
```

Render the animation to an SVG stream, using SVG animations.

Parameters

animation	The Animation to render.
stream	The Stream on which the SVG document will be written.
includeControls	If this is `true`, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
textOption	Defines whether the used fonts should be included in the file.
linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Definition at line 3023 of file SVGContext.cs.

7.97.3.3 SaveAsAnimatedSVG() [3/3]

```
static void VectSharp.SVG.SVGContextInterpreter.SaveAsAnimatedSVG (
            this Animation animation,
            string fileName,
            bool includeControls = false,
            double durationScaling = 1,
            TextOptions textOption = TextOptions.SubsetFonts,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default )  [static]
```

Render the animation to an SVG file, using SVG animations.

Parameters

animation	The Animation to render.
fileName	The output file that will be created.
includeControls	If this is `true`, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
textOption	Defines whether the used fonts should be included in the file.
linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Definition at line 3039 of file SVGContext.cs.

7.97.3.4 SaveAsAnimatedSVGWithFrames() [1/3]

```
static XmlDocument VectSharp.SVG.SVGContextInterpreter.SaveAsAnimatedSVGWithFrames (
            this Animation animation,
            bool includeControls = false,
            double frameRate = 60,
            double durationScaling = 1,
            TextOptions textOption = TextOptions.SubsetFonts,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default )  [static]
```

Render the animation to an SVG document, encoding discrete frames.

Parameters

animation	The Animation to render.
includeControls	If this is `true`, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.
frameRate	The target frame rate of the animation, in frames-per-second (fps).
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
textOption	Defines whether the used fonts should be included in the file.
linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Returns

An XmlDocument containing the animated SVG image.

Definition at line 3058 of file SVGContext.cs.

7.97.3.5 SaveAsAnimatedSVGWithFrames() [2/3]

```
static void VectSharp.SVG.SVGContextInterpreter.SaveAsAnimatedSVGWithFrames (
            this Animation animation,
            Stream stream,
            bool includeControls = false,
            double frameRate = 60,
            double durationScaling = 1,
            TextOptions textOption = TextOptions.SubsetFonts,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default )  [static]
```

Render the animation to an SVG stream, encoding discrete frames.

Parameters

animation	The Animation to render.
includeControls	If this is `true`, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.
stream	The Stream on which the SVG document will be written.
frameRate	The target frame rate of the animation, in frames-per-second (fps).
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
textOption	Defines whether the used fonts should be included in the file.
linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Definition at line 3136 of file SVGContext.cs.

7.97.3.6 SaveAsAnimatedSVGWithFrames() [3/3]

```
static void VectSharp.SVG.SVGContextInterpreter.SaveAsAnimatedSVGWithFrames (
            this Animation animation,
            string fileName,
            bool includeControls = false,
            double frameRate = 60,
            double durationScaling = 1,
            TextOptions textOption = TextOptions.SubsetFonts,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default )  [static]
```

Render the animation to an SVG file, encoding discrete frames.

Parameters

animation	The Animation to render.
includeControls	If this is `true`, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.
fileName	The output file that will be created.
frameRate	The target frame rate of the animation, in frames-per-second (fps).
durationScaling	A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.
textOption	Defines whether the used fonts should be included in the file.
linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.

Definition at line 3153 of file SVGContext.cs.

7.97.3.7 SaveAsSVG() [1/3]

```
static void VectSharp.SVG.SVGContextInterpreter.SaveAsSVG (
            this Page page,
            Stream stream,
            TextOptions textOption = TextOptions.SubsetFonts,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default,
            bool useStyles = false )  [static]
```

Render the page to an SVG stream.

Parameters

page	The Page to render.
stream	The stream to which the SVG data will be written.
textOption	Defines whether the used fonts should be included in the file.
linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.
useStyles	If this is `false`, presentation attributes are set as attributes on SVG elements. If this is `true`, CSS classes are used to set presentation attributes.

Definition at line 2223 of file SVGContext.cs.

7.97.3.8 SaveAsSVG() [2/3]

```
static void VectSharp.SVG.SVGContextInterpreter.SaveAsSVG (
            this Page page,
            string fileName,
            TextOptions textOption = TextOptions.SubsetFonts,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default,
            bool useStyles = false )  [static]
```

Render the page to an SVG file.

Parameters

page	The Page to render.
fileName	The full path to the file to save. If it exists, it will be overwritten.
textOption	Defines whether the used fonts should be included in the file.

Parameters

linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.
useStyles	If this is `false`, presentation attributes are set as attributes on SVG elements. If this is `true`, CSS classes are used to set presentation attributes.

Definition at line 2105 of file SVGContext.cs.

7.97.3.9 SaveAsSVG() [3/3]

```
static XmlDocument VectSharp.SVG.SVGContextInterpreter.SaveAsSVG (
            this Page page,
            TextOptions textOption = TextOptions.SubsetFonts,
            Dictionary< string, string > linkDestinations = null,
            FilterOption filterOption = default,
            bool useStyles = false )  [static]
```

Render the page to an SVG document.

Parameters

page	The Page to render.
textOption	Defines whether the used fonts should be included in the file.
linkDestinations	A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.
filterOption	Defines how and whether image filters should be rasterised when rendering the image.
useStyles	If this is `false`, presentation attributes are set as attributes on SVG elements. If this is `true`, CSS classes are used to set presentation attributes.

Returns

An XmlDocument containing the rendered SVG image.

Definition at line 2238 of file SVGContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.SVG/SVGContext.cs

7.98 VectSharp.Markdown.SyntaxHighlighter Class Reference

Contains methods to perform syntax highlighting.

Static Public Member Functions

- static List< List< FormattedString > > GetSyntaxHighlightedLines (string sourceCode, string language)

  Performs syntax highlighting for a specified language on some source code.

7.98.1 Detailed Description

Contains methods to perform syntax highlighting.

Definition at line 73 of file SyntaxHighlighting.cs.

7.98.2 Member Function Documentation

7.98.2.1 GetSyntaxHighlightedLines()

```
static List< List< FormattedString > > VectSharp.Markdown.SyntaxHighlighter.GetSyntaxHighlighted↩
Lines (
            string sourceCode,
            string language )  [static]
```

Performs syntax highlighting for a specified language on some source code.

Parameters

sourceCode	The source code to be highlighted.
language	The name of the language to use for the highlighting.

Returns

A list of lists of FormattedStrings. Each list of FormattedStrings represents a line.

Definition at line 129 of file SyntaxHighlighting.cs.

The documentation for this class was generated from the following file:

- VectSharp.Markdown/SyntaxHighlighting.cs

7.99 VectSharp.Transition Class Reference

Describes the transition between two successive Frames.

Public Member Functions

- Transition (double duration, IEasing easing=null, Dictionary< string, IEasing > easings=null)

  Creates a new Transition with the specified duration and easings.

Properties

- double Duration [get]

  The duration of the transition, in milliseconds.
- IEasing OverallEasing = null [get]

  The IEasing to apply to all elements for which another easing is not specified. Set to null to use the default linear easing.
- Dictionary< string, IEasing > Easings = null [get]

  A dictionary associating graphic action tags to the corresponding IEasing.

7.99.1 Detailed Description

Describes the transition between two successive Frames.

Definition at line 1382 of file Animation.cs.

7.99.2 Constructor & Destructor Documentation

7.99.2.1 Transition()

```
VectSharp.Transition.Transition (
            double duration,
            IEasing easing = null,
            Dictionary< string, IEasing > easings = null )
```

Creates a new Transition with the specified duration and easings.

Parameters

duration	The duration of the transition, in milliseconds.
easing	The IEasing to apply to all elements for which another easing is not specified. Set to null to use the default linear easing.
easings	A dictionary associating graphic action tags to the corresponding IEasing.

Definition at line 1405 of file Animation.cs.

7.99.3 Property Documentation

7.99.3.1 Duration

```
double VectSharp.Transition.Duration [get]
```

The duration of the transition, in milliseconds.

Definition at line 1387 of file Animation.cs.

7.99.3.2 Easings

```
Dictionary<string, IEasing> VectSharp.Transition.Easings = null  [get]
```

A dictionary associating graphic action tags to the corresponding IEasing.

Definition at line 1397 of file Animation.cs.

7.99.3.3 OverallEasing

```
IEasing VectSharp.Transition.OverallEasing = null  [get]
```

The IEasing to apply to all elements for which another easing is not specified. Set to null to use the default linear easing.

Definition at line 1392 of file Animation.cs.

The documentation for this class was generated from the following file:

- VectSharp/Animation.cs

7.100 VectSharp.TrueTypeFile Class Reference

Represents a font file in TrueType format. Reference: http://stevehanov.ca/blog/?id=143, https://developer.apple.com/fonts/TrueType-Reference-Manual/, https://docs.microsoft.com/en-us/typography/opentype/spec/

Classes

- struct Bearings

  Represents the left- and right-side bearings of a glyph.
- class PairKerning

  Contains information describing how the position of two glyphs in a kerning pair should be altered.
- struct TrueTypePoint

  Represents a point in a TrueType path description.
- struct VerticalMetrics

  Represents the maximum heigth above and depth below the baseline of a glyph.

Public Member Functions

- void Destroy ()

  Remove this TrueType file from the cache, clear the tables and release the FontStream. Only call this when the actual file that was used to create this object needs to be changed!
- TrueTypeFile SubsetFont (string charactersToInclude, bool consolidateAt32=false, Dictionary< char, char > outputEncoding=null)

  Create a subset of the TrueType file, containing only the glyphs for the specified characters.
- string GetFontFamilyName ()

  Obtains the font family name from the TrueType file.
- string GetFullFontFamilyName ()

  Obtains the full font family name from the TrueType file.
- string GetFontName ()

  Obtains the PostScript font name from the TrueType file.
- ushort GetFirstCharIndex ()

  Returns the index of the first character glyph represented by the font.
- ushort GetLastCharIndex ()

  Returns the index of the last character glyph represented by the font.
- bool IsItalic ()

  Determines whether the typeface is Italic or Oblique or not.
- bool IsOblique ()

  Determines whether the typeface is Oblique or not.
- bool IsBold ()

  Determines whether the typeface is Bold or not.
- bool IsFixedPitch ()

  Determines whether the typeface is fixed-pitch (aka monospaces) or not.
- bool IsSerif ()

  Determines whether the typeface is serifed or not.
- bool IsScript ()

  Determines whether the typeface is a script typeface or not.
- int GetGlyphIndex (char glyph)

  Determines the index of the glyph corresponding to a certain character.
- TrueTypePoint[ ][ ] GetGlyphPath (int glyphIndex, double size)

  Get the path that describes the shape of a glyph.
- TrueTypePoint[ ][ ] GetGlyphPath (char glyph, double size)

  Get the path that describes the shape of a glyph.
- double Get1000EmGlyphWidth (char glyph)

  Computes the advance width of a glyph, in thousandths of em unit.
- double Get1000EmGlyphWidth (int glyphIndex)

  Computes the advance width of a glyph, in thousandths of em unit.
- double Get1000EmWinAscent ()

  Computes the font's Win ascent, in thousandths of em unit.
- double Get1000EmAscent ()

  Computes the font ascent, in thousandths of em unit.
- double Get1000EmDescent ()

  Computes the font descent, in thousandths of em unit.
- double Get1000EmYMax ()

  Computes the maximum height over the baseline of the font, in thousandths of em unit.
- double Get1000EmYMin ()

  Computes the maximum depth below the baseline of the font, in thousandths of em unit.
- double Get1000EmXMax ()

Computes the maximum distance to the right of the glyph origin of the font, in thousandths of em unit.

- double Get1000EmXMin ()

  Computes the maximum distance to the left of the glyph origin of the font, in thousandths of em unit.
- Bearings Get1000EmGlyphBearings (char glyph)

  Computes the left- and right- side bearings of a glyph, in thousandths of em unit.
- VerticalMetrics Get1000EmGlyphVerticalMetrics (char glyph)

  Computes the vertical metrics of a glyph, in thousandths of em unit.
- double Get1000EmUnderlinePosition ()

  Computes the distance of the top of the underline from the baseline, in thousandths of em unit.
- double Get1000EmUnderlineThickness ()

  Computes the thickness of the underline, in thousandths of em unit.
- double GetItalicAngle ()

  Computes the italic angle for the current font, in thousandths of em unit. This is computed from the vertical and is negative for text that leans forwards.
- double[ ] Get1000EmUnderlineIntersections (char glyph, double position, double thickness)

  Computes the intersections between an underline at the specified position and thickness and a glyph, in thousandths of em units.
- PairKerning Get1000EmKerning (char glyph1, char glyph2)

  Gets the kerning between two glyphs.
- PairKerning Get1000EmKerning (int glyph1Index, int glyph2Index)

  Gets the kerning between two glyphs.

Properties

- Stream FontStream `[get]`

  A stream pointing to the TrueType file source (either on disk or in memory). Never dispose this stream directly; if you really need to, call Destroy instead.

7.100.1 Detailed Description

Represents a font file in TrueType format. Reference: http://stevehanov.ca/blog/?id=143, https://developer.apple.com/fonts/TrueType-Reference-Manual/, https://docs.microsoft.com/en-us/typography/opentype/spec/

Definition at line 30 of file TrueType.cs.

7.100.2 Member Function Documentation

7.100.2.1 Destroy()

```
void VectSharp.TrueTypeFile.Destroy ( )
```

Remove this TrueType file from the cache, clear the tables and release the FontStream. Only call this when the actual file that was used to create this object needs to be changed!

Definition at line 53 of file TrueType.cs.

7.100.2.2 Get1000EmAscent()

```
double VectSharp.TrueTypeFile.Get1000EmAscent ( )
```

Computes the font ascent, in thousandths of em unit.

Returns

The font ascent in thousandths of em unit.

Definition at line 2131 of file TrueType.cs.

7.100.2.3 Get1000EmDescent()

```
double VectSharp.TrueTypeFile.Get1000EmDescent ( )
```

Computes the font descent, in thousandths of em unit.

Returns

The font descent in thousandths of em unit.

Definition at line 2141 of file TrueType.cs.

7.100.2.4 Get1000EmGlyphBearings()

```
Bearings VectSharp.TrueTypeFile.Get1000EmGlyphBearings (
            char glyph )
```

Computes the left- and right- side bearings of a glyph, in thousandths of em unit.

Parameters

glyph	The glyph whose bearings are to be computed.

Returns

The left- and right- side bearings of the glyph in thousandths of em unit

Definition at line 2223 of file TrueType.cs.

7.100.2.5 Get1000EmGlyphVerticalMetrics()

```
VerticalMetrics VectSharp.TrueTypeFile.Get1000EmGlyphVerticalMetrics (
            char glyph )
```

Computes the vertical metrics of a glyph, in thousandths of em unit.

Parameters

glyph	The glyph whose vertical metrics are to be computed.

Returns

The vertical metrics of a glyph, in thousandths of em unit.

Definition at line 2271 of file TrueType.cs.

7.100.2.6 Get1000EmGlyphWidth() [1/2]

```
double VectSharp.TrueTypeFile.Get1000EmGlyphWidth (
            char glyph )
```

Computes the advance width of a glyph, in thousandths of em unit.

Parameters

glyph	The glyph whose advance width is to be computed.

Returns

The advance width of the glyph in thousandths of em unit.

Definition at line 2082 of file TrueType.cs.

7.100.2.7 Get1000EmGlyphWidth() [2/2]

```
double VectSharp.TrueTypeFile.Get1000EmGlyphWidth (
            int glyphIndex )
```

Computes the advance width of a glyph, in thousandths of em unit.

Parameters

glyphIndex	The index of the glyph whose advance width is to be computed.

Returns

The advance width of the glyph in thousandths of em unit.

Definition at line 2100 of file TrueType.cs.

7.100.2.8 Get1000EmKerning() [1/2]

```
PairKerning VectSharp.TrueTypeFile.Get1000EmKerning (
            char glyph1,
            char glyph2 )
```

Gets the kerning between two glyphs.

Parameters

glyph1	The first glyph of the kerning pair.
glyph2	The second glyph of the kerning pair.

Returns

A PairKerning object containing information about how the position of each glyphs should be altered.

Definition at line 2490 of file TrueType.cs.

7.100.2.9 Get1000EmKerning() [2/2]

```
PairKerning VectSharp.TrueTypeFile.Get1000EmKerning (
            int glyph1Index,
            int glyph2Index )
```

Gets the kerning between two glyphs.

Parameters

glyph1Index	The index of the first glyph of the kerning pair.
glyph2Index	The index of the second glyph of the kerning pair.

Returns

A PairKerning object containing information about how the position of each glyphs should be altered.

Definition at line 2504 of file TrueType.cs.

7.100.2.10 Get1000EmUnderlineIntersections()

```
double[] VectSharp.TrueTypeFile.Get1000EmUnderlineIntersections (
            char glyph,
            double position,
            double thickness )
```

Computes the intersections between an underline at the specified position and thickness and a glyph, in thousandths of em units.

Parameters

glyph	The glyph whose intersections with the underline will be computed.
position	The distance of the top of the underline from the baseline, in thousandths of em unit.
thickness	The thickness of the underline, in thousandths of em unit.

Returns

If the underline does not intersect the glyph, this method returns `null`. Otherwise, it returns an array containing two elements, representing the horizontal coordinates of the leftmost and rightmost intersection points.

Definition at line 2340 of file TrueType.cs.

7.100.2.11 Get1000EmUnderlinePosition()

```
double VectSharp.TrueTypeFile.Get1000EmUnderlinePosition ( )
```

Computes the distance of the top of the underline from the baseline, in thousandths of em unit.

Returns

The distance of the top of the underline from the baseline, in thousandths of em unit.

Definition at line 2288 of file TrueType.cs.

7.100.2.12 Get1000EmUnderlineThickness()

```
double VectSharp.TrueTypeFile.Get1000EmUnderlineThickness ( )
```

Computes the thickness of the underline, in thousandths of em unit.

Returns

The thickness of the underline, in thousandths of em unit.

Definition at line 2304 of file TrueType.cs.

7.100.2.13 Get1000EmWinAscent()

```
double VectSharp.TrueTypeFile.Get1000EmWinAscent ( )
```

Computes the font's Win ascent, in thousandths of em unit.

Returns

The font's Win ascent in thousandths of em unit.

Definition at line 2111 of file TrueType.cs.

7.100.2.14 Get1000EmXMax()

```
double VectSharp.TrueTypeFile.Get1000EmXMax ( )
```

Computes the maximum distance to the right of the glyph origin of the font, in thousandths of em unit.

Returns

The maximum distance to the right of the glyph origin of the font in thousandths of em unit.

Definition at line 2168 of file TrueType.cs.

7.100.2.15 Get1000EmXMin()

```
double VectSharp.TrueTypeFile.Get1000EmXMin ( )
```

Computes the maximum distance to the left of the glyph origin of the font, in thousandths of em unit.

Returns

The maximum distance to the left of the glyph origin of the font in thousandths of em unit.

Definition at line 2177 of file TrueType.cs.

7.100.2.16 Get1000EmYMax()

```
double VectSharp.TrueTypeFile.Get1000EmYMax ( )
```

Computes the maximum height over the baseline of the font, in thousandths of em unit.

Returns

The maximum height over the baseline of the font in thousandths of em unit.

Definition at line 2150 of file TrueType.cs.

7.100.2.17 Get1000EmYMin()

```
double VectSharp.TrueTypeFile.Get1000EmYMin ( )
```

Computes the maximum depth below the baseline of the font, in thousandths of em unit.

Returns

The maximum depth below the baseline of the font in thousandths of em unit.

Definition at line 2159 of file TrueType.cs.

7.100.2.18 GetFirstCharIndex()

```
ushort VectSharp.TrueTypeFile.GetFirstCharIndex ( )
```

Returns the index of the first character glyph represented by the font.

Returns

The index of the first character glyph represented by the font.

Definition at line 1920 of file TrueType.cs.

7.100.2.19 GetFontFamilyName()

```
string VectSharp.TrueTypeFile.GetFontFamilyName ( )
```

Obtains the font family name from the TrueType file.

Returns

The font family name, if available; `null` otherwise.

Definition at line 1846 of file TrueType.cs.

7.100.2.20 GetFontName()

```
string VectSharp.TrueTypeFile.GetFontName ( )
```

Obtains the PostScript font name from the TrueType file.

Returns

The PostScript font name, if available; `null` otherwise.

Definition at line 1901 of file TrueType.cs.

7.100.2.21 GetFullFontFamilyName()

```
string VectSharp.TrueTypeFile.GetFullFontFamilyName ( )
```

Obtains the full font family name from the TrueType file.

Returns

The full font family name, if available; `null` otherwise.

Definition at line 1873 of file TrueType.cs.

7.100.2.22 GetGlyphIndex()

```
int VectSharp.TrueTypeFile.GetGlyphIndex (
            char glyph )
```

Determines the index of the glyph corresponding to a certain character.

Parameters

glyph	The character whose glyph is sought.

Returns

The index of the glyph in the TrueType file.

Definition at line 2010 of file TrueType.cs.

7.100.2.23 GetGlyphPath() [1/2]

```
TrueTypePoint[][] VectSharp.TrueTypeFile.GetGlyphPath (
            char glyph,
            double size )
```

Get the path that describes the shape of a glyph.

Parameters

glyph	The glyph whose path is sought.
size	The font size to be used for the font coordinates.

Returns

An array of contours, each of which is itself an array of TrueType points.

Definition at line 2072 of file TrueType.cs.

7.100.2.24 GetGlyphPath() [2/2]

```
TrueTypePoint[][] VectSharp.TrueTypeFile.GetGlyphPath (
            int glyphIndex,
            double size )
```

Get the path that describes the shape of a glyph.

Parameters

glyphIndex	The index of the glyph whose path is sought.
size	The font size to be used for the font coordinates.

Returns

An array of contours, each of which is itself an array of TrueType points.

Definition at line 2061 of file TrueType.cs.

7.100.2.25 GetItalicAngle()

```
double VectSharp.TrueTypeFile.GetItalicAngle ( )
```

Computes the italic angle for the current font, in thousandths of em unit. This is computed from the vertical and is negative for text that leans forwards.

Returns

Definition at line 2320 of file TrueType.cs.

7.100.2.26 GetLastCharIndex()

```
ushort VectSharp.TrueTypeFile.GetLastCharIndex ( )
```

Returns the index of the last character glyph represented by the font.

Returns

The index of the last character glyph represented by the font.

Definition at line 1931 of file TrueType.cs.

7.100.2.27 IsBold()

```
bool VectSharp.TrueTypeFile.IsBold ( )
```

Determines whether the typeface is Bold or not.

Returns

A bool indicating whether the typeface is Bold or not

Definition at line 1965 of file TrueType.cs.

7.100.2.28 IsFixedPitch()

```
bool VectSharp.TrueTypeFile.IsFixedPitch ( )
```

Determines whether the typeface is fixed-pitch (aka monospaces) or not.

Returns

A bool indicating whether the typeface is fixed-pitch (aka monospaces) or not.

Definition at line 1976 of file TrueType.cs.

7.100.2.29 IsItalic()

```
bool VectSharp.TrueTypeFile.IsItalic ( )
```

Determines whether the typeface is Italic or Oblique or not.

Returns

A bool indicating whether the typeface is Italic or Oblique or not.

Definition at line 1943 of file TrueType.cs.

7.100.2.30 IsOblique()

```
bool VectSharp.TrueTypeFile.IsOblique ( )
```

Determines whether the typeface is Oblique or not.

Returns

A bool indicating whether the typeface is Oblique or not.

Definition at line 1954 of file TrueType.cs.

7.100.2.31 IsScript()

```
bool VectSharp.TrueTypeFile.IsScript ( )
```

Determines whether the typeface is a script typeface or not.

Returns

A bool indicating whether the typeface is a script typeface or not.

Definition at line 1998 of file TrueType.cs.

7.100.2.32 IsSerif()

```
bool VectSharp.TrueTypeFile.IsSerif ( )
```

Determines whether the typeface is serifed or not.

Returns

A bool indicating whether the typeface is serifed or not.

Definition at line 1987 of file TrueType.cs.

7.100.2.33 SubsetFont()

```
TrueTypeFile VectSharp.TrueTypeFile.SubsetFont (
            string charactersToInclude,
            bool consolidateAt32 = false,
            Dictionary< char, char > outputEncoding = null )
```

Create a subset of the TrueType file, containing only the glyphs for the specified characters.

Parameters

charactersToInclude	A string containing the characters for which the glyphs should be included.
consolidateAt32	If true, the character map is rearranged so that the included glyphs start at the unicode U+0032 control point.
outputEncoding	If *consolidateAt32* is true, entries will be added to this dictionary mapping the original characters to the new map (that starts at U+0033).

Returns

Definition at line 562 of file TrueType.cs.

7.100.3 Property Documentation

7.100.3.1 FontStream

```
Stream VectSharp.TrueTypeFile.FontStream [get]
```

A stream pointing to the TrueType file source (either on disk or in memory). Never dispose this stream directly; if you really need to, call Destroy instead.

Definition at line 47 of file TrueType.cs.

The documentation for this class was generated from the following file:

- VectSharp/TrueType.cs

7.101 VectSharp.TrueTypeFile.TrueTypePoint Struct Reference

Represents a point in a TrueType path description.

Public Attributes

- double X

  The horizontal coordinate of the point.
- double Y

  The vertical coordinate of the point.
- bool IsOnCurve

  Whether the point is a point on the curve, or a control point of a quadratic Bezier curve.

7.101.1 Detailed Description

Represents a point in a TrueType path description.

Definition at line 1360 of file TrueType.cs.

7.101.2 Member Data Documentation

7.101.2.1 IsOnCurve

```
bool VectSharp.TrueTypeFile.TrueTypePoint.IsOnCurve
```

Whether the point is a point on the curve, or a control point of a quadratic Bezier curve.

Definition at line 1375 of file TrueType.cs.

7.101.2.2 X

```
double VectSharp.TrueTypeFile.TrueTypePoint.X
```

The horizontal coordinate of the point.

Definition at line 1365 of file TrueType.cs.

7.101.2.3 Y

```
double VectSharp.TrueTypeFile.TrueTypePoint.Y
```

The vertical coordinate of the point.

Definition at line 1370 of file TrueType.cs.

The documentation for this struct was generated from the following file:

- VectSharp/TrueType.cs

7.102 VectSharp.UnbalancedStackException Class Reference

The exception that is thrown when an unbalanced graphics state stack occurs.

Inheritance diagram for VectSharp.UnbalancedStackException:

7.102.1 Detailed Description

The exception that is thrown when an unbalanced graphics state stack occurs.

Definition at line 253 of file Graphics.cs.

The documentation for this class was generated from the following file:

- VectSharp/Graphics.cs

7.103 VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.↩ UnknownFormatException Class Reference

The exception that is raised when the output file format is not specified and the file name does not have an extension corresponding to a known file format.

Inheritance diagram for VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.UnknownFormatException:

Properties

- string Format `[get]`

  The extension of the file that does not correspond to any known file format.

7.103.1 Detailed Description

The exception that is raised when the output file format is not specified and the file name does not have an extension corresponding to a known file format.

Definition at line 1100 of file ImageSharpContext.cs.

7.103.2 Property Documentation

7.103.2.1 Format

```
string VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.UnknownFormatException.Format
[get]
```

The extension of the file that does not correspond to any known file format.

Definition at line 1105 of file ImageSharpContext.cs.

The documentation for this class was generated from the following file:

- VectSharp.Raster.ImageSharp/ImageSharpContext.cs

7.104 VectSharp.TrueTypeFile.VerticalMetrics Struct Reference

Represents the maximum heigth above and depth below the baseline of a glyph.

Public Attributes

- int YMin

  The maximum depth below the baseline of the glyph.
- int YMax

  The maximum height above the baseline of the glyph.

7.104.1 Detailed Description

Represents the maximum heigth above and depth below the baseline of a glyph.

Definition at line 2240 of file TrueType.cs.

7.104.2 Member Data Documentation

7.104.2.1 YMax

```
int VectSharp.TrueTypeFile.VerticalMetrics.YMax
```

The maximum height above the baseline of the glyph.

Definition at line 2250 of file TrueType.cs.

7.104.2.2 YMin

```
int VectSharp.TrueTypeFile.VerticalMetrics.YMin
```

The maximum depth below the baseline of the glyph.

Definition at line 2245 of file TrueType.cs.

The documentation for this struct was generated from the following file:

- VectSharp/TrueType.cs

Chapter 8

File Documentation

8.1 AnimatedCanvas.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2022 Giorgio Bianchini, University of Bristol

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using Avalonia;
using Avalonia.Media;
using System;
using System.Collections.Generic;

namespace VectSharp.Canvas
{
    /// <summary>
    /// An <see cref="Avalonia.Controls.Canvas"/> containing an animation.
    /// </summary>
    public class AnimatedCanvas : Avalonia.Controls.Canvas
    {
        /// <summary>
        /// Defines the <see cref="CurrentFrame"/> property.
        /// </summary>
        public static readonly StyledProperty<int> CurrentFrameProperty = AvaloniaProperty.Register<AnimatedCanvas, int>(nameof(CurrentFrame), coerce: (ownerObject, val) =>
        {
            AnimatedCanvas owner = (AnimatedCanvas)ownerObject;

            if (owner.maxFrameIndex > 0)
            {
                if (val < owner.maxFrameIndex)
                {
                    return val;
                }
                else
                {
                    owner.IsPlaying = false;
                    return owner.maxFrameIndex - 1;
                }
            }
            else
            {
                return val % owner.RenderActions.Length;
            }
        });

        /// <summary>
        /// The current frame in the animation.
        /// </summary>
```

```
        public int CurrentFrame
        {
            get { return GetValue(CurrentFrameProperty); }
            set { SetValue(CurrentFrameProperty, value); }
        }

        /// <summary>
        /// Defines the <see cref="FrameCount"/> property.
        /// </summary>
        public static readonly DirectProperty<AnimatedCanvas, int> FrameCountProperty =
AvaloniaProperty.RegisterDirect<AnimatedCanvas, int>(nameof(FrameCount), o => o.maxFrameIndex);

        private int frameCount;

        /// <summary>
        /// The number of frames in the animation.
        /// </summary>
        public int FrameCount
        {
            get
            {
                return frameCount;
            }
            private set
            {
                SetAndRaise(FrameCountProperty, ref frameCount, value);
            }
        }

        private int maxFrameIndex { get; }

        /// <summary>
        /// Defines the <see cref="FrameRate"/> property.
        /// </summary>
        public static readonly DirectProperty<AnimatedCanvas, double> FrameRateProperty =
AvaloniaProperty.RegisterDirect<AnimatedCanvas, double>(nameof(FrameRate), o => o.FrameRate);

        private double frameRate;

        /// <summary>
        /// The target frame rate of the animation.
        /// </summary>
        public double FrameRate
        {
            get
            {
                return frameRate;
            }
            private set
            {
                SetAndRaise(FrameRateProperty, ref frameRate, value);
            }
        }



        /// <summary>
        /// Defines the <see cref="IsPlaying"/> property.
        /// </summary>
        public static readonly StyledProperty<bool> IsPlayingProperty =
AvaloniaProperty.Register<AnimatedCanvas, bool>(nameof(IsPlaying), false, coerce: (ownerObject, val)
=>
        {
            AnimatedCanvas owner = (AnimatedCanvas)ownerObject;

            if (owner.IsPlaying && !val)
            {
                owner.RefreshTimer.Dispose();
            }
            else if (!owner.IsPlaying && val)
            {
                owner.RefreshTimer = new System.Threading.Timer(_ =>
                {
                    _ = Avalonia.Threading.Dispatcher.UIThread.InvokeAsync(() =>
                    {
                        owner.CurrentFrame++;
                    });
                }, null, 0, (long)(1000.0 / owner.FrameRate));
            }

            return val;
        });

        /// <summary>
        /// The current frame in the animation.
        /// </summary>
        public bool IsPlaying
```

```
        {
            get { return GetValue(IsPlayingProperty); }
            set { SetValue(IsPlayingProperty, value); }
        }


        private List<RenderAction>[] RenderActions;

        static Avalonia.Point Origin = new Avalonia.Point(0, 0);

        private SolidColorBrush BackgroundBrush;

        private Dictionary<string, (IImage, bool)> Images;

        private System.Threading.Timer RefreshTimer { get; set; }

        static AnimatedCanvas()
        {
            AffectsRender<AnimatedCanvas>(CurrentFrameProperty);
        }

        internal AnimatedCanvas(VectSharp.Animation animation, double durationScaling, double frameRate, AvaloniaContextInterpreter.TextOptions textOption, FilterOption filterOption)
        {
            Colour backgroundColour = animation.Background;

            this.BackgroundBrush = new SolidColorBrush(Color.FromArgb((byte)(backgroundColour.A * 255), (byte)(backgroundColour.R * 255), (byte)(backgroundColour.G * 255), (byte)(backgroundColour.B * 255)));

            this.Width = animation.Width;
            this.Height = animation.Height;
            this.Images = new Dictionary<string, (IImage, bool)>();

            int frames = (int)Math.Ceiling(animation.Duration * frameRate * durationScaling / 1000);

            this.RenderActions = new List<RenderAction>[frames];

            for (int i = 0; i < frames; i++)
            {
                double frameTime = i / frameRate / durationScaling * 1000;

                Page pag = animation.GetFrameAtAbsolute(frameTime);

                AvaloniaDrawingContext ctx = new AvaloniaDrawingContext(this.Width, this.Height, false, textOption, this.Images, filterOption);

                pag.Graphics.CopyToIGraphicsContext(ctx);
                this.RenderActions[i] = ctx.RenderActions;
            }

            if (animation.RepeatCount > 0)
            {
                this.maxFrameIndex = animation.RepeatCount * frames;
            }
            else
            {
                this.maxFrameIndex = 0;
            }

            this.FrameCount = frames;

            this.FrameRate = frameRate;
        }

        /// <inheritdoc/>
        public override void Render(DrawingContext context)
        {
            context.FillRectangle(this.BackgroundBrush, new Avalonia.Rect(0, 0, Width, Height));

            foreach (RenderAction act in this.RenderActions[this.CurrentFrame % this.RenderActions.Length])
            {
                if (act.ActionType == RenderAction.ActionTypes.Path)
                {
                    DrawingContext.PushedState? state = null;

                    if (act.ClippingPath != null)
                    {
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                        state = context.PushGeometryClip(act.ClippingPath);
                    }

                    using (context.PushPreTransform(act.Transform))
                    {
```

```
                        context.DrawGeometry(act.Fill, act.Stroke, act.Geometry);
                    }

                    if (state != null)
                    {
                        state?.Dispose();
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                    }
                }
                else if (act.ActionType == RenderAction.ActionTypes.Text)
                {
                    DrawingContext.PushedState? state = null;

                    if (act.ClippingPath != null)
                    {
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                        state = context.PushGeometryClip(act.ClippingPath);
                    }

                    using (context.PushPreTransform(act.Transform))
                    {
                        context.DrawText(act.Fill, Origin, act.Text);
                    }

                    if (state != null)
                    {
                        state?.Dispose();
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                    }
                }
                else if (act.ActionType == RenderAction.ActionTypes.RasterImage)
                {
                    DrawingContext.PushedState? state = null;

                    if (act.ClippingPath != null)
                    {
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                        state = context.PushGeometryClip(act.ClippingPath);
                    }

                    (IImage, bool) image = Images[act.ImageId];

                    using (context.PushPreTransform(act.Transform))
                    {
                        context.DrawImage(image.Item1, act.ImageSource.Value, act.ImageDestination.Value, image.Item2 ? Avalonia.Visuals.Media.Imaging.BitmapInterpolationMode.HighQuality : Avalonia.Visuals.Media.Imaging.BitmapInterpolationMode.Default);
                    }

                    if (state != null)
                    {
                        state?.Dispose();
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                    }
                }
            }

        }

        /// <inheritdoc/>
        protected override void OnDetachedFromVisualTree(VisualTreeAttachmentEventArgs e)
        {
            base.OnDetachedFromVisualTree(e);
            IsPlaying = false;
        }

        /// <inheritdoc/>
        protected override void OnAttachedToVisualTree(VisualTreeAttachmentEventArgs e)
        {
            base.OnAttachedToVisualTree(e);
            IsPlaying = true;
        }
    }

}
```

8.2 AvaloniaContext.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using Avalonia.Controls;
using Avalonia.Controls.Shapes;
using Avalonia.Media;
using Avalonia.Media.Imaging;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using VectSharp.Filters;

namespace VectSharp.Canvas
{
    internal static class MatrixUtils
    {
        public static Avalonia.Matrix ToAvaloniaMatrix(this double[,] matrix)
        {
            return new Avalonia.Matrix(matrix[0, 0], matrix[1, 0], matrix[0, 1], matrix[1, 1], matrix[0, 2], matrix[1, 2]);
        }

        public static double[] Multiply(double[,] matrix, double[] vector)
        {
            double[] tbr = new double[2];

            tbr[0] = matrix[0, 0] * vector[0] + matrix[0, 1] * vector[1] + matrix[0, 2];
            tbr[1] = matrix[1, 0] * vector[0] + matrix[1, 1] * vector[1] + matrix[1, 2];

            return tbr;
        }

        public static double[,] Multiply(double[,] matrix1, double[,] matrix2)
        {
            double[,] tbr = new double[3, 3];

            for (int i = 0; i < 3; i++)
            {
                for (int j = 0; j < 3; j++)
                {
                    for (int k = 0; k < 3; k++)
                    {
                        tbr[i, j] += matrix1[i, k] * matrix2[k, j];
                    }
                }
            }

            return tbr;
        }

        public static double[,] Rotate(double[,] matrix, double angle)
        {
            double[,] rotationMatrix = new double[3, 3];
            rotationMatrix[0, 0] = Math.Cos(angle);
            rotationMatrix[0, 1] = -Math.Sin(angle);
            rotationMatrix[1, 0] = Math.Sin(angle);
            rotationMatrix[1, 1] = Math.Cos(angle);
            rotationMatrix[2, 2] = 1;

            return Multiply(matrix, rotationMatrix);
        }

        public static double[,] Translate(double[,] matrix, double x, double y)
        {
            double[,] translationMatrix = new double[3, 3];
            translationMatrix[0, 0] = 1;
            translationMatrix[0, 2] = x;
            translationMatrix[1, 1] = 1;
            translationMatrix[1, 2] = y;
            translationMatrix[2, 2] = 1;
```

```
            return Multiply(matrix, translationMatrix);
        }

        public static double[,] Scale(double[,] matrix, double scaleX, double scaleY)
        {
            double[,] scaleMatrix = new double[3, 3];
            scaleMatrix[0, 0] = scaleX;
            scaleMatrix[1, 1] = scaleY;
            scaleMatrix[2, 2] = 1;

            return Multiply(matrix, scaleMatrix);
        }

        public static double[,] Identity = new double[,] { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };
        public static double[,] Invert(double[,] m)
        {
            double[,] tbr = new double[3, 3];

            tbr[0, 0] = (m[1, 1] * m[2, 2] - m[1, 2] * m[2, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[0, 1] = -(m[0, 1] * m[2, 2] - m[0, 2] * m[2, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[0, 2] = (m[0, 1] * m[1, 2] - m[0, 2] * m[1, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 0] = -(m[1, 0] * m[2, 2] - m[1, 2] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 1] = (m[0, 0] * m[2, 2] - m[0, 2] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 2] = -(m[0, 0] * m[1, 2] - m[0, 2] * m[1, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 0] = (m[1, 0] * m[2, 1] - m[1, 1] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 1] = -(m[0, 0] * m[2, 1] - m[0, 1] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 2] = (m[0, 0] * m[1, 1] - m[0, 1] * m[1, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);

            return tbr;
        }
    }

    internal static class Utils
    {
        public static void CoerceNaNAndInfinityToZero(ref double val)
        {
            if (double.IsNaN(val) || double.IsInfinity(val) || val == double.MinValue || val == double.MaxValue)
            {
                val = 0;
            }
        }

        public static void CoerceNaNAndInfinityToZero(ref double val1, ref double val2)
        {
            if (double.IsNaN(val1) || double.IsInfinity(val1) || val1 == double.MinValue || val1 == double.MaxValue)
            {
                val1 = 0;
            }

            if (double.IsNaN(val2) || double.IsInfinity(val2) || val2 == double.MinValue || val2 == double.MaxValue)
            {
                val2 = 0;
            }
        }

        public static void CoerceNaNAndInfinityToZero(ref double val1, ref double val2, ref double val3, ref double val4)
        {
            if (double.IsNaN(val1) || double.IsInfinity(val1) || val1 == double.MinValue || val1 == double.MaxValue)
            {
                val1 = 0;
            }

            if (double.IsNaN(val2) || double.IsInfinity(val2) || val2 == double.MinValue || val2 ==
```

```
double.MaxValue)
            {
                val2 = 0;
            }

            if (double.IsNaN(val3) || double.IsInfinity(val3) || val3 == double.MinValue || val3 == double.MaxValue)
            {
                val3 = 0;
            }

            if (double.IsNaN(val4) || double.IsInfinity(val4) || val4 == double.MinValue || val4 == double.MaxValue)
            {
                val4 = 0;
            }
        }

        public static void CoerceNaNAndInfinityToZero(ref double val1, ref double val2, ref double val3, ref double val4, ref double val5, ref double val6)
        {
            if (double.IsNaN(val1) || double.IsInfinity(val1) || val1 == double.MinValue || val1 == double.MaxValue)
            {
                val1 = 0;
            }

            if (double.IsNaN(val2) || double.IsInfinity(val2) || val2 == double.MinValue || val2 == double.MaxValue)
            {
                val2 = 0;
            }

            if (double.IsNaN(val3) || double.IsInfinity(val3) || val3 == double.MinValue || val3 == double.MaxValue)
            {
                val3 = 0;
            }

            if (double.IsNaN(val4) || double.IsInfinity(val4) || val4 == double.MinValue || val4 == double.MaxValue)
            {
                val4 = 0;
            }

            if (double.IsNaN(val5) || double.IsInfinity(val5) || val5 == double.MinValue || val5 == double.MaxValue)
            {
                val5 = 0;
            }

            if (double.IsNaN(val6) || double.IsInfinity(val6) || val6 == double.MinValue || val6 == double.MaxValue)
            {
                val6 = 0;
            }
        }
    }

    internal class AvaloniaContext : IGraphicsContext
    {
        public Dictionary<string, Delegate> TaggedActions { get; set; } = new Dictionary<string, Delegate>();

        private bool removeTaggedActions = true;

        public string Tag { get; set; }

        private AvaloniaContextInterpreter.TextOptions _textOption;

        private Avalonia.Controls.Canvas currControlElement;

        private Stack<Avalonia.Controls.Canvas> controlElements;
        private FilterOption _filterOption;

        public AvaloniaContext(double width, double height, bool removeTaggedActionsAfterExecution, AvaloniaContextInterpreter.TextOptions textOption, FilterOption filterOption)
        {
            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
            ControlItem = new Avalonia.Controls.Canvas() { Width = width, Height = height, ClipToBounds = true };
            removeTaggedActions = removeTaggedActionsAfterExecution;

            _transform = new double[3, 3];

```

```
            _transform[0, 0] = 1;
            _transform[1, 1] = 1;
            _transform[2, 2] = 1;

            states = new Stack<double[,]>();

            _textOption = textOption;

            currControlElement = ControlItem;
            controlElements = new Stack<Avalonia.Controls.Canvas>();
            controlElements.Push(ControlItem);
            _filterOption = filterOption;
        }

        public Avalonia.Controls.Canvas ControlItem { get; }

        public double Width { get { return ControlItem.Width; } }
        public double Height { get { return ControlItem.Height; } }

        public void Translate(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            _transform = MatrixUtils.Translate(_transform, x, y);

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
        }

        public TextBaselines TextBaseline { get; set; }

        private void PathText(string text, double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            GraphicsPath textPath = new GraphicsPath().AddText(x, y, text, Font, TextBaseline);

            for (int j = 0; j < textPath.Segments.Count; j++)
            {
                switch (textPath.Segments[j].Type)
                {
                    case VectSharp.SegmentType.Move:
                        this.MoveTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.Line:
                        this.LineTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.CubicBezier:
                        this.CubicBezierTo(textPath.Segments[j].Points[0].X,
textPath.Segments[j].Points[0].Y, textPath.Segments[j].Points[1].X, textPath.Segments[j].Points[1].Y,
textPath.Segments[j].Points[2].X, textPath.Segments[j].Points[2].Y);
                        break;
                    case VectSharp.SegmentType.Close:
                        this.Close();
                        break;
                }
            }
        }

        public void StrokeText(string text, double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            PathText(text, x, y);
            Stroke();
        }

        public void FillText(string text, double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if (_textOption == AvaloniaContextInterpreter.TextOptions.NeverConvert || (_textOption ==
AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary && Font.FontFamily.IsStandardFamily &&
Font.FontFamily.FileName != "ZapfDingbats" && Font.FontFamily.FileName != "Symbol"))
            {
                TextBlock blk = new TextBlock() { ClipToBounds = false, Text = text, FontFamily =
Avalonia.Media.FontFamily.Parse(FontFamily), FontSize = Font.FontSize, FontStyle =
(Font.FontFamily.IsOblique ? FontStyle.Oblique : Font.FontFamily.IsItalic ? FontStyle.Italic :
FontStyle.Normal), FontWeight = (Font.FontFamily.IsBold ? FontWeight.Bold : FontWeight.Regular) };

                double top = y;
                double left = x;

                double[,] currTransform = null;
                double[,] deltaTransform = MatrixUtils.Identity;

```

```
                if (Font.FontFamily.TrueTypeFile != null)
                {
                    currTransform = MatrixUtils.Translate(_transform, x, y);
                }

                Font.DetailedFontMetrics metrics = Font.MeasureTextAdvanced(text);

                if (TextBaseline == TextBaselines.Top)
                {
                    blk.VerticalAlignment = Avalonia.Layout.VerticalAlignment.Top;

                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top + metrics.Top - Font.WinAscent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top + metrics.Top - Font.WinAscent);
                        }
                        else
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top + metrics.Top - Font.Ascent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top + metrics.Top - Font.Ascent);
                        }
                    }
                }
                else if (TextBaseline == TextBaselines.Middle)
                {
                    blk.VerticalAlignment = Avalonia.Layout.VerticalAlignment.Top;

                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top + metrics.Top / 2 + metrics.Bottom / 2 - Font.WinAscent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top + metrics.Top / 2 + metrics.Bottom / 2 - Font.WinAscent);
                        }
                        else
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top + metrics.Top / 2 + metrics.Bottom / 2 - Font.Ascent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top + metrics.Top / 2 + metrics.Bottom / 2 - Font.Ascent);
                        }

                    }
                }
                else if (TextBaseline == TextBaselines.Baseline)
                {
                    double lsb = Font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(text[0]).LeftSideBearing * Font.FontSize / 1000;

                    blk.VerticalAlignment = Avalonia.Layout.VerticalAlignment.Top;

                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - lsb, top - Font.WinAscent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - lsb, top - Font.YMax);
                        }
                        else
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - lsb, top - Font.Ascent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - lsb, top - Font.YMax);
                        }
                    }
                }
                else if (TextBaseline == TextBaselines.Bottom)
                {
                    blk.VerticalAlignment = Avalonia.Layout.VerticalAlignment.Bottom;

                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top - Font.WinAscent + metrics.Bottom);
```

```
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top - Font.WinAscent + metrics.Bottom);
                        }
                        else
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top - Font.Ascent + metrics.Bottom);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top - Font.Ascent + metrics.Bottom);
                        }
                    }
                }

                blk.RenderTransform = new MatrixTransform(currTransform.ToAvaloniaMatrix());
                blk.RenderTransformOrigin = new Avalonia.RelativePoint(0, 0, Avalonia.RelativeUnit.Absolute);

                Avalonia.Media.Brush foreground = null;

                if (this.FillStyle is SolidColourBrush solid)
                {
                    foreground = new SolidColorBrush(Color.FromArgb(FillAlpha, (byte)(solid.R * 255), (byte)(solid.G * 255), (byte)(solid.B * 255)));
                }
                else if (this.FillStyle is LinearGradientBrush linearGradient)
                {
                    foreground = linearGradient.ToLinearGradientBrush(deltaTransform);
                }
                else if (this.FillStyle is RadialGradientBrush radialGradient)
                {
                    foreground = radialGradient.ToRadialGradientBrush(metrics.Width + metrics.LeftSideBearing + metrics.RightSideBearing, deltaTransform);
                }

                blk.Foreground = foreground;

                currControlElement.Children.Add(blk);

                if (!string.IsNullOrEmpty(Tag))
                {
                    if (TaggedActions.ContainsKey(Tag))
                    {
                        TaggedActions[Tag].DynamicInvoke(blk);

                        if (removeTaggedActions)
                        {
                            TaggedActions.Remove(Tag);
                        }
                    }
                }
            }
            else
            {
                PathText(text, x, y);
                Fill();
            }
        }

        public Brush StrokeStyle { get; private set; } = Colour.FromRgb(0, 0, 0);
        private byte StrokeAlpha = 255;

        public Brush FillStyle { get; private set; } = Colour.FromRgb(0, 0, 0);
        private byte FillAlpha = 255;

        public void SetFillStyle((int r, int g, int b, double a) style)
        {
            FillStyle = Colour.FromRgba(style.r, style.g, style.b, (int)(style.a * 255));
            FillAlpha = (byte)(style.a * 255);
        }

        public void SetFillStyle(Brush style)
        {
            FillStyle = style;

            if (style is SolidColourBrush solid)
            {
                FillAlpha = (byte)(solid.A * 255);
            }
            else
            {
                FillAlpha = 255;
            }
        }

        public void SetStrokeStyle((int r, int g, int b, double a) style)
        {
            StrokeStyle = Colour.FromRgba(style.r, style.g, style.b, (int)(style.a * 255));
```

```
            StrokeAlpha = (byte)(style.a * 255);
        }

        public void SetStrokeStyle(Brush style)
        {
            StrokeStyle = style;

            if (style is SolidColourBrush solid)
            {
                StrokeAlpha = (byte)(solid.A * 255);
            }
            else
            {
                StrokeAlpha = 255;
            }
        }

        private double[] LineDash;

        public void SetLineDash(LineDash dash)
        {
            LineDash = new double[] { dash.UnitsOn, dash.UnitsOff, dash.Phase };
        }

        public void Rotate(double angle)
        {
            Utils.CoerceNaNAndInfinityToZero(ref angle);

            _transform = MatrixUtils.Rotate(_transform, angle);

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
        }

        public void Transform(double a, double b, double c, double d, double e, double f)
        {
            Utils.CoerceNaNAndInfinityToZero(ref a, ref b, ref c, ref d, ref e, ref f);

            double[,] transfMatrix = new double[3, 3] { { a, c, e }, { b, d, f }, { 0, 0, 1 } };
            _transform = MatrixUtils.Multiply(_transform, transfMatrix);

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
        }

        public void Scale(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            _transform = MatrixUtils.Scale(_transform, x, y);

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
        }

        private double[,] _transform;

        private readonly Stack<double[,]> states;

        public void Save()
        {
            states.Push((double[,])_transform.Clone());
            controlElements.Push(currControlElement);
        }

        public void Restore()
        {
            _transform = states.Pop();
            currControlElement = controlElements.Pop();
        }

        public double LineWidth { get; set; }
        public LineCaps LineCap { get; set; }
        public LineJoins LineJoin { get; set; }

        private string FontFamily;
        private Font _Font;

        public Font Font
        {
            get
            {
                return _Font;
            }
```

```
            set
            {
                _Font = value;

                if (Font.FontFamily is ResourceFontFamily fam)
                {
                    FontFamily = fam.ResourceName;
                }
                else
                {
                    if (!Font.FontFamily.IsStandardFamily)
                    {
                        if (Font.FontFamily.TrueTypeFile != null)
                        {
                            FontFamily = Font.FontFamily.TrueTypeFile.GetFontFamilyName();
                        }
                        else
                        {
                            FontFamily = Font.FontFamily.FileName;
                        }
                    }
                    else
                    {
                        FontFamily = "resm:VectSharp.StandardFonts.?assembly=VectSharp#" + Font.FontFamily.TrueTypeFile.GetFontFamilyName();
                    }
                }
            }
        }

        public (double Width, double Height) MeasureText(string text)
        {
            Avalonia.Media.FormattedText txt = new Avalonia.Media.FormattedText() { Text = text, Typeface = new Typeface(FontFamily), FontSize = Font.FontSize };
            return (txt.Bounds.Width, txt.Bounds.Height);
        }

        private PathGeometry currentPath;
        private PathFigure currentFigure;

        private bool figureInitialised = false;

        public void MoveTo(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if (figureInitialised)
            {
                currentPath.Figures.Add(currentFigure);
            }

            currentFigure = new PathFigure() { StartPoint = new Avalonia.Point(x, y), IsClosed = false };
            figureInitialised = true;
        }

        public void LineTo(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if (figureInitialised)
            {
                currentFigure.Segments.Add(new Avalonia.Media.LineSegment() { Point = new Avalonia.Point(x, y) });
            }
            else
            {
                currentFigure = new PathFigure() { StartPoint = new Avalonia.Point(x, y), IsClosed = false };
                figureInitialised = true;
            }
        }

        public void Rectangle(double x0, double y0, double width, double height)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x0, ref y0, ref width, ref height);

            if (currentFigure != null && figureInitialised)
            {
                currentPath.Figures.Add(currentFigure);
            }

            currentFigure = new PathFigure() { StartPoint = new Avalonia.Point(x0, y0), IsClosed = false };
            currentFigure.Segments.Add(new Avalonia.Media.LineSegment() { Point = new Avalonia.Point(x0 + width, y0) });
```

```
            currentFigure.Segments.Add(new Avalonia.Media.LineSegment() { Point = new Avalonia.Point(x0 + width, y0 + height) });
            currentFigure.Segments.Add(new Avalonia.Media.LineSegment() { Point = new Avalonia.Point(x0, y0 + height) });
            currentFigure.IsClosed = true;

            currentPath.Figures.Add(currentFigure);
            figureInitialised = false;
        }

        public void CubicBezierTo(double p1X, double p1Y, double p2X, double p2Y, double p3X, double p3Y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref p1X, ref p1Y, ref p2X, ref p2Y, ref p3X, ref p3Y);

            if (figureInitialised)
            {
                currentFigure.Segments.Add(new Avalonia.Media.BezierSegment() { Point1 = new Avalonia.Point(p1X, p1Y), Point2 = new Avalonia.Point(p2X, p2Y), Point3 = new Avalonia.Point(p3X, p3Y) });
            }
            else
            {
                currentFigure = new PathFigure() { StartPoint = new Avalonia.Point(p1X, p1Y), IsClosed = false };
                figureInitialised = true;
            }
        }

        public void Close()
        {
            currentFigure.IsClosed = true;
            currentPath.Figures.Add(currentFigure);
            figureInitialised = false;
        }

        public void Stroke()
        {
            if (figureInitialised)
            {
                currentFigure.IsClosed = false;
                currentPath.Figures.Add(currentFigure);
            }

            Avalonia.Media.Brush stroke = null;

            if (this.StrokeStyle is SolidColourBrush solid)
            {
                stroke = new SolidColorBrush(Color.FromArgb(StrokeAlpha, (byte)(solid.R * 255), (byte)(solid.G * 255), (byte)(solid.B * 255)));
            }
            else if (this.StrokeStyle is LinearGradientBrush linearGradient)
            {
                stroke = linearGradient.ToLinearGradientBrush();
            }
            else if (this.StrokeStyle is RadialGradientBrush radialGradient)
            {
                stroke = radialGradient.ToRadialGradientBrush(currentPath.Bounds.Width);
            }


            Path pth = new Path() { Fill = null, Stroke = stroke, StrokeThickness = LineWidth, StrokeDashArray = new Avalonia.Collections.AvaloniaList<double> { (LineDash[0] + (LineCap == LineCaps.Butt ? 0 : LineWidth)) / LineWidth, (LineDash[1] - (LineCap == LineCaps.Butt ? 0 : LineWidth)) / LineWidth }, StrokeDashOffset = LineDash[2] / LineWidth };

            switch (LineCap)
            {
                case LineCaps.Butt:
                    pth.StrokeLineCap = PenLineCap.Flat;
                    break;
                case LineCaps.Round:
                    pth.StrokeLineCap = PenLineCap.Round;
                    break;
                case LineCaps.Square:
                    pth.StrokeLineCap = PenLineCap.Square;
                    break;
            }

            switch (LineJoin)
            {
                case LineJoins.Bevel:
                    pth.StrokeJoin = PenLineJoin.Bevel;
                    break;
                case LineJoins.Round:
                    pth.StrokeJoin = PenLineJoin.Round;
                    break;
```

```
                case LineJoins.Miter:
                    pth.StrokeJoin = PenLineJoin.Miter;
                    break;
            }

            pth.Data = currentPath;

            pth.RenderTransform = new MatrixTransform(_transform.ToAvaloniaMatrix());
            pth.RenderTransformOrigin = new Avalonia.RelativePoint(0, 0,
    Avalonia.RelativeUnit.Absolute);

            currControlElement.Children.Add(pth);

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;

            if (!string.IsNullOrEmpty(Tag))
            {
                if (TaggedActions.ContainsKey(Tag))
                {
                    TaggedActions[Tag].DynamicInvoke(pth);

                    if (removeTaggedActions)
                    {
                        TaggedActions.Remove(Tag);
                    }
                }
            }
        }

        public void Fill()
        {
            if (figureInitialised)
            {
                currentPath.Figures.Add(currentFigure);
            }

            Avalonia.Media.Brush fill = null;

            if (this.FillStyle is SolidColourBrush solid)
            {
                fill = new SolidColorBrush(Color.FromArgb(FillAlpha, (byte)(solid.R * 255),
    (byte)(solid.G * 255), (byte)(solid.B * 255)));
            }
            else if (this.FillStyle is LinearGradientBrush linearGradient)
            {
                fill = linearGradient.ToLinearGradientBrush();
            }
            else if (this.FillStyle is RadialGradientBrush radialGradient)
            {
                fill = radialGradient.ToRadialGradientBrush(currentPath.Bounds.Width);
            }

            Path pth = new Path() { Fill = fill, Stroke = null };

            pth.Data = currentPath;

            pth.RenderTransform = new MatrixTransform(_transform.ToAvaloniaMatrix());
            pth.RenderTransformOrigin = new Avalonia.RelativePoint(0, 0,
    Avalonia.RelativeUnit.Absolute);

            currControlElement.Children.Add(pth);

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;

            if (!string.IsNullOrEmpty(Tag))
            {
                if (TaggedActions.ContainsKey(Tag))
                {
                    TaggedActions[Tag].DynamicInvoke(pth);

                    if (removeTaggedActions)
                    {
                        TaggedActions.Remove(Tag);
                    }
                }
            }
        }

        public void SetClippingPath()
        {
            if (figureInitialised)
            {
                currentPath.Figures.Add(currentFigure);
```

```
            }

            Avalonia.Controls.Canvas newControlElement = new Avalonia.Controls.Canvas();

            newControlElement.Clip = currentPath;

            newControlElement.RenderTransformOrigin = new Avalonia.RelativePoint(0, 0, Avalonia.RelativeUnit.Absolute);
            newControlElement.RenderTransform = new MatrixTransform(_transform.ToAvaloniaMatrix());

            _transform = new double[3, 3];

            _transform[0, 0] = 1;
            _transform[1, 1] = 1;
            _transform[2, 2] = 1;

            currControlElement.Children.Add(newControlElement);
            currControlElement = newControlElement;

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
        }


        public void DrawRasterImage(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image)
        {
            Utils.CoerceNaNAndInfinityToZero(ref destinationX, ref destinationY, ref destinationWidth, ref destinationHeight);

            Image img = new Image() { Source = new CroppedBitmap(new Bitmap(image.PNGStream), new Avalonia.PixelRect(sourceX, sourceY, sourceWidth, sourceHeight)), Width = destinationWidth, Height = destinationHeight };

            if (image.Interpolate)
            {
                RenderOptions.SetBitmapInterpolationMode(img, Avalonia.Visuals.Media.Imaging.BitmapInterpolationMode.HighQuality);
            }
            else
            {
                RenderOptions.SetBitmapInterpolationMode(img, Avalonia.Visuals.Media.Imaging.BitmapInterpolationMode.Default);
            }

            double[,] transf = MatrixUtils.Translate(_transform, destinationX, destinationY);
            img.RenderTransform = new MatrixTransform(transf.ToAvaloniaMatrix());
            img.RenderTransformOrigin = new Avalonia.RelativePoint(0, 0, Avalonia.RelativeUnit.Absolute);

            currControlElement.Children.Add(img);

            if (!string.IsNullOrEmpty(Tag))
            {
                if (TaggedActions.ContainsKey(Tag))
                {
                    TaggedActions[Tag].DynamicInvoke(img);

                    if (removeTaggedActions)
                    {
                        TaggedActions.Remove(Tag);
                    }
                }
            }
        }

        public void DrawFilteredGraphics(Graphics graphics, IFilter filter)
        {
            if (this._filterOption.Operation == FilterOption.FilterOperations.RasteriseAllWithSkia)
            {
                double scale = this._filterOption.RasterisationResolution;

                Rectangle bounds = graphics.GetBounds();

                bounds = new Rectangle(bounds.Location.X - filter.TopLeftMargin.X, bounds.Location.Y - filter.TopLeftMargin.Y, bounds.Size.Width + filter.TopLeftMargin.X + filter.BottomRightMargin.X, bounds.Size.Height + filter.TopLeftMargin.Y + filter.BottomRightMargin.Y);

                if (bounds.Size.Width > 0 && bounds.Size.Height > 0)
                {
                    if (!this._filterOption.RasterisationResolutionRelative)
                    {
                        scale = scale / Math.Min(bounds.Size.Width, bounds.Size.Height);
                    }

```

```
                    RasterImage rasterised = SKRenderContextInterpreter.Rasterise(graphics, bounds, scale, true);
                    RasterImage filtered = null;

                    if (filter is IFilterWithRasterisableParameter filterWithRastParam)
                    {
                        filterWithRastParam.RasteriseParameter(SKRenderContextInterpreter.Rasterise, scale);
                    }

                    if (filter is ILocationInvariantFilter locInvFilter)
                    {
                        filtered = locInvFilter.Filter(rasterised, scale);
                    }
                    else if (filter is IFilterWithLocation filterWithLoc)
                    {
                        filtered = filterWithLoc.Filter(rasterised, bounds, scale);
                    }

                    if (filtered != null)
                    {
                        rasterised.Dispose();

                        DrawRasterImage(0, 0, filtered.Width, filtered.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, filtered);
                    }
                }
            }
            else if (this._filterOption.Operation == FilterOption.FilterOperations.RasteriseAllWithVectSharp)
            {
                double scale = this._filterOption.RasterisationResolution;

                Rectangle bounds = graphics.GetBounds();

                bounds = new Rectangle(bounds.Location.X - filter.TopLeftMargin.X, bounds.Location.Y - filter.TopLeftMargin.Y, bounds.Size.Width + filter.TopLeftMargin.X + filter.BottomRightMargin.X, bounds.Size.Height + filter.TopLeftMargin.Y + filter.BottomRightMargin.Y);

                if (bounds.Size.Width > 0 && bounds.Size.Height > 0)
                {
                    if (!this._filterOption.RasterisationResolutionRelative)
                    {
                        scale = scale / Math.Min(bounds.Size.Width, bounds.Size.Height);
                    }

                    if (graphics.TryRasterise(bounds, scale, true, out RasterImage rasterised))
                    {
                        RasterImage filtered = null;

                        if (filter is IFilterWithRasterisableParameter filterWithRastParam)
                        {
                            filterWithRastParam.RasteriseParameter(SKRenderContextInterpreter.Rasterise, scale);
                        }

                        if (filter is ILocationInvariantFilter locInvFilter)
                        {
                            filtered = locInvFilter.Filter(rasterised, scale);
                        }
                        else if (filter is IFilterWithLocation filterWithLoc)
                        {
                            filtered = filterWithLoc.Filter(rasterised, bounds, scale);
                        }

                        if (filtered != null)
                        {
                            rasterised.Dispose();

                            DrawRasterImage(0, 0, filtered.Width, filtered.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, filtered);
                        }
                    }
                    else
                    {
                        throw new NotImplementedException(@"The filter could not be rasterised! You can avoid this error by doing one of the following:
• Add a reference to VectSharp.Raster or VectSharp.Raster.ImageSharp (you may also need to add a using directive somewhere to force the assembly to be loaded).
• Provide your own implementation of Graphics.RasterisationMethod.
• Set the FilterOption.Operation to ""RasteriseAllWithSkia"", ""IgnoreAll"" or ""SkipAll"".");
                    }
                }
            }
            else if (this._filterOption.Operation == FilterOption.FilterOperations.IgnoreAll)
            {
                graphics.CopyToIGraphicsContext(this);
```

```
            }
            else
            {

            }
        }
    }

    internal class RenderCanvas : Avalonia.Controls.Canvas
    {
        private List<RenderAction> RenderActions;
        private List<RenderAction> TaggedRenderActions;

        private SolidColorBrush BackgroundBrush;

        static Avalonia.Point Origin = new Avalonia.Point(0, 0);

        public Dictionary<string, (IImage, bool)> Images;

        public void BringToFront(RenderAction action)
        {
            this.RenderActions.Remove(action);
            this.RenderActions.Add(action);

            if (!string.IsNullOrEmpty(action.Tag))
            {
                int index = this.TaggedRenderActions.IndexOf(action);
                if (index >= 0)
                {
                    this.TaggedRenderActions.RemoveAt(index);
                    this.TaggedRenderActions.Insert(0, action);
                }
            }
        }

        public void SendToBack(RenderAction action)
        {
            this.RenderActions.Remove(action);
            this.RenderActions.Insert(0, action);

            if (!string.IsNullOrEmpty(action.Tag))
            {
                int index = this.TaggedRenderActions.IndexOf(action);
                if (index >= 0)
                {
                    this.TaggedRenderActions.RemoveAt(index);
                    this.TaggedRenderActions.Add(action);
                }
            }
        }

        public RenderCanvas(Graphics content, Colour backgroundColour, double width, double height,
Dictionary<string, Delegate> taggedActions, bool removeTaggedActionsAfterExecution,
AvaloniaContextInterpreter.TextOptions textOption, FilterOption filterOption)
        {
            this.BackgroundBrush = new SolidColorBrush(Color.FromArgb((byte)(backgroundColour.A *
255), (byte)(backgroundColour.R * 255), (byte)(backgroundColour.G * 255), (byte)(backgroundColour.B *
255)));

            this.Width = width;
            this.Height = height;
            this.Images = new Dictionary<string, (IImage, bool)>();
            AvaloniaDrawingContext ctx = new AvaloniaDrawingContext(this.Width, this.Height,
removeTaggedActionsAfterExecution, textOption, this.Images, filterOption);
            foreach (KeyValuePair<string, Delegate> action in taggedActions)
            {
                ctx.TaggedActions.Add(action.Key, (Func<RenderAction,
IEnumerable<RenderAction»)action.Value);
            }

            content.CopyToIGraphicsContext(ctx);
            this.RenderActions = ctx.RenderActions;

            this.TaggedRenderActions = new List<RenderAction>();

            for (int i = this.RenderActions.Count - 1; i >= 0; i--)
            {
                RenderActions[i].InternalParent = this;
                if (!string.IsNullOrEmpty(this.RenderActions[i].Tag))
                {
                    TaggedRenderActions.Add(this.RenderActions[i]);
                }
            }

            this.PointerPressed += PointerPressedAction;
            this.PointerReleased += PointerReleasedAction;
            this.PointerMoved += PointerMoveAction;
```

```
            this.PointerLeave += PointerLeaveAction;
        }


        private int CurrentPressedAction = -1;
        private void PointerPressedAction(object sender, Avalonia.Input.PointerPressedEventArgs e)
        {
            Avalonia.Point position = e.GetPosition(this);

            for (int i = 0; i < TaggedRenderActions.Count; i++)
            {
                if (TaggedRenderActions[i].ClippingPath == null || TaggedRenderActions[i].ClippingPath.FillContains(position))
                {
                    Avalonia.Point localPosition = position.Transform(TaggedRenderActions[i].InverseTransform);

                    if (TaggedRenderActions[i].ActionType == RenderAction.ActionTypes.Path)
                    {
                        if ((TaggedRenderActions[i].Fill != null && TaggedRenderActions[i].Geometry.FillContains(localPosition)) || TaggedRenderActions[i].Geometry.StrokeContains(TaggedRenderActions[i].Stroke, localPosition))
                        {
                            TaggedRenderActions[i].FirePointerPressed(e);
                            CurrentPressedAction = i;
                            break;
                        }
                    }
                    else if (TaggedRenderActions[i].ActionType == RenderAction.ActionTypes.Text)
                    {
                        if (TaggedRenderActions[i].Fill != null && TaggedRenderActions[i].Text.HitTestPoint(localPosition).IsInside)
                        {
                            TaggedRenderActions[i].FirePointerPressed(e);
                            CurrentPressedAction = i;
                            break;
                        }
                    }
                    else if (TaggedRenderActions[i].ActionType == RenderAction.ActionTypes.RasterImage)
                    {
                        if (TaggedRenderActions[i].ImageDestination.Value.Contains(localPosition))
                        {
                            TaggedRenderActions[i].FirePointerPressed(e);
                            CurrentPressedAction = i;
                            break;
                        }
                    }
                }
            }
        }

        private void PointerReleasedAction(object sender, Avalonia.Input.PointerReleasedEventArgs e)
        {
            if (CurrentPressedAction >= 0)
            {
                TaggedRenderActions[CurrentPressedAction].FirePointerReleased(e);
                CurrentPressedAction = -1;
            }
            else
            {
                Avalonia.Point position = e.GetPosition(this);

                for (int i = 0; i < TaggedRenderActions.Count; i++)
                {
                    if (TaggedRenderActions[i].ClippingPath == null || TaggedRenderActions[i].ClippingPath.FillContains(position))
                    {
                        Avalonia.Point localPosition = position.Transform(TaggedRenderActions[i].InverseTransform);

                        if (TaggedRenderActions[i].ActionType == RenderAction.ActionTypes.Path)
                        {
                            if ((TaggedRenderActions[i].Fill != null && TaggedRenderActions[i].Geometry.FillContains(localPosition)) || TaggedRenderActions[i].Geometry.StrokeContains(TaggedRenderActions[i].Stroke, localPosition))
                            {
                                TaggedRenderActions[i].FirePointerReleased(e);
                                break;
                            }
                        }
                        else if (TaggedRenderActions[i].ActionType == RenderAction.ActionTypes.Text)
                        {
                            if (TaggedRenderActions[i].Fill != null && TaggedRenderActions[i].Text.HitTestPoint(localPosition).IsInside)
                            {
                                TaggedRenderActions[i].FirePointerReleased(e);
```

```
                                break;
                            }
                        }
                        else if (TaggedRenderActions[i].ActionType == RenderAction.ActionTypes.RasterImage)
                        {
                            if (TaggedRenderActions[i].ImageDestination.Value.Contains(localPosition))
                            {
                                TaggedRenderActions[i].FirePointerReleased(e);
                                break;
                            }
                        }
                    }
                }
            }
        }

        private int CurrentOverAction = -1;
        private void PointerMoveAction(object sender, Avalonia.Input.PointerEventArgs e)
        {
            Avalonia.Point position = e.GetPosition(this);

            bool found = false;

            for (int i = 0; i < TaggedRenderActions.Count; i++)
            {
                if (TaggedRenderActions[i].ClippingPath == null || TaggedRenderActions[i].ClippingPath.FillContains(position))
                {
                    Avalonia.Point localPosition = position.Transform(TaggedRenderActions[i].InverseTransform);

                    if (TaggedRenderActions[i].ActionType == RenderAction.ActionTypes.Path)
                    {
                        if ((TaggedRenderActions[i].Fill != null && TaggedRenderActions[i].Geometry.FillContains(localPosition)) || TaggedRenderActions[i].Geometry.StrokeContains(TaggedRenderActions[i].Stroke, localPosition))
                        {
                            found = true;

                            if (CurrentOverAction != i)
                            {
                                if (CurrentOverAction >= 0)
                                {
                                    TaggedRenderActions[CurrentOverAction].FirePointerLeave(e);
                                }
                                CurrentOverAction = i;
                                TaggedRenderActions[CurrentOverAction].FirePointerEnter(e);
                            }

                            break;
                        }
                    }
                    else if (TaggedRenderActions[i].ActionType == RenderAction.ActionTypes.Text)
                    {
                        if (TaggedRenderActions[i].Fill != null && TaggedRenderActions[i].Text.HitTestPoint(localPosition).IsInside)
                        {
                            found = true;

                            if (CurrentOverAction != i)
                            {
                                if (CurrentOverAction >= 0)
                                {
                                    TaggedRenderActions[CurrentOverAction].FirePointerLeave(e);
                                }
                                CurrentOverAction = i;
                                TaggedRenderActions[CurrentOverAction].FirePointerEnter(e);
                            }

                            break;
                        }
                    }
                    else if (TaggedRenderActions[i].ActionType == RenderAction.ActionTypes.RasterImage)
                    {
                        if (TaggedRenderActions[i].ImageDestination.Value.Contains(localPosition))
                        {
                            found = true;

                            if (CurrentOverAction != i)
                            {
                                if (CurrentOverAction >= 0)
                                {
                                    TaggedRenderActions[CurrentOverAction].FirePointerLeave(e);
                                }
                                CurrentOverAction = i;
```

```
                                TaggedRenderActions[CurrentOverAction].FirePointerEnter(e);
                            }

                            break;
                        }
                    }
                }
            }

            if (!found)
            {
                if (CurrentOverAction >= 0)
                {
                    TaggedRenderActions[CurrentOverAction].FirePointerLeave(e);
                }
                CurrentOverAction = -1;
            }
        }

        private void PointerLeaveAction(object sender, Avalonia.Input.PointerEventArgs e)
        {
            if (CurrentOverAction >= 0)
            {
                TaggedRenderActions[CurrentOverAction].FirePointerLeave(e);
            }
            CurrentOverAction = -1;
        }


        public override void Render(DrawingContext context)
        {
            context.FillRectangle(this.BackgroundBrush, new Avalonia.Rect(0, 0, Width, Height));

            foreach (RenderAction act in this.RenderActions)
            {
                if (act.ActionType == RenderAction.ActionTypes.Path)
                {
                    DrawingContext.PushedState? state = null;

                    if (act.ClippingPath != null)
                    {
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                        state = context.PushGeometryClip(act.ClippingPath);
                    }

                    using (context.PushPreTransform(act.Transform))
                    {
                        context.DrawGeometry(act.Fill, act.Stroke, act.Geometry);
                    }

                    if (state != null)
                    {
                        state?.Dispose();
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                    }
                }
                else if (act.ActionType == RenderAction.ActionTypes.Text)
                {
                    DrawingContext.PushedState? state = null;

                    if (act.ClippingPath != null)
                    {
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                        state = context.PushGeometryClip(act.ClippingPath);
                    }

                    using (context.PushPreTransform(act.Transform))
                    {
                        context.DrawText(act.Fill, Origin, act.Text);
                    }

                    if (state != null)
                    {
                        state?.Dispose();
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                    }
                }
                else if (act.ActionType == RenderAction.ActionTypes.RasterImage)
                {
```

```
                    DrawingContext.PushedState? state = null;

                    if (act.ClippingPath != null)
                    {
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                        state = context.PushGeometryClip(act.ClippingPath);
                    }

                    (IImage, bool) image = Images[act.ImageId];

                    using (context.PushPreTransform(act.Transform))
                    {
                        context.DrawImage(image.Item1, act.ImageSource.Value, act.ImageDestination.Value, image.Item2 ? Avalonia.Visuals.Media.Imaging.BitmapInterpolationMode.HighQuality : Avalonia.Visuals.Media.Imaging.BitmapInterpolationMode.Default);
                    }

                    if (state != null)
                    {
                        state?.Dispose();
                        //Random draw operation needed due to https://github.com/AvaloniaUI/Avalonia/issues/4408
                        context.DrawGeometry(null, new Pen(Brushes.Transparent), act.ClippingPath);
                    }
                }
            }

        }
    }


    /// <summary>
    /// Represents a light-weight rendering action.
    /// </summary>
    public class RenderAction
    {
        /// <summary>
        /// Types of rendering actions.
        /// </summary>
        public enum ActionTypes
        {
            /// <summary>
            /// The render action represents a path object.
            /// </summary>
            Path,

            /// <summary>
            /// The render action represents a text object.
            /// </summary>
            Text,

            /// <summary>
            /// The render action represents a raster image.
            /// </summary>
            RasterImage
        }

        /// <summary>
        /// Type of the rendering action.
        /// </summary>
        public ActionTypes ActionType { get; private set; }

        /// <summary>
        /// Geometry that needs to be rendered (null if the action type is <see cref="ActionTypes.Text"/>). If you change this, you need to invalidate the <see cref="Parent"/>'s visual.
        /// </summary>
        public Geometry Geometry { get; set; }

        /// <summary>
        /// Text that needs to be rendered (null if the action type is <see cref="ActionTypes.Path"/>). If you change this, you need to invalidate the <see cref="Parent"/>'s visual.
        /// </summary>
        public Avalonia.Media.FormattedText Text { get; set; }

        /// <summary>
        /// Rendering stroke (null if the action type is <see cref="ActionTypes.Text"/> or if the rendered action only has a <see cref="Fill"/>). If you change this, you need to invalidate the <see cref="Parent"/>'s visual.
        /// </summary>
        public Pen Stroke { get; set; }

        /// <summary>
```

```
        /// Rendering fill (null if the rendered action only has a <see cref="Stroke"/>). If you change this, you need to invalidate the <see cref="Parent"/>'s visual.
        /// </summary>
        public IBrush Fill { get; set; }

        /// <summary>
        /// Univocal identifier of the image that needs to be drawn.
        /// </summary>
        public string ImageId { get; set; }

        /// <summary>
        /// The source rectangle of the image.
        /// </summary>
        public Avalonia.Rect? ImageSource { get; set; }

        /// <summary>
        /// The destination rectangle of the image.
        /// </summary>
        public Avalonia.Rect? ImageDestination { get; set; }

        /// <summary>
        /// The current clipping path.
        /// </summary>
        public Geometry ClippingPath { get; set; }

        private Avalonia.Matrix _transform = Avalonia.Matrix.Identity;

        /// <summary>
        /// Inverse transformation matrix.
        /// </summary>
        public Avalonia.Matrix InverseTransform { get; private set; } = Avalonia.Matrix.Identity;

        /// <summary>
        /// Rendering transformation matrix. If you change this, you need to invalidate the <see cref="Parent"/>'s visual.
        /// </summary>
        public Avalonia.Matrix Transform
        {
            get { return _transform; }
            set
            {
                _transform = value;
                InverseTransform = _transform.Invert();
            }
        }

        /// <summary>
        /// A tag to access the <see cref="RenderAction"/>.
        /// </summary>
        public string Tag { get; set; }

        internal RenderCanvas InternalParent { get; set; }

        /// <summary>
        /// The container of this <see cref="RenderAction"/>.
        /// </summary>
        public Avalonia.Controls.Canvas Parent
        {
            get
            {
                return InternalParent;
            }
        }

        /// <summary>
        /// Raised when the pointer enters the area covered by the <see cref="RenderAction"/>.
        /// </summary>
        public event EventHandler<Avalonia.Input.PointerEventArgs> PointerEnter;

        /// <summary>
        /// Raised when the pointer leaves the area covered by the <see cref="RenderAction"/>.
        /// </summary>
        public event EventHandler<Avalonia.Input.PointerEventArgs> PointerLeave;

        /// <summary>
        /// Raised when the pointer is pressed while over the area covered by the <see cref="RenderAction"/>.
        /// </summary>
        public event EventHandler<Avalonia.Input.PointerPressedEventArgs> PointerPressed;

        /// <summary>
        /// Raised when the pointer is released after a <see cref="PointerPressed"/> event.
        /// </summary>
        public event EventHandler<Avalonia.Input.PointerReleasedEventArgs> PointerReleased;


        internal void FirePointerEnter(Avalonia.Input.PointerEventArgs e)
```

```
        {
            this.PointerEnter?.Invoke(this, e);
        }

        internal void FirePointerLeave(Avalonia.Input.PointerEventArgs e)
        {
            this.PointerLeave?.Invoke(this, e);
        }

        internal void FirePointerPressed(Avalonia.Input.PointerPressedEventArgs e)
        {
            this.PointerPressed?.Invoke(this, e);
        }

        internal void FirePointerReleased(Avalonia.Input.PointerReleasedEventArgs e)
        {
            this.PointerReleased?.Invoke(this, e);
        }

        private RenderAction()
        {

        }

        /// <summary>
        /// Creates a new <see cref="RenderAction"/> representing a path.
        /// </summary>
        /// <param name="geometry">The geometry to be rendered.</param>
        /// <param name="stroke">The stroke of the path (can be null).</param>
        /// <param name="fill">The fill of the path (can be null).</param>
        /// <param name="transform">The transform that will be applied to the path.</param>
        /// <param name="clippingPath">The clipping path.</param>
        /// <param name="tag">A tag to access the <see cref="RenderAction"/>. If this is null this
<see cref="RenderAction"/> is not visible in the hit test.</param>
        /// <returns>A new <see cref="RenderAction"/> representing a path.</returns>
        public static RenderAction PathAction(Geometry geometry, Pen stroke, IBrush fill,
Avalonia.Matrix transform, Geometry clippingPath, string tag = null)
        {
            return new RenderAction()
            {
                ActionType = ActionTypes.Path,
                Geometry = geometry,
                Stroke = stroke,
                Fill = fill,
                Transform = transform,
                ClippingPath = clippingPath,
                Tag = tag
            };
        }

        /// <summary>
        /// Creates a new <see cref="RenderAction"/> representing text.
        /// </summary>
        /// <param name="text">The text to be rendered.</param>
        /// <param name="fill">The fill of the text (can be null).</param>
        /// <param name="transform">The transform that will be applied to the text.</param>
        /// <param name="clippingPath">The clipping path.</param>
        /// <param name="tag">A tag to access the <see cref="RenderAction"/>. If this is null this
<see cref="RenderAction"/> is not visible in the hit test.</param>
        /// <returns>A new <see cref="RenderAction"/> representing text.</returns>
        public static RenderAction TextAction(Avalonia.Media.FormattedText text, IBrush fill,
Avalonia.Matrix transform, Geometry clippingPath, string tag = null)
        {
            return new RenderAction()
            {
                ActionType = ActionTypes.Text,
                Text = text,
                Stroke = null,
                Fill = fill,
                Transform = transform,
                ClippingPath = clippingPath,
                Tag = tag
            };
        }

        /// <summary>
        /// Creates a new <see cref="RenderAction"/> representing an image.
        /// </summary>
        /// <param name="imageId">The univocal identifier of the image to draw.</param>
        /// <param name="sourceRect">The source rectangle of the image.</param>
        /// <param name="destinationRect">The destination rectangle of the image.</param>
        /// <param name="transform">The transform that will be applied to the image.</param>
        /// <param name="clippingPath">The clipping path.</param>
        /// <param name="tag">A tag to access the <see cref="RenderAction"/>. If this is null this
<see cref="RenderAction"/> is not visible in the hit test.</param>
        /// <returns>A new <see cref="RenderAction"/> representing an image.</returns>
        public static RenderAction ImageAction(string imageId, Avalonia.Rect sourceRect, Avalonia.Rect
```

```
destinationRect, Avalonia.Matrix transform, Geometry clippingPath, string tag = null)
        {
            return new RenderAction()
            {
                ActionType = ActionTypes.RasterImage,
                ImageId = imageId,
                ImageSource = sourceRect,
                ImageDestination = destinationRect,
                Transform = transform,
                ClippingPath = clippingPath,
                Tag = tag
            };
        }

        /// <summary>
        /// Brings the render action to the front of the rendering queue. This method can only be invoked after the output has been fully initialised.
        /// </summary>
        public void BringToFront()
        {
            this.InternalParent.BringToFront(this);
        }

        /// <summary>
        /// Brings the render action to the back of the rendering queue. This method can only be invoked after the output has been fully initialised.
        /// </summary>
        public void SendToBack()
        {
            this.InternalParent.SendToBack(this);
        }
    }


    internal class AvaloniaDrawingContext : IGraphicsContext
    {
        public Dictionary<string, Func<RenderAction, IEnumerable<RenderAction>> TaggedActions { get; set; } = new Dictionary<string, Func<RenderAction, IEnumerable<RenderAction>>();

        private bool removeTaggedActions = true;

        public string Tag { get; set; }

        AvaloniaContextInterpreter.TextOptions _textOption;

        private Dictionary<string, (IImage, bool)> Images;

        private Geometry _clippingPath;
        private Stack<Geometry> clippingPaths;
        private FilterOption _filterOption;

        public AvaloniaDrawingContext(double width, double height, bool removeTaggedActionsAfterExecution, AvaloniaContextInterpreter.TextOptions textOption, Dictionary<string, (IImage, bool)> images, FilterOption filterOption)
        {
            this.Images = images;

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;

            RenderActions = new List<RenderAction>();
            removeTaggedActions = removeTaggedActionsAfterExecution;

            Width = width;
            Height = height;

            _transform = new double[3, 3];

            _transform[0, 0] = 1;
            _transform[1, 1] = 1;
            _transform[2, 2] = 1;

            states = new Stack<double[,]>();

            _textOption = textOption;

            _clippingPath = null;
            clippingPaths = new Stack<Geometry>();
            clippingPaths.Push(_clippingPath);

            _filterOption = filterOption;
        }

        public List<RenderAction> RenderActions { get; set; }

        public double Width { get; private set; }
```

```
        public double Height { get; private set; }

        public void Translate(double x, double y)
        {
            _transform = MatrixUtils.Translate(_transform, x, y);

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
        }

        public TextBaselines TextBaseline { get; set; }

        private void PathText(string text, double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            GraphicsPath textPath = new GraphicsPath().AddText(x, y, text, Font, TextBaseline);

            for (int j = 0; j < textPath.Segments.Count; j++)
            {
                switch (textPath.Segments[j].Type)
                {
                    case VectSharp.SegmentType.Move:
                        this.MoveTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.Line:
                        this.LineTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.CubicBezier:
                        this.CubicBezierTo(textPath.Segments[j].Points[0].X, textPath.Segments[j].Points[0].Y, textPath.Segments[j].Points[1].X, textPath.Segments[j].Points[1].Y, textPath.Segments[j].Points[2].X, textPath.Segments[j].Points[2].Y);
                        break;
                    case VectSharp.SegmentType.Close:
                        this.Close();
                        break;
                }
            }
        }

        public void StrokeText(string text, double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            PathText(text, x, y);
            Stroke();
        }

        public void FillText(string text, double x, double y)
        {
            if (!Font.EnableKerning)
            {
                FillSimpleText(text, x, y);
            }
            else
            {
                List<(string, Point)> tSpans = new List<(string, Point)>();

                System.Text.StringBuilder currentRun = new System.Text.StringBuilder();
                Point currentKerning = new Point();

                Point currentGlyphPlacementDelta = new Point();
                Point currentGlyphAdvanceDelta = new Point();
                Point nextGlyphPlacementDelta = new Point();
                Point nextGlyphAdvanceDelta = new Point();

                for (int i = 0; i < text.Length; i++)
                {
                    if (i < text.Length - 1)
                    {
                        currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                        currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                        nextGlyphAdvanceDelta = new Point();
                        nextGlyphPlacementDelta = new Point();

                        TrueTypeFile.PairKerning kerning = Font.FontFamily.TrueTypeFile.Get1000EmKerning(text[i], text[i + 1]);

                        if (kerning != null)
                        {
                            currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                            currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                            nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X +
```

```
kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                            nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                        }
                    }

                    if (currentGlyphPlacementDelta.X != 0 || currentGlyphPlacementDelta.Y != 0 || currentGlyphAdvanceDelta.X != 0 || currentGlyphAdvanceDelta.Y != 0)
                    {
                        if (currentRun.Length > 0)
                        {
                            tSpans.Add((currentRun.ToString(), currentKerning));

                            tSpans.Add((text[i].ToString(), new Point(currentGlyphPlacementDelta.X * Font.FontSize / 1000, currentGlyphPlacementDelta.Y * Font.FontSize / 1000)));

                            currentRun.Clear();
                            currentKerning = new Point((currentGlyphAdvanceDelta.X - currentGlyphPlacementDelta.X) * Font.FontSize / 1000, (currentGlyphAdvanceDelta.Y - currentGlyphPlacementDelta.Y) * Font.FontSize / 1000);
                        }
                        else
                        {
                            tSpans.Add((text[i].ToString(), new Point(currentGlyphPlacementDelta.X * Font.FontSize / 1000 + currentKerning.X, currentGlyphPlacementDelta.Y * Font.FontSize / 1000 + currentKerning.Y)));

                            currentRun.Clear();
                            currentKerning = new Point((currentGlyphAdvanceDelta.X - currentGlyphPlacementDelta.X) * Font.FontSize / 1000, (currentGlyphAdvanceDelta.Y - currentGlyphPlacementDelta.Y) * Font.FontSize / 1000);
                        }
                    }
                    else
                    {
                        currentRun.Append(text[i]);
                    }
                }

                if (currentRun.Length > 0)
                {
                    tSpans.Add((currentRun.ToString(), currentKerning));
                }

                double currX = x;
                double currY = y;

                Font.DetailedFontMetrics fullMetrics = Font.MeasureTextAdvanced(text);

                if (TextBaseline == TextBaselines.Top)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        currY += fullMetrics.Top;
                    }
                }
                else if (TextBaseline == TextBaselines.Middle)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        currY += (fullMetrics.Top + fullMetrics.Bottom) * 0.5;
                    }
                }
                else if (TextBaseline == TextBaselines.Bottom)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        currY += fullMetrics.Bottom;
                    }
                }

                TextBaseline = TextBaselines.Baseline;

                for (int i = 0; i < tSpans.Count; i++)
                {
                    Font.DetailedFontMetrics metrics = Font.MeasureTextAdvanced(tSpans[i].Item1);

                    if (i == 0)
                    {
                        FillSimpleText(tSpans[i].Item1, currX + tSpans[i].Item2.X, currY + tSpans[i].Item2.Y);
                    }
                    else
                    {
                        FillSimpleText(tSpans[i].Item1, currX + metrics.LeftSideBearing - fullMetrics.LeftSideBearing + tSpans[i].Item2.X, currY + tSpans[i].Item2.Y);
                    }
```

```
                    currX += metrics.AdvanceWidth + tSpans[i].Item2.X;
                    currY += tSpans[i].Item2.Y;
                }
            }
        }

        public void FillSimpleText(string text, double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if (_textOption == AvaloniaContextInterpreter.TextOptions.NeverConvert || (_textOption == AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary && Font.FontFamily.IsStandardFamily && Font.FontFamily.FileName != "ZapfDingbats" && Font.FontFamily.FileName != "Symbol"))
            {
                Avalonia.Media.FormattedText txt = new Avalonia.Media.FormattedText()
                {
                    Text = text,
                    Typeface = new Typeface(Avalonia.Media.FontFamily.Parse(FontFamily), (Font.FontFamily.IsOblique ? FontStyle.Oblique : Font.FontFamily.IsItalic ? FontStyle.Italic : FontStyle.Normal), (Font.FontFamily.IsBold ? FontWeight.Bold : FontWeight.Regular)),
                    FontSize = Font.FontSize
                };



                double top = y;
                double left = x;

                double[,] currTransform = null;
                double[,] deltaTransform = MatrixUtils.Identity;

                if (Font.FontFamily.TrueTypeFile != null)
                {
                    currTransform = MatrixUtils.Translate(_transform, x, y);
                }

                Font.DetailedFontMetrics metrics = Font.MeasureTextAdvanced(text);

                if (TextBaseline == TextBaselines.Top)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top + metrics.Top - Font.WinAscent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top + metrics.Top - Font.WinAscent);
                        }
                        else
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top + metrics.Top - Font.Ascent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top + metrics.Top - Font.Ascent);
                        }

                    }
                }
                else if (TextBaseline == TextBaselines.Middle)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top + metrics.Top / 2 + metrics.Bottom / 2 - Font.WinAscent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top + metrics.Top / 2 + metrics.Bottom / 2 - Font.WinAscent);
                        }
                        else
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top + metrics.Top / 2 + metrics.Bottom / 2 - Font.Ascent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top + metrics.Top / 2 + metrics.Bottom / 2 - Font.Ascent);
                        }
                    }
                }
                else if (TextBaseline == TextBaselines.Baseline)
                {
                    double lsb = Font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(text[0]).LeftSideBearing * Font.FontSize / 1000;

                    if (Font.FontFamily.TrueTypeFile != null)
```

```
                    {
                        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - lsb, top - Font.WinAscent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - lsb, top - Font.YMax);
                        }
                        else
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - lsb, top - Font.Ascent);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - lsb, top - Font.YMax);
                        }
                    }
                }
                else if (TextBaseline == TextBaselines.Bottom)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top - Font.WinAscent + metrics.Bottom);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top - Font.WinAscent + metrics.Bottom);
                        }
                        else
                        {
                            currTransform = MatrixUtils.Translate(_transform, left - metrics.LeftSideBearing, top - Font.Ascent + metrics.Bottom);
                            deltaTransform = MatrixUtils.Translate(deltaTransform, left - metrics.LeftSideBearing, top - Font.Ascent + metrics.Bottom);
                        }
                    }
                }

                Avalonia.Media.Brush fill = null;

                if (this.FillStyle is SolidColourBrush solid)
                {
                    fill = new SolidColorBrush(Color.FromArgb(FillAlpha, (byte)(solid.R * 255), (byte)(solid.G * 255), (byte)(solid.B * 255)));
                }
                else if (this.FillStyle is LinearGradientBrush linearGradient)
                {
                    fill = linearGradient.ToLinearGradientBrush(deltaTransform);
                }
                else if (this.FillStyle is RadialGradientBrush radialGradient)
                {
                    fill = radialGradient.ToRadialGradientBrush(metrics.Width + metrics.LeftSideBearing + metrics.RightSideBearing, deltaTransform);
                }

                RenderAction act = RenderAction.TextAction(txt, fill, currTransform.ToAvaloniaMatrix(), _clippingPath?.Clone(), Tag);

                if (!string.IsNullOrEmpty(Tag))
                {
                    if (TaggedActions.ContainsKey(Tag))
                    {
                        IEnumerable<RenderAction> actions = TaggedActions[Tag](act);

                        foreach (RenderAction action in actions)
                        {
                            RenderActions.Add(action);
                        }

                        if (removeTaggedActions)
                        {
                            TaggedActions.Remove(Tag);
                        }
                    }
                    else
                    {
                        RenderActions.Add(act);
                    }
                }
                else if (TaggedActions.ContainsKey(""))
                {
                    IEnumerable<RenderAction> actions = TaggedActions[""](act);

                    foreach (RenderAction action in actions)
                    {
                        RenderActions.Add(action);
                    }
```

```
                }
                else
                {
                    RenderActions.Add(act);
                }
            }
            else
            {
                PathText(text, x, y);
                Fill();
            }
        }

        public Brush StrokeStyle { get; private set; } = Colour.FromRgb(0, 0, 0);
        private byte StrokeAlpha = 255;

        public Brush FillStyle { get; private set; } = Colour.FromRgb(0, 0, 0);
        private byte FillAlpha = 255;

        public void SetFillStyle((int r, int g, int b, double a) style)
        {
            FillStyle = Colour.FromRgba(style.r, style.g, style.b, (int)(style.a * 255));
            FillAlpha = (byte)(style.a * 255);
        }

        public void SetFillStyle(Brush style)
        {
            FillStyle = style;

            if (style is SolidColourBrush solid)
            {
                FillAlpha = (byte)(solid.A * 255);
            }
            else
            {
                FillAlpha = 255;
            }
        }

        public void SetStrokeStyle((int r, int g, int b, double a) style)
        {
            StrokeStyle = Colour.FromRgba(style.r, style.g, style.b, (int)(style.a * 255));
            StrokeAlpha = (byte)(style.a * 255);
        }

        public void SetStrokeStyle(Brush style)
        {
            StrokeStyle = style;

            if (style is SolidColourBrush solid)
            {
                StrokeAlpha = (byte)(solid.A * 255);
            }
            else
            {
                StrokeAlpha = 255;
            }
        }

        private double[] LineDash;

        public void SetLineDash(LineDash dash)
        {
            LineDash = new double[] { dash.UnitsOn, dash.UnitsOff, dash.Phase };
        }

        public void Rotate(double angle)
        {
            Utils.CoerceNaNAndInfinityToZero(ref angle);

            _transform = MatrixUtils.Rotate(_transform, angle);

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
        }

        public void Transform(double a, double b, double c, double d, double e, double f)
        {
            Utils.CoerceNaNAndInfinityToZero(ref a, ref b, ref c, ref d, ref e, ref f);

            double[,] transfMatrix = new double[3, 3] { { a, c, e }, { b, d, f }, { 0, 0, 1 } };
            _transform = MatrixUtils.Multiply(_transform, transfMatrix);

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
```

```
        }

        public void Scale(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            _transform = MatrixUtils.Scale(_transform, x, y);

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
        }

        private double[,] _transform;

        private readonly Stack<double[,]> states;

        public void Save()
        {
            states.Push((double[,])_transform.Clone());
            clippingPaths.Push(_clippingPath?.Clone());
        }

        public void Restore()
        {
            _transform = states.Pop();
            _clippingPath = clippingPaths.Pop();
        }

        public double LineWidth { get; set; }
        public LineCaps LineCap { get; set; }
        public LineJoins LineJoin { get; set; }

        private string FontFamily;
        private Font _Font;

        public Font Font
        {
            get
            {
                return _Font;
            }

            set
            {
                _Font = value;

                if (!Font.FontFamily.IsStandardFamily)
                {
                    if (Font.FontFamily is ResourceFontFamily fam)
                    {
                        FontFamily = fam.ResourceName;
                    }
                    else
                    {
                        if (Font.FontFamily.TrueTypeFile != null)
                        {
                            FontFamily = Font.FontFamily.TrueTypeFile.GetFontFamilyName();
                        }
                        else
                        {
                            FontFamily = Font.FontFamily.FileName;
                        }
                    }
                }
                else
                {
                    FontFamily = "resm:VectSharp.StandardFonts.?assembly=VectSharp#" +
Font.FontFamily.TrueTypeFile.GetFontFamilyName();
                }
            }
        }

        public (double Width, double Height) MeasureText(string text)
        {
            Avalonia.Media.FormattedText txt = new Avalonia.Media.FormattedText() { Text = text,
Typeface = new Typeface(FontFamily), FontSize = Font.FontSize };
            return (txt.Bounds.Width, txt.Bounds.Height);
        }

        private PathGeometry currentPath;
        private PathFigure currentFigure;

        private bool figureInitialised = false;

        public void MoveTo(double x, double y)
        {
```

```
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if (figureInitialised)
            {
                currentPath.Figures.Add(currentFigure);
            }

            currentFigure = new PathFigure() { StartPoint = new Avalonia.Point(x, y), IsClosed = false };
            figureInitialised = true;
        }

        public void LineTo(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if (figureInitialised)
            {
                currentFigure.Segments.Add(new Avalonia.Media.LineSegment() { Point = new Avalonia.Point(x, y) });
            }
            else
            {
                currentFigure = new PathFigure() { StartPoint = new Avalonia.Point(x, y), IsClosed = false };
                figureInitialised = true;
            }
        }

        public void Rectangle(double x0, double y0, double width, double height)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x0, ref y0, ref width, ref height);

            if (currentFigure != null && figureInitialised)
            {
                currentPath.Figures.Add(currentFigure);
            }

            currentFigure = new PathFigure() { StartPoint = new Avalonia.Point(x0, y0), IsClosed = false };
            currentFigure.Segments.Add(new Avalonia.Media.LineSegment() { Point = new Avalonia.Point(x0 + width, y0) });
            currentFigure.Segments.Add(new Avalonia.Media.LineSegment() { Point = new Avalonia.Point(x0 + width, y0 + height) });
            currentFigure.Segments.Add(new Avalonia.Media.LineSegment() { Point = new Avalonia.Point(x0, y0 + height) });
            currentFigure.IsClosed = true;

            currentPath.Figures.Add(currentFigure);
            figureInitialised = false;
        }

        public void CubicBezierTo(double p1X, double p1Y, double p2X, double p2Y, double p3X, double p3Y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref p1X, ref p1Y, ref p2X, ref p2Y, ref p3X, ref p3Y);

            if (figureInitialised)
            {
                currentFigure.Segments.Add(new Avalonia.Media.BezierSegment() { Point1 = new Avalonia.Point(p1X, p1Y), Point2 = new Avalonia.Point(p2X, p2Y), Point3 = new Avalonia.Point(p3X, p3Y) });
            }
            else
            {
                currentFigure = new PathFigure() { StartPoint = new Avalonia.Point(p1X, p1Y), IsClosed = false };
                figureInitialised = true;
            }
        }

        public void Close()
        {
            currentFigure.IsClosed = true;
            currentPath.Figures.Add(currentFigure);
            figureInitialised = false;
        }

        public void Stroke()
        {
            if (figureInitialised)
            {
                currentFigure.IsClosed = false;
                currentPath.Figures.Add(currentFigure);
            }

            Avalonia.Media.Brush stroke = null;
```

```
            if (this.StrokeStyle is SolidColourBrush solid)
            {
                stroke = new SolidColorBrush(Color.FromArgb(StrokeAlpha, (byte)(solid.R * 255), (byte)(solid.G * 255), (byte)(solid.B * 255)));
            }
            else if (this.StrokeStyle is LinearGradientBrush linearGradient)
            {
                stroke = linearGradient.ToLinearGradientBrush();
            }
            else if (this.StrokeStyle is RadialGradientBrush radialGradient)
            {
                stroke = radialGradient.ToRadialGradientBrush(currentPath.Bounds.Width);
            }

            Pen pen = new Pen(stroke,
                    LineWidth,
                    new DashStyle(new double[] { (LineDash[0] + (LineCap == LineCaps.Butt ? 0 : LineWidth)) / LineWidth, (LineDash[1] - (LineCap == LineCaps.Butt ? 0 : LineWidth)) / LineWidth }, LineDash[2] / LineWidth));

            switch (LineCap)
            {
                case LineCaps.Butt:
                    pen.LineCap = PenLineCap.Flat;
                    break;
                case LineCaps.Round:
                    pen.LineCap = PenLineCap.Round;
                    break;
                case LineCaps.Square:
                    pen.LineCap = PenLineCap.Square;
                    break;
            }

            switch (LineJoin)
            {
                case LineJoins.Bevel:
                    pen.LineJoin = PenLineJoin.Bevel;
                    break;
                case LineJoins.Round:
                    pen.LineJoin = PenLineJoin.Round;
                    break;
                case LineJoins.Miter:
                    pen.LineJoin = PenLineJoin.Miter;
                    break;
            }

            RenderAction act = RenderAction.PathAction(currentPath, pen, null, _transform.ToAvaloniaMatrix(), _clippingPath?.Clone(), Tag);

            if (!string.IsNullOrEmpty(Tag))
            {
                if (TaggedActions.ContainsKey(Tag))
                {
                    IEnumerable<RenderAction> actions = TaggedActions[Tag](act);

                    foreach (RenderAction action in actions)
                    {
                        RenderActions.Add(action);
                    }

                    if (removeTaggedActions)
                    {
                        TaggedActions.Remove(Tag);
                    }
                }
                else
                {
                    RenderActions.Add(act);
                }
            }
            else if (TaggedActions.ContainsKey(""))
            {
                IEnumerable<RenderAction> actions = TaggedActions[""](act);

                foreach (RenderAction action in actions)
                {
                    RenderActions.Add(action);
                }
            }
            else
            {
                RenderActions.Add(act);
            }

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
```

```
            figureInitialised = false;
        }

        public void Fill()
        {
            if (figureInitialised)
            {
                currentPath.Figures.Add(currentFigure);
            }

            Avalonia.Media.Brush fill = null;

            if (this.FillStyle is SolidColourBrush solid)
            {
                fill = new SolidColorBrush(Color.FromArgb(FillAlpha, (byte)(solid.R * 255), (byte)(solid.G * 255), (byte)(solid.B * 255)));
            }
            else if (this.FillStyle is LinearGradientBrush linearGradient)
            {
                fill = linearGradient.ToLinearGradientBrush();
            }
            else if (this.FillStyle is RadialGradientBrush radialGradient)
            {
                fill = radialGradient.ToRadialGradientBrush(currentPath.Bounds.Width);
            }

            RenderAction act = RenderAction.PathAction(currentPath, null, fill, _transform.ToAvaloniaMatrix(), _clippingPath?.Clone(), Tag);

            if (!string.IsNullOrEmpty(Tag))
            {
                if (TaggedActions.ContainsKey(Tag))
                {
                    IEnumerable<RenderAction> actions = TaggedActions[Tag](act);

                    foreach (RenderAction action in actions)
                    {
                        RenderActions.Add(action);
                    }

                    if (removeTaggedActions)
                    {
                        TaggedActions.Remove(Tag);
                    }
                }
                else
                {
                    RenderActions.Add(act);
                }
            }
            else if (TaggedActions.ContainsKey(""))
            {
                IEnumerable<RenderAction> actions = TaggedActions[""](act);

                foreach (RenderAction action in actions)
                {
                    RenderActions.Add(action);
                }
            }
            else
            {
                RenderActions.Add(act);
            }

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
        }

        private static void TransformGeometry(PathGeometry geo, double[,] transf)
        {
            for (int i = 0; i < geo.Figures.Count; i++)
            {
                double[] tP = MatrixUtils.Multiply(transf, new double[] { geo.Figures[i].StartPoint.X, geo.Figures[i].StartPoint.Y });
                geo.Figures[i].StartPoint = new Avalonia.Point(tP[0], tP[1]);

                for (int j = 0; j < geo.Figures[i].Segments.Count; j++)
                {
                    if (geo.Figures[i].Segments[j] is LineSegment lS)
                    {
                        tP = MatrixUtils.Multiply(transf, new double[] { lS.Point.X, lS.Point.Y });
                        lS.Point = new Avalonia.Point(tP[0], tP[1]);
                    }
                    else if (geo.Figures[i].Segments[j] is BezierSegment bS)
                    {
                        tP = MatrixUtils.Multiply(transf, new double[] { bS.Point1.X, bS.Point1.Y });
```

```
                        double[] tP2 = MatrixUtils.Multiply(transf, new double[] { bS.Point2.X, bS.Point2.Y });
                        double[] tP3 = MatrixUtils.Multiply(transf, new double[] { bS.Point3.X, bS.Point3.Y });

                        bS.Point1 = new Avalonia.Point(tP[0], tP[1]);
                        bS.Point2 = new Avalonia.Point(tP2[0], tP2[1]);
                        bS.Point3 = new Avalonia.Point(tP3[0], tP3[1]);
                    }
                }
            }
        }

        public void SetClippingPath()
        {
            if (figureInitialised)
            {
                currentPath.Figures.Add(currentFigure);
            }

            TransformGeometry(currentPath, _transform);

            if (_clippingPath == null)
            {
                _clippingPath = currentPath;
            }
            else
            {
                //Can't find a better way of transforming an IStreamGeometryImpl into a Geometry...
                _clippingPath = (StreamGeometry)Activator.CreateInstance(typeof(StreamGeometry), System.Reflection.BindingFlags.NonPublic | System.Reflection.BindingFlags.Instance, null, new object[] { _clippingPath.PlatformImpl.Intersect(currentPath.PlatformImpl) }, null);
            }

            currentPath = new PathGeometry();
            currentFigure = new PathFigure() { IsClosed = false };
            figureInitialised = false;
        }

        public void DrawRasterImage(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image)
        {
            Utils.CoerceNaNAndInfinityToZero(ref destinationX, ref destinationY, ref destinationWidth, ref destinationHeight);

            if (!this.Images.ContainsKey(image.Id))
            {
                Bitmap bmp = new Bitmap(image.PNGStream);
                this.Images.Add(image.Id, (bmp, image.Interpolate));
            }

            RenderAction act = RenderAction.ImageAction(image.Id, new Avalonia.Rect(sourceX, sourceY, sourceWidth, sourceHeight), new Avalonia.Rect(destinationX, destinationY, destinationWidth, destinationHeight), _transform.ToAvaloniaMatrix(), _clippingPath?.Clone(), Tag);

            if (!string.IsNullOrEmpty(Tag))
            {
                if (TaggedActions.ContainsKey(Tag))
                {
                    IEnumerable<RenderAction> actions = TaggedActions[Tag](act);

                    foreach (RenderAction action in actions)
                    {
                        RenderActions.Add(action);
                    }

                    if (removeTaggedActions)
                    {
                        TaggedActions.Remove(Tag);
                    }
                }
                else
                {
                    RenderActions.Add(act);
                }
            }
            else if (TaggedActions.ContainsKey(""))
            {
                IEnumerable<RenderAction> actions = TaggedActions[""](act);

                foreach (RenderAction action in actions)
                {
                    RenderActions.Add(action);
                }
            }
            else
```

```
            {
                RenderActions.Add(act);
            }
        }

        public void DrawFilteredGraphics(Graphics graphics, IFilter filter)
        {
            if (this._filterOption.Operation == FilterOption.FilterOperations.RasteriseAllWithSkia)
            {
                double scale = this._filterOption.RasterisationResolution;

                Rectangle bounds = graphics.GetBounds();

                bounds = new Rectangle(bounds.Location.X - filter.TopLeftMargin.X, bounds.Location.Y - filter.TopLeftMargin.Y, bounds.Size.Width + filter.TopLeftMargin.X + filter.BottomRightMargin.X, bounds.Size.Height + filter.TopLeftMargin.Y + filter.BottomRightMargin.Y);

                if (bounds.Size.Width > 0 && bounds.Size.Height > 0)
                {
                    if (!this._filterOption.RasterisationResolutionRelative)
                    {
                        scale = scale / Math.Min(bounds.Size.Width, bounds.Size.Height);
                    }

                    RasterImage rasterised = SKRenderContextInterpreter.Rasterise(graphics, bounds, scale, true);
                    RasterImage filtered = null;

                    if (filter is IFilterWithRasterisableParameter filterWithRastParam)
                    {
                        filterWithRastParam.RasteriseParameter(SKRenderContextInterpreter.Rasterise, scale);
                    }

                    if (filter is ILocationInvariantFilter locInvFilter)
                    {
                        filtered = locInvFilter.Filter(rasterised, scale);
                    }
                    else if (filter is IFilterWithLocation filterWithLoc)
                    {
                        filtered = filterWithLoc.Filter(rasterised, bounds, scale);
                    }

                    if (filtered != null)
                    {
                        rasterised.Dispose();

                        DrawRasterImage(0, 0, filtered.Width, filtered.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, filtered);
                    }
                }
            }
            else if (this._filterOption.Operation == FilterOption.FilterOperations.RasteriseAllWithVectSharp)
            {
                double scale = this._filterOption.RasterisationResolution;

                Rectangle bounds = graphics.GetBounds();

                bounds = new Rectangle(bounds.Location.X - filter.TopLeftMargin.X, bounds.Location.Y - filter.TopLeftMargin.Y, bounds.Size.Width + filter.TopLeftMargin.X + filter.BottomRightMargin.X, bounds.Size.Height + filter.TopLeftMargin.Y + filter.BottomRightMargin.Y);

                if (bounds.Size.Width > 0 && bounds.Size.Height > 0)
                {
                    if (!this._filterOption.RasterisationResolutionRelative)
                    {
                        scale = scale / Math.Min(bounds.Size.Width, bounds.Size.Height);
                    }

                    if (graphics.TryRasterise(bounds, scale, true, out RasterImage rasterised))
                    {
                        RasterImage filtered = null;

                        if (filter is IFilterWithRasterisableParameter filterWithRastParam)
                        {
                            filterWithRastParam.RasteriseParameter(SKRenderContextInterpreter.Rasterise, scale);
                        }

                        if (filter is ILocationInvariantFilter locInvFilter)
                        {
                            filtered = locInvFilter.Filter(rasterised, scale);
                        }
                        else if (filter is IFilterWithLocation filterWithLoc)
                        {
                            filtered = filterWithLoc.Filter(rasterised, bounds, scale);
```

```
                        }

                        if (filtered != null)
                        {
                            rasterised.Dispose();

                            DrawRasterImage(0, 0, filtered.Width, filtered.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, filtered);
                        }
                    }
                    else
                    {
                        throw new NotImplementedException(@"The filter could not be rasterised! You can avoid this error by doing one of the following:
• Add a reference to VectSharp.Raster or VectSharp.Raster.ImageSharp (you may also need to add a using directive somewhere to force the assembly to be loaded).
• Provide your own implementation of Graphics.RasterisationMethod.
• Set the FilterOption.Operation to ""RasteriseAllWithSkia"", ""IgnoreAll"" or ""SkipAll"".");
                    }
                }
            }
            else if (this._filterOption.Operation == FilterOption.FilterOperations.IgnoreAll)
            {
                graphics.CopyToIGraphicsContext(this);
            }
            else
            {

            }
        }
    }


    /// <summary>
    /// Contains methods to render a <see cref="Page"/> to an <see cref="Avalonia.Controls.Canvas"/>.
    /// </summary>
    public static class AvaloniaContextInterpreter
    {
        /// <summary>
        /// Defines whether text items should be converted into paths when drawing.
        /// </summary>
        public enum TextOptions
        {
            /// <summary>
            /// Converts all text items into paths.
            /// </summary>
            AlwaysConvert,

            /// <summary>
            /// Converts all text items into paths, with the exception of those that use a standard font.
            /// </summary>
            ConvertIfNecessary,

            /// <summary>
            /// Does not convert any text items into paths.
            /// </summary>
            NeverConvert
        }

        /// <summary>
        /// Render a <see cref="Page"/> to an <see cref="Avalonia.Controls.Canvas"/>.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered graphics objects.</returns>
        public static Avalonia.Controls.Canvas PaintToCanvas(this Page page, TextOptions textOption = TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            if (filterOption == null)
            {
                filterOption = FilterOption.Default;
            }

            AvaloniaContext ctx = new AvaloniaContext(page.Width, page.Height, true, textOption, filterOption);
            page.Graphics.CopyToIGraphicsContext(ctx);
            ctx.ControlItem.Background = new SolidColorBrush(Color.FromArgb((byte)(page.Background.A * 255), (byte)(page.Background.R * 255), (byte)(page.Background.G * 255), (byte)(page.Background.B * 255)));
            return ctx.ControlItem;
        }

```

```
        /// <summary>
        /// Render a <see cref="Page"/> to an <see cref="Avalonia.Controls.Canvas"/>.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="graphicsAsControls">If this is true, each graphics object (e.g. paths, text...) is rendered as a separate <see cref="Avalonia.Controls.Control"/>. Otherwise, they are directly rendered onto the drawing context (which is faster, but does not allow interactivity).</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered graphics objects.</returns>
        public static Avalonia.Controls.Canvas PaintToCanvas(this Page page, bool graphicsAsControls, TextOptions textOption = TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            if (graphicsAsControls)
            {
                Avalonia.Controls.Canvas tbr = page.PaintToCanvas(textOption, filterOption);
                tbr.Background = new SolidColorBrush(Color.FromArgb((byte)(page.Background.A * 255), (byte)(page.Background.R * 255), (byte)(page.Background.G * 255), (byte)(page.Background.B * 255)));
                return tbr;
            }
            else
            {
                return new RenderCanvas(page.Graphics, page.Background, page.Width, page.Height, new Dictionary<string, Delegate>(), true, textOption, filterOption) { Background = new SolidColorBrush(Color.FromArgb((byte)(page.Background.A * 255), (byte)(page.Background.R * 255), (byte)(page.Background.G * 255), (byte)(page.Background.B * 255))) };
            }
        }

        /// <summary>
        /// Render a <see cref="Page"/> to an <see cref="Avalonia.Controls.Canvas"/>.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="graphicsAsControls">If this is true, each graphics object (e.g. paths, text...) is rendered as a separate <see cref="Avalonia.Controls.Control"/>. Otherwise, they are directly rendered onto the drawing context (which is faster, but does not allow interactivity).</param>
        /// <param name="taggedActions">A <see cref="Dictionary{String, Delegate}"/> containing the <see cref="Action"/>s that will be performed on items with the corresponding tag.
        /// If <paramref name="graphicsAsControls"/> is true, the delegates should be voids that accept one parameter of type <see cref="TextBlock"/> or <see cref="Path"/> (depending on the tagged item), otherwise, they should accept one parameter of type <see cref="RenderAction"/> and return an <see cref="IEnumerable{RenderAction}"/> of the actions that will actually be performed.</param>
        /// <param name="removeTaggedActionsAfterExecution">Whether the <see cref="Action"/>s should be removed from <paramref name="taggedActions"/> after their execution. Set to false if the same <see cref="Action"/> should be performed on multiple items with the same tag.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered graphics objects.</returns>
        public static Avalonia.Controls.Canvas PaintToCanvas(this Page page, bool graphicsAsControls, Dictionary<string, Delegate> taggedActions, bool removeTaggedActionsAfterExecution = true, TextOptions textOption = TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            if (graphicsAsControls)
            {
                Avalonia.Controls.Canvas tbr = page.PaintToCanvas(taggedActions, removeTaggedActionsAfterExecution, textOption, filterOption);
                tbr.Background = new SolidColorBrush(Color.FromArgb((byte)(page.Background.A * 255), (byte)(page.Background.R * 255), (byte)(page.Background.G * 255), (byte)(page.Background.B * 255)));
                return tbr;
            }
            else
            {
                return new RenderCanvas(page.Graphics, page.Background, page.Width, page.Height, taggedActions, removeTaggedActionsAfterExecution, textOption, filterOption) { Background = new SolidColorBrush(Color.FromArgb((byte)(page.Background.A * 255), (byte)(page.Background.R * 255), (byte)(page.Background.G * 255), (byte)(page.Background.B * 255))) }; ;
            }
        }

        /// <summary>
        /// Render a <see cref="Page"/> to an <see cref="Avalonia.Controls.Canvas"/>.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="taggedActions">A <see cref="Dictionary{String, Delegate}"/> containing the <see cref="Action"/>s that will be performed on items with the corresponding tag.
        /// The delegates should accept one parameter of type <see cref="TextBlock"/> or <see cref="Path"/> (depending on the tagged item).</param>
        /// <param name="removeTaggedActionsAfterExecution">Whether the <see cref="Action"/>s should be removed from <paramref name="taggedActions"/> after their execution. Set to false if the same <see
```

```
cref="Action"/> should be performed on multiple items with the same tag.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered graphics objects.</returns>
        public static Avalonia.Controls.Canvas PaintToCanvas(this Page page, Dictionary<string, Delegate> taggedActions, bool removeTaggedActionsAfterExecution = true, TextOptions textOption = TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            if (filterOption == null)
            {
                filterOption = FilterOption.Default;
            }

            AvaloniaContext ctx = new AvaloniaContext(page.Width, page.Height, removeTaggedActionsAfterExecution, textOption, filterOption)
            {
                TaggedActions = taggedActions
            };
            page.Graphics.CopyToIGraphicsContext(ctx);
            ctx.ControlItem.Background = new SolidColorBrush(Color.FromArgb((byte)(page.Background.A * 255), (byte)(page.Background.R * 255), (byte)(page.Background.G * 255), (byte)(page.Background.B * 255)));
            return ctx.ControlItem;
        }

        /// <summary>
        /// Render an <see cref="Animation"/> to an <see cref="Avalonia.Controls.Canvas"/>.
        /// </summary>
        /// <param name="animation">The <see cref="Animation"/> to render.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second (fps).</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered animation.</returns>
        public static AnimatedCanvas PaintToAnimatedCanvas(this Animation animation, double frameRate = 60, double durationScaling = 1, TextOptions textOption = TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            if (filterOption == null)
            {
                filterOption = FilterOption.Default;
            }

            return new AnimatedCanvas(animation, durationScaling, frameRate, textOption, filterOption) { Background = new SolidColorBrush(Color.FromArgb((byte)(animation.Background.A * 255), (byte)(animation.Background.R * 255), (byte)(animation.Background.G * 255), (byte)(animation.Background.B * 255))) };
        }

        internal static Avalonia.Media.LinearGradientBrush ToLinearGradientBrush(this LinearGradientBrush brush, double[,] transformMatrix = null)
        {
            Point start = brush.StartPoint;
            Point end = brush.EndPoint;

            if (transformMatrix != null)
            {
                double[,] inverse = MatrixUtils.Invert(transformMatrix);

                double[] startVec = MatrixUtils.Multiply(inverse, new double[] { start.X, start.Y });
                double[] endVec = MatrixUtils.Multiply(inverse, new double[] { end.X, end.Y });

                start = new Point(startVec[0], startVec[1]);
                end = new Point(endVec[0], endVec[1]);
            }

            Avalonia.Media.LinearGradientBrush tbr = new Avalonia.Media.LinearGradientBrush()
            {
                SpreadMethod = GradientSpreadMethod.Pad,
                StartPoint = new Avalonia.RelativePoint(start.X, start.Y, Avalonia.RelativeUnit.Absolute),
                EndPoint = new Avalonia.RelativePoint(end.X, end.Y, Avalonia.RelativeUnit.Absolute)
            };

            Avalonia.Media.GradientStops stops = new Avalonia.Media.GradientStops();
            stops.AddRange(from el in brush.GradientStops select new Avalonia.Media.GradientStop(Color.FromArgb((byte)(el.Colour.A * 255), (byte)(el.Colour.R * 255), (byte)(el.Colour.G * 255), (byte)(el.Colour.B * 255)), el.Offset));
```

```

            tbr.GradientStops = stops;

            return tbr;
        }

        internal static Avalonia.Media.RadialGradientBrush ToRadialGradientBrush(this RadialGradientBrush brush, double objectWidth, double[,] transformMatrix = null)
        {
            Point focus = brush.FocalPoint;
            Point centre = brush.Centre;

            if (transformMatrix != null)
            {
                double[,] inverse = MatrixUtils.Invert(transformMatrix);

                double[] focusVec = MatrixUtils.Multiply(inverse, new double[] { focus.X, focus.Y });
                double[] centreVec = MatrixUtils.Multiply(inverse, new double[] { centre.X, centre.Y });

                focus = new Point(focusVec[0], focusVec[1]);
                centre = new Point(centreVec[0], centreVec[1]);
            }

            Avalonia.Media.RadialGradientBrush tbr = new Avalonia.Media.RadialGradientBrush()
            {
                SpreadMethod = GradientSpreadMethod.Pad,
                Center = new Avalonia.RelativePoint(centre.X, centre.Y, Avalonia.RelativeUnit.Absolute),
                GradientOrigin = new Avalonia.RelativePoint(focus.X, focus.Y, Avalonia.RelativeUnit.Absolute),
                Radius = brush.Radius / objectWidth
            };

            Avalonia.Media.GradientStops stops = new Avalonia.Media.GradientStops();
            stops.AddRange(from el in brush.GradientStops select new Avalonia.Media.GradientStop(Color.FromArgb((byte)(el.Colour.A * 255), (byte)(el.Colour.R * 255), (byte)(el.Colour.G * 255), (byte)(el.Colour.B * 255)), el.Offset));
            tbr.GradientStops = stops;

            return tbr;
        }
    }
}
```

8.3 RenderingParameters.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;

namespace VectSharp.Canvas
{
    internal class RenderingParameters : IEquatable<RenderingParameters>
    {
        public static float Tolerance = 1e-5f;
        public float Left { get; }
        public float Top { get; }
        public float Width { get; }
        public float Height { get; }
        public float Scale { get; }
        public int RenderWidth { get; }
        public int RenderHeight { get; }

        public RenderingParameters(float left, float top, float width, float height, float scale, int renderWidth, int renderHeight)
        {
            this.Left = left;
```

```
            this.Top = top;
            this.Width = width;
            this.Height = height;
            this.Scale = scale;
            this.RenderWidth = renderWidth;
            this.RenderHeight = renderHeight;
        }

        public RenderingParameters Clone()
        {
            return new RenderingParameters(this.Left, this.Top, this.Width, this.Height, this.Scale, this.RenderWidth, this.RenderHeight);
        }

        public bool Equals(RenderingParameters other)
        {
            if (!object.ReferenceEquals(other, null))
            {
                return this.Scale == other.Scale && this.Left == other.Left && this.Top == other.Top && this.Width == other.Width && this.Height == other.Height && this.RenderWidth == other.RenderWidth && this.RenderHeight == other.RenderHeight;
            }
            else
            {
                return false;
            }
        }

        public override bool Equals(object obj)
        {
            if (obj is RenderingParameters other)
            {
                return this.Equals(other);
            }
            else
            {
                return false;
            }
        }

        public override int GetHashCode()
        {
            int hash = 13;

            unchecked
            {
                hash = (hash * 7) + this.Left.GetHashCode();
                hash = (hash * 7) + this.Top.GetHashCode();
                hash = (hash * 7) + this.Width.GetHashCode();
                hash = (hash * 7) + this.Height.GetHashCode();
                hash = (hash * 7) + this.Scale.GetHashCode();
                hash = (hash * 7) + this.RenderWidth.GetHashCode();
                hash = (hash * 7) + this.RenderHeight.GetHashCode();
            }

            return hash;
        }

        public static bool operator ==(RenderingParameters param1, RenderingParameters param2)
        {
            if (object.ReferenceEquals(param1, null))
            {
                return object.ReferenceEquals(param2, null);
            }
            else
            {
                return param1.Equals(param2);
            }
        }

        public static bool operator !=(RenderingParameters param1, RenderingParameters param2)
        {
            if (!object.ReferenceEquals(param1, null) && !object.ReferenceEquals(param2, null))
            {
                return param1.Scale != param2.Scale || param1.Left != param2.Left || param1.Top != param2.Top || param1.Width != param2.Width || param1.Height != param2.Height || param1.RenderWidth != param2.RenderWidth || param1.RenderHeight != param2.RenderHeight;
            }
            else
            {
                return !(object.ReferenceEquals(param1, null) && object.ReferenceEquals(param2, null));
            }
        }

        public bool GoodEnough(RenderingParameters other)
        {
```

```
            if (this.Scale == other.Scale)
            {
                return (this.Left <= other.Left && (this.Left + this.Width - other.Left - other.Width) / (this.Left + this.Width + other.Left + other.Width) >= -Tolerance && this.Top <= other.Top && (this.Top + this.Height - other.Top - other.Height) / (this.Top + this.Height + other.Top + other.Height) >= -Tolerance);
            }
            else
            {
                return false;
            }
        }
    }
}
```

8.4 SKMultiLayerRenderCanvas.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using Avalonia;
using Avalonia.Media;
using Avalonia.Media.Imaging;
using Avalonia.Platform;
using Avalonia.Rendering.SceneGraph;
using Avalonia.Skia;
using SkiaSharp;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;
using System.Reflection;
using System.Threading;

namespace VectSharp.Canvas
{
    internal interface ISKRenderCanvas
    {
        void InvalidateDirty();
        void InvalidateZIndex();
    }

    /// <summary>
    /// Represents a multi-threaded, triple-buffered canvas on which the image is drawn using SkiaSharp.
    /// </summary>
    public class SKMultiLayerRenderCanvas : Avalonia.Controls.Canvas, IDisposable, ISKRenderCanvas
    {
        /// <summary>
        /// The width of the page that is rendered on this canvas.
        /// </summary>
        public double PageWidth { get; set; }

        /// <summary>
        /// The height of the page that is rendered on this canvas.
        /// </summary>
        public double PageHeight { get; set; }
        private SKColor BackgroundColour;

        /// <summary>
        /// The list of render actions, each element in this list is itself a list, containing the actions that correspond to a layer in the image.
        /// </summary>
        public List<List<SKRenderAction>> RenderActions;

        /// <summary>
        /// The list of transforms associated with each layer.
        /// </summary>
        public List<SKRenderAction> LayerTransforms;
```

```
        /// <summary>
        /// If the image to draw is not already cached, this method is called with an argument containing the number of milliseconds since the image was last rendered.
        /// The method can return a Bitmap that will be drawn on the canvas in order to let users know that the image is being rendered in background.
        /// </summary>
        public Func<long, Bitmap> Spinner { get; set; } = null;

        private readonly System.Diagnostics.Stopwatch LastFrameStopwatch = new System.Diagnostics.Stopwatch();
        static readonly Func<IDrawingContextImpl, IVisualNode> GetNode;

        static SKMultiLayerRenderCanvas()
        {
            Type DeferredDrawingContextImpl = Assembly.GetAssembly(typeof(Avalonia.Rendering.SceneGraph.Scene)).GetType("Avalonia.Rendering.SceneGraph.DeferredDrawi

            FieldInfo NodeFieldInfo = DeferredDrawingContextImpl.GetField("_node", BindingFlags.NonPublic | BindingFlags.Instance);

            ParameterExpression ownerParameter = Expression.Parameter(typeof(IDrawingContextImpl));

            MemberExpression exp = Expression.Field(Expression.Convert(ownerParameter, DeferredDrawingContextImpl), NodeFieldInfo);
            GetNode = Expression.Lambda<Func<IDrawingContextImpl, IVisualNode»(exp, ownerParameter).Compile();
        }

        private Dictionary<string, (SKBitmap, bool)> Images;
        private List<List<SKRenderAction» TaggedRenderActions;

        /// <summary>
        /// Create a new SKMultiLayerRenderCanvas from a <see cref="Document"/>, where each page represents a layer.
        /// </summary>
        /// <param name="document">The document containing the layers as <see cref="Page"/>s.</param>
        /// <param name="layerTransforms">A list of transforms associated with each layer. This list should contain the same number of elements as the number of pages in <paramref name="document"/>. This is useful to manipulate the position of each layer individually. If this is null, an identity transform is applied to each layer.</param>
        /// <param name="backgroundColour">The background colour of the canvas.</param>
        /// <param name="width">The width of the canvas and the pages it contains.</param>
        /// <param name="height">The height of the canvas and the pages it contains.</param>
        public SKMultiLayerRenderCanvas(Document document, Colour backgroundColour, double width, double height, List<SKRenderAction> layerTransforms = null) : this(document.Pages, backgroundColour, width, height, layerTransforms) { }

        /// <summary>
        /// Create a new SKMultiLayerRenderCanvas from a collection of <see cref="Page"/>s, each representing a layer.
        /// </summary>
        /// <param name="layers">The contents of the canvas. Each element in this list represents a layer.</param>
        /// <param name="layerTransforms">A list of transforms associated with each layer. This list should contain the same number of elements as <paramref name="layers"/>. This is useful to manipulate the position of each layer individually. If this is null, an identity transform is applied to each layer.</param>
        /// <param name="backgroundColour">The background colour of the canvas.</param>
        /// <param name="width">The width of the canvas and the pages it contains.</param>
        /// <param name="height">The height of the canvas and the pages it contains.</param>
        public SKMultiLayerRenderCanvas(IEnumerable<Page> layers, Colour backgroundColour, double width, double height, List<SKRenderAction> layerTransforms = null)
        {
            List<SKRenderContext> contents = (from el in layers select el.CopyToSKRenderContext()).ToList();
            List<SKRenderAction> contentTransforms;

            if (layerTransforms== null)
            {
                contentTransforms = (from el in Enumerable.Range(0, contents.Count) select SKRenderAction.TransformAction(SKMatrix.Identity)).ToList();
            }
            else
            {
                contentTransforms = layerTransforms;
            }

            UpdateWith(contents, contentTransforms, backgroundColour, width, height);

            this.Width = width;
            this.Height = height;

            this.PointerPressed += this.PointerPressedAction;
            this.PointerReleased += this.PointerReleasedAction;
            this.PointerMoved += this.PointerMoveAction;
            this.PointerLeave += this.PointerLeaveAction;
```

```
            LastFrameStopwatch.Start();
        }

        /// <summary>
        /// Create a new SKMultiLayerRenderCanvas from a list of SKRenderContexts, each representing a layer.
        /// </summary>
        /// <param name="contents">The contents of the canvas. Each element in this list represents a layer. A Page can be converded to a SKRenderContext through the CopyToSKRenderContext method.</param>
        /// <param name="contentTransforms">A list of transforms associated with each layer. This list should contain the same number of elements as <paramref name="contents"/>. This is useful to manipulate the position of each layer individually.</param>
        /// <param name="backgroundColour">The background colour of the canvas.</param>
        /// <param name="width">The width of the canvas and the page it contains.</param>
        /// <param name="height">The height of the canvas and the page it contains.</param>
        public SKMultiLayerRenderCanvas(List<SKRenderContext> contents, List<SKRenderAction> contentTransforms, Colour backgroundColour, double width, double height)
        {
            UpdateWith(contents, contentTransforms, backgroundColour, width, height);

            this.Width = width;
            this.Height = height;

            this.PointerPressed += this.PointerPressedAction;
            this.PointerReleased += this.PointerReleasedAction;
            this.PointerMoved += this.PointerMoveAction;
            this.PointerLeave += this.PointerLeaveAction;

            LastFrameStopwatch.Start();
        }

        /// <summary>
        /// Replace the contents of the SKMultiLayerRenderCanvas with the specified layers.
        /// </summary>
        /// <param name="contents">The contents of the canvas. Each element in this list represents a layer. A Page can be converded to a SKRenderContext through the CopyToSKRenderContext method.</param>
        /// <param name="contentTransforms">A list of transforms associated with each layer. This list should contain the same number of elements as <paramref name="contents"/>. This is useful to manipulate the position of each layer individually.</param>
        /// <param name="backgroundColour">The background colour of the canvas.</param>
        /// <param name="width">The width of the canvas and the page it contains.</param>
        /// <param name="height">The height of the canvas and the page it contains.</param>
        public void UpdateWith(List<SKRenderContext> contents, List<SKRenderAction> contentTransforms, Colour backgroundColour, double width, double height)
        {
            lock (RenderLock)
            {
                if (this.RenderActions == null)
                {
                    this.RenderActions = new List<List<SKRenderAction»();
                }
                else
                {
                    this.RenderActions.Clear();
                }

                if (this.LayerTransforms == null)
                {
                    this.LayerTransforms = new List<SKRenderAction>();
                }
                else
                {
                    this.LayerTransforms.Clear();
                }

                this.PageWidth = width;
                this.PageHeight = height;
                this.BackgroundColour = backgroundColour.ToSKColor();

                if (this.Images == null)
                {
                    Images = new Dictionary<string, (SKBitmap, bool)>();
                }
                else
                {
                    Images.Clear();
                }

                for (int i = 0; i < contents.Count; i++)
                {
                    this.LayerTransforms.Add(contentTransforms[i]);
                    this.RenderActions.Add(contents[i].SKRenderActions);

                    foreach (KeyValuePair<string, (SKBitmap, bool)> kvp in contents[i].Images)
                    {
                        this.Images[kvp.Key] = kvp.Value;
```

```
                    }
                }

                if (this.TaggedRenderActions == null)
                {
                    this.TaggedRenderActions = new List<List<SKRenderAction»();
                }
                else
                {
                    this.TaggedRenderActions.Clear();
                }


                for (int i = this.RenderActions.Count - 1; i >= 0; i--)
                {
                    TaggedRenderActions.Add(new List<SKRenderAction>());

                    for (int j = this.RenderActions[i].Count - 1; j >= 0; j--)
                    {
                        RenderActions[i][j].InternalParent = this;
                        if (!string.IsNullOrEmpty(this.RenderActions[i][j].Tag))
                        {
                            TaggedRenderActions[this.RenderActions.Count - 1 - i].Add(this.RenderActions[i][j]);
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Replace a single layer with the specified content.
        /// </summary>
        /// <param name="layer">The index of the layer to replace.</param>
        /// <param name="newContent">The new contents of the layer. A Page can be converded to a SKRenderContext through the CopyToSKRenderContext method.</param>
        /// <param name="newTransform">The new transform for the layer.</param>
        public void UpdateLayer(int layer, SKRenderContext newContent, SKRenderAction newTransform)
        {
            lock (RenderLock)
            {
                this.LayerTransforms[layer] = newTransform;
                this.RenderActions[layer] = newContent.SKRenderActions;

                foreach (KeyValuePair<string, (SKBitmap, bool)> kvp in newContent.Images)
                {
                    this.Images[kvp.Key] = kvp.Value;
                }

                TaggedRenderActions[this.RenderActions.Count - 1 - layer].Clear();

                for (int j = this.RenderActions[layer].Count - 1; j >= 0; j--)
                {
                    RenderActions[layer][j].InternalParent = this;
                    if (!string.IsNullOrEmpty(this.RenderActions[layer][j].Tag))
                    {
                        TaggedRenderActions[this.RenderActions.Count - 1 - layer].Add(this.RenderActions[layer][j]);
                    }
                }
            }

            this.InvalidateDirty();
        }

        /// <summary>
        /// Add a new layer to the image.
        /// </summary>
        /// <param name="newContent">The contents of the new layer. A Page can be converded to a SKRenderContext through the CopyToSKRenderContext method.</param>
        /// <param name="newTransform">The transform for the new layer.</param>
        public void AddLayer(SKRenderContext newContent, SKRenderAction newTransform)
        {
            lock (RenderLock)
            {
                this.LayerTransforms.Add(newTransform);
                this.RenderActions.Add(newContent.SKRenderActions);

                foreach (KeyValuePair<string, (SKBitmap, bool)> kvp in newContent.Images)
                {
                    this.Images[kvp.Key] = kvp.Value;
                }

                TaggedRenderActions.Insert(0, new List<SKRenderAction>());

                for (int j = this.RenderActions[this.RenderActions.Count - 1].Count - 1; j >= 0; j--)
                {
```

```
                    RenderActions[this.RenderActions.Count - 1][j].InternalParent = this;
                    if (!string.IsNullOrEmpty(this.RenderActions[this.RenderActions.Count - 1][j].Tag))
                    {
                        TaggedRenderActions[0].Add(this.RenderActions[this.RenderActions.Count - 1][j]);
                    }
                }
            }

            this.InvalidateDirty();
        }

        /// <summary>
        /// Insert a new layer at the specified index.
        /// </summary>
        /// <param name="index">The position at which the new layer will be inserted.</param>
        /// <param name="newContent">The contents of the new layer.</param>
        /// <param name="newTransform">The transform for the new layer.</param>
        public void InsertLayer(int index, SKRenderContext newContent, SKRenderAction newTransform)
        {
            lock (RenderLock)
            {
                this.LayerTransforms.Insert(index, newTransform);
                this.RenderActions.Insert(index, newContent.SKRenderActions);

                foreach (KeyValuePair<string, (SKBitmap, bool)> kvp in newContent.Images)
                {
                    this.Images[kvp.Key] = kvp.Value;
                }

                TaggedRenderActions.Insert(this.RenderActions.Count - 1 - index, new List<SKRenderAction>());

                for (int j = this.RenderActions[index].Count - 1; j >= 0; j--)
                {
                    RenderActions[index][j].InternalParent = this;
                    if (!string.IsNullOrEmpty(this.RenderActions[index][j].Tag))
                    {
                        TaggedRenderActions[this.RenderActions.Count - 1 - index].Add(this.RenderActions[index][j]);
                    }
                }
            }

            this.InvalidateDirty();
        }

        /// <summary>
        /// Remove the specified layer from the image.
        /// </summary>
        /// <param name="layer">The index of the layer to remove.</param>
        public void RemoveLayer(int layer)
        {
            lock (RenderLock)
            {
                this.LayerTransforms[layer].Dispose();
                this.LayerTransforms.RemoveAt(layer);

                for (int i = 0; i < this.RenderActions[layer].Count; i++)
                {
                    this.RenderActions[layer][i].Dispose();
                }
                this.RenderActions.RemoveAt(layer);

                TaggedRenderActions.RemoveAt(this.TaggedRenderActions.Count - 1 - layer);
            }

            this.InvalidateDirty();
        }

        /// <summary>
        /// Switch the position of the two specified layers.
        /// </summary>
        /// <param name="layer1">The index of the first layer to switch.</param>
        /// <param name="layer2">The index of the second layer to switch.</param>
        public void SwitchLayers(int layer1, int layer2)
        {
            lock (RenderLock)
            {
                var temp = this.LayerTransforms[layer1];
                this.LayerTransforms[layer1] = this.LayerTransforms[layer2];
                this.LayerTransforms[layer2] = temp;

                var temp2 = this.RenderActions[layer1];
                this.RenderActions[layer1] = this.RenderActions[layer2];
                this.RenderActions[layer2] = temp2;
```

```
                var temp3 = this.TaggedRenderActions[this.RenderActions.Count - 1 - layer1];
                this.TaggedRenderActions[this.RenderActions.Count - 1 - layer1] = this.TaggedRenderActions[this.RenderActions.Count - 1 - layer2];
                this.TaggedRenderActions[this.RenderActions.Count - 1 - layer2] = temp3;
            }

            this.InvalidateDirty();
        }

        /// <summary>
        /// Move the specified layer to the specified position, shifting all other layers as necessary.
        /// </summary>
        /// <param name="oldIndex">The current index of the layer to move.</param>
        /// <param name="newIndex">The final index of the layer. Layers after this will be shifted by 1 in order to accommodate the moved layer.</param>
        public void MoveLayer(int oldIndex, int newIndex)
        {
            lock (RenderLock)
            {
                var temp = this.LayerTransforms[oldIndex];
                this.LayerTransforms.RemoveAt(oldIndex);
                this.LayerTransforms.Insert(newIndex, temp);

                var temp2 = this.RenderActions[oldIndex];
                this.RenderActions.RemoveAt(oldIndex);
                this.RenderActions.Insert(newIndex, temp2);

                var temp3 = this.TaggedRenderActions[this.RenderActions.Count - 1 - oldIndex];
                this.TaggedRenderActions.RemoveAt(this.RenderActions.Count - 1 - oldIndex);
                this.TaggedRenderActions.Insert(this.RenderActions.Count - 1 - newIndex, temp3);
            }

            this.InvalidateDirty();
        }


        private SKRenderAction CurrentPressedAction = null;
        private void PointerPressedAction(object sender, Avalonia.Input.PointerPressedEventArgs e)
        {
            SKPoint position = e.GetPosition(this).ToSKPoint();

            for (int i = 0; i < TaggedRenderActions.Count; i++)
            {
                bool found = false;

                for (int j = 0; j < TaggedRenderActions[i].Count; j++)
                {
                    if (TaggedRenderActions[i][j].LastRenderedGlobalHitTestPath?.Contains(position.X, position.Y) == true)
                    {

                        if (TaggedRenderActions[i][j].ActionType == SKRenderAction.ActionTypes.Path)
                        {
                            CurrentPressedAction = TaggedRenderActions[i][j];
                            TaggedRenderActions[i][j].FirePointerPressed(e);
                            found = true;
                            break;
                        }
                        else if (TaggedRenderActions[i][j].ActionType == SKRenderAction.ActionTypes.Text)
                        {
                            CurrentPressedAction = TaggedRenderActions[i][j];
                            TaggedRenderActions[i][j].FirePointerPressed(e);
                            found = true;
                            break;
                        }
                        else if (TaggedRenderActions[i][j].ActionType == SKRenderAction.ActionTypes.RasterImage)
                        {
                            CurrentPressedAction = TaggedRenderActions[i][j];
                            TaggedRenderActions[i][j].FirePointerPressed(e);
                            found = true;
                            break;
                        }
                    }
                }

                if (found)
                {
                    break;
                }
            }
        }

        private void PointerReleasedAction(object sender, Avalonia.Input.PointerReleasedEventArgs e)
```

```
        {
            if (CurrentPressedAction != null)
            {
                if (!CurrentPressedAction.Disposed)
                {
                    CurrentPressedAction.FirePointerReleased(e);
                }
                CurrentPressedAction = null;
            }
            else
            {
                SKPoint position = e.GetPosition(this).ToSKPoint();

                for (int i = 0; i < TaggedRenderActions.Count; i++)
                {
                    bool found = false;

                    for (int j = 0; j < TaggedRenderActions[i].Count; j++)
                    {
                        if (TaggedRenderActions[i][j].LastRenderedGlobalHitTestPath?.Contains(position.X, position.Y) == true)
                        {
                            if (TaggedRenderActions[i][j].ActionType == SKRenderAction.ActionTypes.Path)
                            {
                                TaggedRenderActions[i][j].FirePointerReleased(e);
                                found = true;
                                break;
                            }
                            else if (TaggedRenderActions[i][j].ActionType == SKRenderAction.ActionTypes.Text)
                            {
                                TaggedRenderActions[i][j].FirePointerReleased(e);
                                found = true;
                                break;
                            }
                            else if (TaggedRenderActions[i][j].ActionType == SKRenderAction.ActionTypes.RasterImage)
                            {
                                TaggedRenderActions[i][j].FirePointerReleased(e);
                                found = true;
                                break;
                            }
                        }
                    }

                    if (found)
                    {
                        break;
                    }
                }
            }
        }

        private SKRenderAction CurrentOverAction = null;
        private void PointerMoveAction(object sender, Avalonia.Input.PointerEventArgs e)
        {
            SKPoint position = e.GetPosition(this).ToSKPoint();

            bool found = false;

            for (int i = 0; i < TaggedRenderActions.Count; i++)
            {
                for (int j = 0; j < TaggedRenderActions[i].Count; j++)
                {
                    if (TaggedRenderActions[i][j].LastRenderedGlobalHitTestPath?.Contains(position.X, position.Y) == true)
                    {
                        if (TaggedRenderActions[i][j].ActionType == SKRenderAction.ActionTypes.Path)
                        {
                            found = true;

                            if (CurrentOverAction != TaggedRenderActions[i][j])
                            {
                                if (CurrentOverAction != null && !CurrentOverAction.Disposed)
                                {
                                    CurrentOverAction.FirePointerLeave(e);
                                }
                                CurrentOverAction = TaggedRenderActions[i][j];
                                CurrentOverAction.FirePointerEnter(e);
                            }

                            break;
                        }
                        else if (TaggedRenderActions[i][j].ActionType == SKRenderAction.ActionTypes.Text)
                        {
```

```
                                found = true;

                                if (CurrentOverAction != TaggedRenderActions[i][j])
                                {
                                    if (CurrentOverAction != null && !CurrentOverAction.Disposed)
                                    {
                                        CurrentOverAction.FirePointerLeave(e);
                                    }
                                    CurrentOverAction = TaggedRenderActions[i][j];
                                    CurrentOverAction.FirePointerEnter(e);
                                }

                                break;
                            }
                            else if (TaggedRenderActions[i][j].ActionType ==
        SKRenderAction.ActionTypes.RasterImage)
                            {
                                found = true;

                                if (CurrentOverAction != TaggedRenderActions[i][j])
                                {
                                    if (CurrentOverAction != null && !CurrentOverAction.Disposed)
                                    {
                                        CurrentOverAction.FirePointerLeave(e);
                                    }
                                    CurrentOverAction = TaggedRenderActions[i][j];
                                    CurrentOverAction.FirePointerEnter(e);
                                }

                                break;
                            }
                        }
                    }

                    if (found)
                    {
                        break;
                    }
                }

                if (!found)
                {
                    if (CurrentOverAction != null && !CurrentOverAction.Disposed)
                    {
                        CurrentOverAction.FirePointerLeave(e);
                    }
                    CurrentOverAction = null;
                }
            }

            private void PointerLeaveAction(object sender, Avalonia.Input.PointerEventArgs e)
            {
                if (CurrentOverAction != null && !CurrentOverAction.Disposed)
                {
                    CurrentOverAction.FirePointerLeave(e);
                }
                CurrentOverAction = null;
            }

            /// <inheritdoc/>
            protected override void OnAttachedToVisualTree(VisualTreeAttachmentEventArgs e)
            {
                base.OnAttachedToVisualTree(e);

                StartRenderingThread();
            }

            /// <inheritdoc/>
            protected override void OnDetachedFromVisualTree(VisualTreeAttachmentEventArgs e)
            {
                StopRenderingThread();
            }

            private void StopRenderingThread()
            {
                DisposedHandle.Set();
            }

            private EventWaitHandle DisposedHandle;
            private EventWaitHandle RenderRequestedHandle;

            private RenderingParameters RenderingRequest = null;
            private readonly object RenderingRequestLock = new object();

            private void StartRenderingThread()
            {
                DisposedHandle = new EventWaitHandle(false, EventResetMode.ManualReset);
```

```
            RenderRequestedHandle = new EventWaitHandle(false, EventResetMode.ManualReset);

            Thread renderingThread = new Thread(async () =>
            {
                bool finished = false;
                WaitHandle[] handles = new WaitHandle[] { DisposedHandle, RenderRequestedHandle };

                while (!finished)
                {
                    int handle = EventWaitHandle.WaitAny(handles);

                    if (handle == 0)
                    {
                        finished = true;
                    }
                    else if (handle == 1)
                    {
                        RenderingParameters requestParams;

                        lock (RenderingRequestLock)
                        {
                            requestParams = RenderingRequest.Clone();
                            RenderRequestedHandle.Reset();
                        }

                        SKCanvas canvas;

                        if (BackBuffer == null || BackBufferRenderingParams == null || requestParams.RenderWidth > BackBufferRenderingParams.RenderWidth || requestParams.RenderHeight > BackBufferRenderingParams.RenderHeight)
                        {
                            await Avalonia.Threading.Dispatcher.UIThread.InvokeAsync(() =>
                            {
                                SKCanvas tempCanvasReference = BackBufferSkiaCanvas;
                                ISkiaDrawingContextImpl tempContextReference = BackBufferSkiaContext;
                                RenderTargetBitmap tempBufferReference = BackBuffer;

                                _ = Avalonia.Threading.Dispatcher.UIThread.InvokeAsync(() =>
                                {
                                    tempCanvasReference?.Dispose();
                                    tempContextReference?.Dispose();
                                    tempBufferReference?.Dispose();
                                }, Avalonia.Threading.DispatcherPriority.MinValue);


                                BackBuffer = new RenderTargetBitmap(new PixelSize(requestParams.RenderWidth, requestParams.RenderHeight), new Vector(96, 96));
                                BackBufferSkiaContext = BackBuffer.CreateDrawingContext(null) as ISkiaDrawingContextImpl;
                                BackBufferSkiaCanvas = BackBufferSkiaContext.SkCanvas;
                            }, Avalonia.Threading.DispatcherPriority.MaxValue);
                        }

                        canvas = BackBufferSkiaCanvas;

                        canvas.Save();

                        canvas.Scale(requestParams.Scale);
                        canvas.Translate(-requestParams.Left, -requestParams.Top);

                        canvas.Clear(BackgroundColour);

                        canvas.ClipRect(new SKRect(requestParams.Left, requestParams.Top, requestParams.Left + requestParams.Width, requestParams.Top + requestParams.Height));

                        RenderImage(canvas);

                        canvas.RestoreToCount(-1);

                        BackBufferRenderingParams = requestParams;

                        lock (FrontBufferLock)
                        {
                            RenderTargetBitmap tempFrontReference = FrontBuffer;
                            RenderingParameters tempFrontRenderingParameters = FrontBufferRenderingParams;
                            SKCanvas tempFrontCanvas = FrontBufferSkiaCanvas;
                            ISkiaDrawingContextImpl tempFrontContext = FrontBufferSkiaContext;

                            FrontBuffer = BackBuffer;
                            FrontBufferRenderingParams = BackBufferRenderingParams;
                            FrontBufferSkiaCanvas = BackBufferSkiaCanvas;
                            FrontBufferSkiaContext = BackBufferSkiaContext;

                            BackBuffer = BackBuffer2;
                            BackBufferRenderingParams = BackBufferRenderingParams2;
                            BackBufferSkiaCanvas = BackBufferSkiaCanvas2;
```

```
                            BackBufferSkiaContext = BackBufferSkiaContext2;

                            BackBuffer2 = tempFrontReference;
                            BackBufferRenderingParams2 = tempFrontRenderingParameters;
                            BackBufferSkiaCanvas2 = tempFrontCanvas;
                            BackBufferSkiaContext2 = tempFrontContext;
                        }

                        await Avalonia.Threading.Dispatcher.UIThread.InvokeAsync(() =>
                        {
                            base.InvalidateVisual();
                        }, Avalonia.Threading.DispatcherPriority.MaxValue);
                    }
                }
            });

            renderingThread.Start();
        }

        /// <summary>
        /// An lock for the rendering loop. The public methods of this class already lock on this, but you may need it if you want to directly manipulate the contents of the canvas.
        /// </summary>
        public object RenderLock = new object();

        /// <summary>
        /// Render the image at to a bitmap at the specified resolution.
        /// </summary>
        /// <param name="width">The width of the rendered image. Note that the actual width of the returned image might be lower than this, depending on the aspect ratio of the image.</param>
        /// <param name="height">The height of the rendered image. Note that the actual height of the returned image might be lower than this, depending on the aspect ratio of the image.</param>
        /// <param name="background">The background colour for the image. If this is <see langword="null" />, the current background colour is used.</param>
        /// <returns>A <see cref="RenderTargetBitmap"/> containing the image rendered at the specified resolution.</returns>
        public RenderTargetBitmap RenderAtResolution(int width, int height, SKColor? background = null)
        {
            SKColor realBackground = background ?? this.BackgroundColour;

            double scale = Math.Min(width / this.PageWidth, height / this.PageHeight);

            width = (int)Math.Round(this.PageWidth * scale);
            height = (int)Math.Round(this.PageHeight * scale);

            RenderTargetBitmap tbr = new RenderTargetBitmap(new PixelSize(width, height), new Vector(96, 96));
            ISkiaDrawingContextImpl context = tbr.CreateDrawingContext(null) as ISkiaDrawingContextImpl;
            SKCanvas canvas = context.SkCanvas;

            canvas.Clear(realBackground);

            canvas.Save();

            canvas.Scale((float)scale);

            RenderImage(canvas);

            canvas.RestoreToCount(-1);

            canvas.Dispose();
            context.Dispose();

            return tbr;
        }

        private void RenderImage(SKCanvas canvas)
        {
            lock (RenderLock)
            {
                for (int i = 0; i < this.RenderActions.Count; i++)
                {
                    canvas.Save();

                    if (this.LayerTransforms[i].ActionType == SKRenderAction.ActionTypes.Transform)
                    {
                        SKMatrix mat = this.LayerTransforms[i].Transform.Value;
                        canvas.Concat(ref mat);
                    }

                    RenderLayer(canvas, i);
                    canvas.Restore();
                }
            }

```

```
            /*for (int i = 0; i < this.RenderActions.Count; i++)
            {
                SKPaint selectionPaint = new SKPaint() { Color = new SKColor(255, 0, 255, 128) };

                for (int j = 0; j < this.TaggedRenderActions[i].Count; j++)
                {
                    if (this.TaggedRenderActions[i][j].LastRenderedGlobalHitTestPath != null)
                    {
                        canvas.DrawPath(this.TaggedRenderActions[i][j].LastRenderedGlobalHitTestPath, selectionPaint);
                    }
                }
            }*/
        }

        private void RenderLayer(SKCanvas canvas, int layer)
        {
            bool updateHitTests = this.TaggedRenderActions[this.TaggedRenderActions.Count - 1 - layer].Count > 0;

            HashSet<uint> ZIndices = new HashSet<uint>();

            canvas.Save();

            SKMatrix invertedInitialTransform;

            if (this.LayerTransforms[layer].ActionType == SKRenderAction.ActionTypes.Transform)
            {
                invertedInitialTransform = (canvas.TotalMatrix.PreConcat(this.LayerTransforms[layer].Transform.Value.Invert())).Invert();
            }
            else
            {
                invertedInitialTransform = canvas.TotalMatrix.Invert();
            }

            SKPath clipPath = null;
            Stack<SKPath> clipPaths = new Stack<SKPath>();
            clipPaths.Push(null);

            for (int i = 0; i < this.RenderActions[layer].Count; i++)
            {
                ZIndices.Add(this.RenderActions[layer][i].ZIndex);

                if (this.RenderActions[layer][i].ActionType == SKRenderAction.ActionTypes.Clip)
                {
                    canvas.ClipPath(this.RenderActions[layer][i].Path, antialias: true);

                    if (updateHitTests)
                    {
                        if (clipPath == null)
                        {
                            clipPath = this.RenderActions[layer][i].Path.Clone();
                            clipPath.Transform(canvas.TotalMatrix.PostConcat(invertedInitialTransform));
                        }
                        else
                        {
                            using (SKPath tempPath = this.RenderActions[layer][i].Path.Clone())
                            {
                                tempPath.Transform(canvas.TotalMatrix.PostConcat(invertedInitialTransform));
                                clipPath.Op(tempPath, SKPathOp.Intersect);
                            }
                        }
                    }
                }
                else if (this.RenderActions[layer][i].ActionType == SKRenderAction.ActionTypes.Restore)
                {
                    canvas.Restore();

                    if (updateHitTests)
                    {
                        clipPath = clipPaths.Pop();
                    }
                }
                else if (this.RenderActions[layer][i].ActionType == SKRenderAction.ActionTypes.Save)
                {
                    canvas.Save();

                    if (updateHitTests)
                    {
                        clipPaths.Push(clipPath?.Clone());
                    }
                }
                else if (this.RenderActions[layer][i].ActionType ==
```

```
SKRenderAction.ActionTypes.Transform)
                {
                    SKMatrix mat = this.RenderActions[layer][i].Transform.Value;
                    canvas.Concat(ref mat);
                }
                else
                {
                    if (this.RenderActions[layer][i].ZIndex == 0)
                    {

                        if (this.RenderActions[layer][i].ActionType == SKRenderAction.ActionTypes.Path
&& this.RenderActions[layer][i].Path != null)
                        {
                            canvas.DrawPath(this.RenderActions[layer][i].Path,
this.RenderActions[layer][i].Paint);
                        }
                        else if (this.RenderActions[layer][i].ActionType ==
SKRenderAction.ActionTypes.Text)
                        {
                            canvas.DrawText(this.RenderActions[layer][i].Text,
this.RenderActions[layer][i].TextX, this.RenderActions[layer][i].TextY,
this.RenderActions[layer][i].Font, this.RenderActions[layer][i].Paint);
                        }
                        else if (this.RenderActions[layer][i].ActionType ==
SKRenderAction.ActionTypes.RasterImage)
                        {
                            (SKBitmap image, bool interpolate) =
this.Images[this.RenderActions[layer][i].ImageId];

                            SKPaint paint;

                            if (!interpolate)
                            {
                                paint = null;
                            }
                            else
                            {
                                paint = new SKPaint() { FilterQuality = SKFilterQuality.Medium };
                            }

                            canvas.DrawBitmap(image, this.RenderActions[layer][i].ImageSource.Value,
this.RenderActions[layer][i].ImageDestination.Value, paint);

                            paint?.Dispose();
                        }
                    }

                    if (updateHitTests && this.RenderActions[layer][i].Tag != null &&
this.RenderActions[layer][i].HitTestPath != null)
                    {
                        SKPath hitTestPath = this.RenderActions[layer][i].HitTestPath.Clone();

hitTestPath.Transform(canvas.TotalMatrix.PostConcat(invertedInitialTransform));

                        if (clipPath != null)
                        {
                            hitTestPath.Op(clipPath, SKPathOp.Intersect);
                        }

                        this.RenderActions[layer][i].LastRenderedGlobalHitTestPath = hitTestPath;
                    }
                }
            }

            canvas.Restore();

            uint[] sortedIndices = ZIndices.OrderBy(x => x).ToArray();

            if (sortedIndices.Length > 1 || sortedIndices[0] != 0)
            {
                for (int j = 0; j < sortedIndices.Length; j++)
                {
                    canvas.Save();

                    for (int i = 0; i < this.RenderActions[layer].Count; i++)
                    {
                        if (this.RenderActions[layer][i].ActionType ==
SKRenderAction.ActionTypes.Clip)
                        {
                            canvas.ClipPath(this.RenderActions[layer][i].Path, antialias: true);
                        }
                        else if (this.RenderActions[layer][i].ActionType ==
SKRenderAction.ActionTypes.Restore)
                        {
                            canvas.Restore();
                        }
                        else if (this.RenderActions[layer][i].ActionType ==
```

```
SKRenderAction.ActionTypes.Save)
                        {
                            canvas.Save();
                        }
                        else if (this.RenderActions[layer][i].ActionType ==
SKRenderAction.ActionTypes.Transform)
                        {
                            SKMatrix mat = this.RenderActions[layer][i].Transform.Value;
                            canvas.Concat(ref mat);
                        }
                        else if (sortedIndices[j] != 0 && this.RenderActions[layer][i].ZIndex ==
sortedIndices[j])
                        {
                            if (this.RenderActions[layer][i].ActionType ==
SKRenderAction.ActionTypes.Path && this.RenderActions[layer][i].Path != null)
                            {
                                canvas.DrawPath(this.RenderActions[layer][i].Path,
this.RenderActions[layer][i].Paint);
                            }
                            else if (this.RenderActions[layer][i].ActionType ==
SKRenderAction.ActionTypes.Text)
                            {
                                canvas.DrawText(this.RenderActions[layer][i].Text,
this.RenderActions[layer][i].TextX, this.RenderActions[layer][i].TextY,
this.RenderActions[layer][i].Font, this.RenderActions[layer][i].Paint);
                            }
                            else if (this.RenderActions[layer][i].ActionType ==
SKRenderAction.ActionTypes.RasterImage)
                            {
                                (SKBitmap image, bool interpolate) =
this.Images[this.RenderActions[layer][i].ImageId];

                                SKPaint paint;

                                if (!interpolate)
                                {
                                    paint = null;
                                }
                                else
                                {
                                    paint = new SKPaint() { FilterQuality = SKFilterQuality.Medium };
                                }

                                canvas.DrawBitmap(image,
this.RenderActions[layer][i].ImageSource.Value, this.RenderActions[layer][i].ImageDestination.Value,
paint);

                                paint?.Dispose();
                            }
                        }
                    }

                    canvas.Restore();
                }
            }
        }

        private bool IsDirty = false;

        /// <summary>
        /// Invalidate the contents of the canvas, forcing it to redraw itself.
        /// </summary>
        public void InvalidateDirty()
        {
            this.IsDirty = true;
            base.InvalidateVisual();
        }

        /// <summary>
        /// Invalidate the contents of the canvas, specifying that the order of the layers has
changed.
        /// </summary>
        public void InvalidateZIndex()
        {
            for (int i = 0; i < this.TaggedRenderActions.Count; i++)
            {
                this.TaggedRenderActions[i] = this.TaggedRenderActions[i].OrderByDescending(a =>
a.ZIndex).ToList();
            }
            this.InvalidateDirty();
        }


        private RenderTargetBitmap FrontBuffer = null;
        private RenderingParameters FrontBufferRenderingParams = null;
        private readonly object FrontBufferLock = new object();
        ISkiaDrawingContextImpl FrontBufferSkiaContext = null;
```

```
        SKCanvas FrontBufferSkiaCanvas = null;

        private RenderTargetBitmap BackBuffer = null;
        private RenderingParameters BackBufferRenderingParams = null;
        ISkiaDrawingContextImpl BackBufferSkiaContext = null;
        SKCanvas BackBufferSkiaCanvas = null;

        private RenderTargetBitmap BackBuffer2 = null;
        private RenderingParameters BackBufferRenderingParams2 = null;
        ISkiaDrawingContextImpl BackBufferSkiaContext2 = null;
        SKCanvas BackBufferSkiaCanvas2 = null;

        /// <inheritdoc/>
        public override void Render(DrawingContext context)
        {
            lock (FrontBufferLock)
            {
                IVisualNode node = GetNode(context.PlatformImpl);

                Rect layoutBounds = node.LayoutBounds;
                Rect clipBounds = node.ClipBounds;


                double DPIscaling = (VisualRoot as Avalonia.Layout.ILayoutRoot)?.LayoutScaling ?? 1;

                double scale = 0.5 * (this.PageWidth / layoutBounds.Width + this.PageHeight / layoutBounds.Height);


                double left = Math.Max(0, (clipBounds.Left - layoutBounds.Left) * scale);
                double top = Math.Max(0, (clipBounds.Top - layoutBounds.Top) * scale);

                double width = Math.Min((clipBounds.Right - layoutBounds.Left) * scale, this.PageWidth) - left;
                double height = Math.Min((clipBounds.Bottom - layoutBounds.Top) * scale, this.PageHeight) - top;

                int pixelWidth = (int)Math.Round(width / scale * DPIscaling);
                int pixelHeight = (int)Math.Round(height / scale * DPIscaling);

                if (pixelWidth > 0 && pixelHeight > 0)
                {
                    RenderingParameters currentParameters = new RenderingParameters((float)left, (float)top, (float)width, (float)height, (float)(1 / scale * DPIscaling), pixelWidth, pixelHeight);

                    if (FrontBuffer != null && FrontBufferRenderingParams == currentParameters && !IsDirty)
                    {
                        if (!RenderRequestedHandle.WaitOne(0))
                        {
                            LastFrameStopwatch.Reset();
                        }
                        else
                        {
                            Avalonia.Threading.Dispatcher.UIThread.InvokeAsync(() =>
                            {
                                this.InvalidateVisual();
                            });
                        }
                        context.DrawImage(FrontBuffer, new Rect(0, 0, pixelWidth, pixelHeight), new Rect(left, top, width, height));
                    }
                    else if (FrontBuffer != null && FrontBufferRenderingParams.GoodEnough(currentParameters) && !IsDirty)
                    {
                        if (!RenderRequestedHandle.WaitOne(0))
                        {
                            LastFrameStopwatch.Reset();
                        }
                        else
                        {
                            Avalonia.Threading.Dispatcher.UIThread.InvokeAsync(() =>
                            {
                                this.InvalidateVisual();
                            });
                        }
                        context.DrawImage(FrontBuffer, new Rect(0, 0, FrontBufferRenderingParams.RenderWidth, FrontBufferRenderingParams.RenderHeight), new Rect(FrontBufferRenderingParams.Left, FrontBufferRenderingParams.Top, FrontBufferRenderingParams.Width, FrontBufferRenderingParams.Height));
                    }
                    else
                    {
                        lock (RenderingRequestLock)
                        {
                            IsDirty = false;
                            RenderingRequest = currentParameters;
```

```
                            RenderRequestedHandle.Set();
                            LastFrameStopwatch.Start();
                        }

                        if (FrontBuffer != null)
                        {
                            context.DrawImage(FrontBuffer, new Rect(0, 0, FrontBufferRenderingParams.RenderWidth, FrontBufferRenderingParams.RenderHeight), new Rect(FrontBufferRenderingParams.Left, FrontBufferRenderingParams.Top, FrontBufferRenderingParams.Width, FrontBufferRenderingParams.Height));
                        }

                        long milliseconds = LastFrameStopwatch.ElapsedMilliseconds;

                        Bitmap spinner = Spinner?.Invoke(milliseconds);

                        if (spinner != null)
                        {
                            double spinnerWidth = spinner.Size.Width * scale;
                            double spinnerHeight = spinner.Size.Height * scale;

                            double actualLeft = (clipBounds.Left - layoutBounds.Left) * scale;
                            double actualTop = (clipBounds.Top - layoutBounds.Top) * scale;

                            double actualRight = (clipBounds.Right - layoutBounds.Left) * scale;
                            double actualBottom = (clipBounds.Bottom - layoutBounds.Top) * scale;

                            context.DrawImage(spinner, new Rect(0, 0, spinner.Size.Width, spinner.Size.Height), new Rect((actualLeft + actualRight - spinnerWidth) * 0.5, (actualTop + actualBottom - spinnerHeight) * 0.5, spinnerWidth, spinnerHeight));
                        }

                        Avalonia.Threading.Dispatcher.UIThread.InvokeAsync(() =>
                        {
                            this.InvalidateVisual();
                        });
                    }
                }
            }
        }

        private bool disposedValue;

        /// <inheritdoc cref="Dispose()"/>
        protected virtual void Dispose(bool disposing)
        {
            if (!disposedValue)
            {
                if (disposing)
                {
                    this.BackBufferSkiaCanvas?.Dispose();
                    this.BackBufferSkiaCanvas2?.Dispose();
                    this.BackBufferSkiaContext?.Dispose();
                    this.BackBufferSkiaContext2?.Dispose();
                    this.BackBuffer?.Dispose();
                    this.BackBuffer2?.Dispose();
                    this.DisposedHandle?.Set();
                    this.FrontBufferSkiaCanvas?.Dispose();
                    this.FrontBufferSkiaContext?.Dispose();
                    this.FrontBuffer?.Dispose();

                    foreach (KeyValuePair<string, (SKBitmap, bool)> image in this.Images)
                    {
                        image.Value.Item1?.Dispose();
                    }

                    for (int i = 0; i < this.RenderActions?.Count; i++)
                    {
                        foreach (SKRenderAction act in this.RenderActions[i])
                        {
                            act?.Dispose();
                        }

                        this.LayerTransforms?[i]?.Dispose();
                    }
                }

                disposedValue = true;
            }
        }

        /// <inheritdoc/>
        public void Dispose()
        {
            Dispose(disposing: true);
            GC.SuppressFinalize(this);
        }
```

```
    }
}
```

8.5 SKRenderContext.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using SkiaSharp;
using System;
using System.Collections.Generic;
using System.Linq;
using VectSharp.Filters;

namespace VectSharp.Canvas
{

    /// <summary>
    /// Represents a light-weight rendering action.
    /// </summary>
    public class SKRenderAction : IDisposable
    {
        /// <summary>
        /// Returns a boolean value indicating whether the current instance has been disposed.
        /// </summary>
        public bool Disposed => disposedValue;

        private bool disposedValue;

        /// <summary>
        /// Types of rendering actions.
        /// </summary>
        public enum ActionTypes
        {
            /// <summary>
            /// The render action represents a path object.
            /// </summary>
            Path,

            /// <summary>
            /// The render action represents a text object.
            /// </summary>
            Text,

            /// <summary>
            /// The render action represents a raster image.
            /// </summary>
            RasterImage,

            /// <summary>
            /// The render action represents a transformation of the coordinate space.
            /// </summary>
            Transform,

            /// <summary>
            /// The render action represents saving the current graphics state.
            /// </summary>
            Save,

            /// <summary>
            /// The render action represents restoring the last saved graphics state.
            /// </summary>
            Restore,

            /// <summary>
            /// The render action represents an update of the current clip path.
            /// </summary>
            Clip,

```

```
            /// <summary>
            /// The render action represents rendering a graphics object with a filter.
            /// </summary>
            DrawFiltered
        }

        /// <summary>
        /// Type of the rendering action.
        /// </summary>
        public ActionTypes ActionType { get; private set; }

        /// <summary>
        /// Path that needs to be rendered (null if the action type is not <see cref="ActionTypes.Path"/>). If you change this, you probably want to call this object's <see cref="InvalidateHitTestPath"/> method.
        /// </summary>
        public SKPath Path { get; set; }

        internal SKPath HitTestPath { get; set; }
        internal SKPath LastRenderedGlobalHitTestPath { get; set; }

        /// <summary>
        /// Text that needs to be rendered (null if the action type is not <see cref="ActionTypes.Text"/>). If you change this, you probably want to call this object's <see cref="InvalidateHitTestPath"/> method.
        /// </summary>
        public string Text { get; set; }

        /// <summary>
        /// The font that will be used to render the text (null if the action type is not <see cref="ActionTypes.Text"/>). If you change this, you probably want to call this object's <see cref="InvalidateHitTestPath"/> method.
        /// </summary>
        public SKFont Font { get; set; }

        /// <summary>
        /// The X coordainate at which the text will be drawn (null if the action type is not <see cref="ActionTypes.Text"/>). If you change this, you probably want to call this object's <see cref="InvalidateHitTestPath"/> method.
        /// </summary>
        public float TextX { get; set; }

        /// <summary>
        /// The Y coordainate at which the text will be drawn (null if the action type is not <see cref="ActionTypes.Text"/>). If you change this, you probably want to call this object's <see cref="InvalidateHitTestPath"/> method.
        /// </summary>
        public float TextY { get; set; }

        /// <summary>
        /// Paint used to render the text or path (<see langword="null"/> if the action type is neither <see cref="ActionTypes.Text"/> nor <see cref="ActionTypes.Path"/>). If you change this, you probably want to call this object's <see cref="InvalidateHitTestPath"/> method.
        /// </summary>
        public SKPaint Paint { get; set; }

        /// <summary>
        /// Univocal identifier of the image that needs to be drawn.
        /// </summary>
        public string ImageId { get; set; }

        /// <summary>
        /// The source rectangle of the image (<see langword="null"/> if the action type is not <see cref="ActionTypes.RasterImage"/>). If you change this, you probably want to call this object's <see cref="InvalidateVisual"/> method.
        /// </summary>
        public SKRect? ImageSource { get; set; }

        /// <summary>
        /// The destination rectangle of the image (<see langword="null"/> if the action type is not <see cref="ActionTypes.RasterImage"/>). If you change this, you probably want to call this object's <see cref="InvalidateHitTestPath"/> method.
        /// </summary>
        public SKRect? ImageDestination { get; set; }

        /// <summary>
        /// The transformation matrix that will be applied to the current coordinate system (<see langword="null"/> if the action type is not <see cref="ActionTypes.Transform"/>). If you change this, you probably want to call this object's <see cref="InvalidateVisual"/> method.
        /// </summary>
        public SKMatrix? Transform { get; set; } = null;

        /// <summary>
        /// A tag to access the <see cref="SKRenderAction"/>.
        /// </summary>
        public string Tag { get; private set; }

```

```
        /// <summary>
        /// The Z-index of the rendering action (an action with a higher Z-index will always appear above an action with a lower Z-index).
        /// The more different values there are for the Z-index, the slower the rendering, so keep use of this property to a minimum.
        /// If you change this, you probably want to call this object's <see cref="InvalidateZIndex"/> method.
        /// </summary>
        public uint ZIndex { get; set; } = 0;

        /// <summary>
        /// The graphics that will be drawn with the specified <see cref="Filter"/>. If you change this, you probably want to call this object's <see cref="InvalidateVisual"/> method.
        /// </summary>
        public SKRenderContext Graphics { get; set; } = null;

        /// <summary>
        /// The filter with which the <see cref="Graphics"/> is drawn. If you change this, you probably want to call this object's <see cref="InvalidateVisual"/> method.
        /// </summary>
        public IFilter Filter { get; set; }

        /// <summary>
        /// An arbitrary object associated with the RenderAction.
        /// </summary>
        public object Payload { get; set; }

        internal ISKRenderCanvas InternalParent { get; set; }

        /// <summary>
        /// The container of this <see cref="SKRenderAction"/>.
        /// </summary>
        public Avalonia.Controls.Canvas Parent
        {
            get
            {
                return (Avalonia.Controls.Canvas)InternalParent;
            }
        }

        /// <summary>
        /// Raised when the pointer enters the area covered by the <see cref="SKRenderAction"/>.
        /// </summary>
        public event EventHandler<Avalonia.Input.PointerEventArgs> PointerEnter;

        /// <summary>
        /// Raised when the pointer leaves the area covered by the <see cref="SKRenderAction"/>.
        /// </summary>
        public event EventHandler<Avalonia.Input.PointerEventArgs> PointerLeave;

        /// <summary>
        /// Raised when the pointer is pressed while over the area covered by the <see cref="SKRenderAction"/>.
        /// </summary>
        public event EventHandler<Avalonia.Input.PointerPressedEventArgs> PointerPressed;

        /// <summary>
        /// Raised when the pointer is released after a <see cref="PointerPressed"/> event.
        /// </summary>
        public event EventHandler<Avalonia.Input.PointerReleasedEventArgs> PointerReleased;


        internal void FirePointerEnter(Avalonia.Input.PointerEventArgs e)
        {
            this.PointerEnter?.Invoke(this, e);
        }

        internal void FirePointerLeave(Avalonia.Input.PointerEventArgs e)
        {
            this.PointerLeave?.Invoke(this, e);
        }

        internal void FirePointerPressed(Avalonia.Input.PointerPressedEventArgs e)
        {
            this.PointerPressed?.Invoke(this, e);
        }

        internal void FirePointerReleased(Avalonia.Input.PointerReleasedEventArgs e)
        {
            this.PointerReleased?.Invoke(this, e);
        }

        /// <summary>
        /// <para>Signals to this object that its shape has changed a new path needs to be computed for the purpose of hit-testing.
        /// Also signals to the <see cref="Parent"/> that the visual properties of this object have changed and triggers a redraw.</para>
```

```
        /// <para>This method should be called whenever the "shape" of the object represented by the <see cref="SKRenderAction"/> changes.
        /// If only the visual properties of this object have changed (e.g. the colour), call the <see cref="InvalidateVisual"/> method instead.</para>
        /// <para>If you make changes to more than one <see cref="SKRenderAction"/> contained in the same <see cref="Canvas"/>, you only need to invalidate the last one.</para>
        /// <para>This method should only be called after the output has been fully initialized.</para>
        /// </summary>
        public void InvalidateHitTestPath()
        {
            CreateHitTestPath();

            this.InvalidateVisual();
        }

        internal void CreateHitTestPath()
        {
            if (this.ActionType == ActionTypes.Path)
            {
                if (this.Path != null && this.Paint != null)
                {
                    this.HitTestPath?.Dispose();
                    this.HitTestPath = this.Paint.GetFillPath(this.Path);
                }
            }
            else if (this.ActionType == ActionTypes.RasterImage)
            {
                if (this.ImageDestination != null)
                {
                    this.HitTestPath?.Dispose();
                    this.HitTestPath = new SKPath();
                    this.HitTestPath.AddRect(this.ImageDestination.Value);
                }
            }
            else if (this.ActionType == ActionTypes.Text)
            {
                if (this.Paint != null && this.Font != null)
                {
                    this.HitTestPath?.Dispose();
                    this.Paint.Typeface = this.Font.Typeface;
                    this.Paint.TextSize = this.Font.Size;

                    using (SKPath pth = this.Paint.GetTextPath(this.Text, this.TextX, this.TextY))
                    {
                        this.HitTestPath = this.Paint.GetFillPath(pth);
                    }
                }
            }
        }

        /// <summary>
        /// <para>This methods signals to the <see cref="Parent"/> that the visual properties (e.g. the colour) of this object have changed and triggers a redraw.</para>
        /// <para>If the "shape" of the object has changed as well, call the <see cref="InvalidateHitTestPath"/> method instead. If the Z-index of the
        /// object has changed, call the <see cref="InvalidateZIndex"/> method instead. If both the "shape" and the Z-index of the object have changed,
        /// call the <see cref="InvalidateAll"/> method.</para>
        /// <para>If you make changes to more than one <see cref="SKRenderAction"/> contained in the same <see cref="Canvas"/>, you only need to invalidate the last one.</para>
        /// <para>This method should only be called after the output has been fully initialized.</para>
        /// </summary>
        public void InvalidateVisual()
        {
            this.InternalParent?.InvalidateDirty();
        }

        /// <summary>
        /// <para>This methods signals to the <see cref="Parent"/> that the Z-index and visual properties (e.g. the colour) of this object have changed and triggers a redraw.</para>
        /// <para>If the "shape" of the object has changed as well, call the <see cref="InvalidateAll"/> method instead.</para>
        /// <para>If you make changes to more than one <see cref="SKRenderAction"/> contained in the same <see cref="Canvas"/>, you only need to invalidate the last one.</para>
        /// <para>This method should only be called after the output has been fully initialized.</para>
        /// </summary>
        public void InvalidateZIndex()
        {
            this.InternalParent?.InvalidateZIndex();
        }

        /// <summary>
        /// <para>This methods signals to the <see cref="Parent"/> that the Z-index, shape and visual properties (e.g. the colour) of this object have changed and triggers a redraw.</para>
```

```
        /// <para>If you make changes to more than one <see cref="SKRenderAction"/> contained in the same <see cref="Canvas"/>, you only need to invalidate the last one.</para>
        /// <para>This method should only be called after the output has been fully initialized.</para>
        /// </summary>
        public void InvalidateAll()
        {
            this.InvalidateHitTestPath();
            this.InvalidateZIndex();
        }

        private SKRenderAction()
        {

        }

        /// <summary>
        /// Creates a new <see cref="SKRenderAction"/> representing a path.
        /// </summary>
        /// <param name="path">The geometry to be rendered.</param>
        /// <param name="paint">The paint used to fill or stroke the path.</param>
        /// <param name="tag">A tag to access the <see cref="SKRenderAction"/>. If this is null this <see cref="SKRenderAction"/> is not visible in the hit test.</param>
        /// <returns>A new <see cref="SKRenderAction"/> representing a path.</returns>
        public static SKRenderAction PathAction(SKPath path, SKPaint paint, string tag = null)
        {
            SKRenderAction act = new SKRenderAction()
            {
                ActionType = ActionTypes.Path,
                Path = path,
                Paint = paint,
                Tag = tag
            };

            if (!string.IsNullOrEmpty(tag))
            {
                act.CreateHitTestPath();
            }

            return act;
        }

        /// <summary>
        /// Creates a new <see cref="SKRenderAction"/> representing a clipping action.
        /// </summary>
        /// <param name="clippingPath">The path to be used for clipping.</param>
        /// <param name="tag">A tag to access the <see cref="SKRenderAction"/>.</param>
        /// <returns>A new <see cref="SKRenderAction"/> representing a clipping action.</returns>
        public static SKRenderAction ClipAction(SKPath clippingPath, string tag = null)
        {
            return new SKRenderAction()
            {
                ActionType = ActionTypes.Clip,
                Path = clippingPath,
                Tag = tag
            };
        }

        /// <summary>
        /// Creates a new <see cref="SKRenderAction"/> representing text.
        /// </summary>
        /// <param name="text">The text to be rendered.</param>
        /// <param name="x">The X coordinate at which the text will be drawn.</param>
        /// <param name="y">The Y coordinate at which the text will be drawn.</param>
        /// <param name="font">The font to be used to render the text.</param>
        /// <param name="paint">The paint to be used to fill or stroke the text.</param>
        /// <param name="tag">A tag to access the <see cref="SKRenderAction"/>. If this is null this <see cref="SKRenderAction"/> is not visible in the hit test.</param>
        /// <returns>A new <see cref="SKRenderAction"/> representing text.</returns>
        public static SKRenderAction TextAction(string text, float x, float y, SKFont font, SKPaint paint, string tag = null)
        {
            SKRenderAction act = new SKRenderAction()
            {
                ActionType = ActionTypes.Text,
                Text = text,
                TextX = x,
                TextY = y,
                Font = font,
                Paint = paint,
                Tag = tag
            };

            act.Paint.Typeface = font.Typeface;
            act.Paint.TextSize = font.Size;

            if (!string.IsNullOrEmpty(tag))
```

```
            {
                act.CreateHitTestPath();
            }

            return act;
        }

        /// <summary>
        /// Creates a new <see cref="SKRenderAction"/> representing an image.
        /// </summary>
        /// <param name="imageId">The univocal identifier of the image to draw.</param>
        /// <param name="sourceRect">The source rectangle of the image.</param>
        /// <param name="destinationRect">The destination rectangle of the image.</param>
        /// <param name="tag">A tag to access the <see cref="SKRenderAction"/>. If this is null this <see cref="SKRenderAction"/> is not visible in the hit test.</param>
        /// <returns>A new <see cref="SKRenderAction"/> representing an image.</returns>
        public static SKRenderAction ImageAction(string imageId, SKRect sourceRect, SKRect destinationRect, string tag = null)
        {
            SKRenderAction act = new SKRenderAction()
            {
                ActionType = ActionTypes.RasterImage,
                ImageId = imageId,
                ImageSource = sourceRect,
                ImageDestination = destinationRect,
                Tag = tag
            };

            if (!string.IsNullOrEmpty(tag))
            {
                act.CreateHitTestPath();
            }

            return act;
        }

        /// <summary>
        /// Creates a new <see cref="SKRenderAction"/> representing a transform.
        /// </summary>
        /// <param name="transform">The transform to apply.</param>
        /// <param name="tag">A tag to access the <see cref="SKRenderAction"/>.</param>
        /// <returns>A new <see cref="SKRenderAction"/> representing a transform.</returns>
        public static SKRenderAction TransformAction(SKMatrix transform, string tag = null)
        {
            return new SKRenderAction()
            {
                ActionType = ActionTypes.Transform,
                Transform = transform,
                Tag = tag
            };
        }

        /// <summary>
        /// Creates a new <see cref="SKRenderAction"/> that saves the current graphics state.
        /// </summary>
        /// <param name="tag">A tag to access the <see cref="SKRenderAction"/>.</param>
        /// <returns>A new <see cref="SKRenderAction"/> that saves the current graphics state.</returns>
        public static SKRenderAction SaveAction(string tag = null)
        {
            return new SKRenderAction()
            {
                ActionType = ActionTypes.Save,
                Tag = tag
            };
        }

        /// <summary>
        /// Creates a new <see cref="SKRenderAction"/> that saves the current graphics state.
        /// </summary>
        /// <param name="tag">A tag to access the <see cref="SKRenderAction"/>.</param>
        /// <returns>A new <see cref="SKRenderAction"/> that restores the last saved graphics state.</returns>
        public static SKRenderAction RestoreAction(string tag = null)
        {
            return new SKRenderAction()
            {
                ActionType = ActionTypes.Restore,
                Tag = tag
            };
        }

        /// <summary>
        /// Create a new <see cref="SKRenderAction"/> that draws some graphics with a filter.
        /// </summary>
        /// <returns>A new <see cref="SKRenderAction"/> that draws some graphics with a filter.</returns>
```

```
        public static SKRenderAction DrawFilteredGraphicsAction(SKRenderContext graphics, IFilter filter, string tag = null)
        {
            return new SKRenderAction()
            {
                ActionType = ActionTypes.DrawFiltered,
                Graphics = graphics,
                Filter = filter,
                Tag = tag
            };
        }

        /// <inheritdoc cref="IDisposable.Dispose"/>
        protected virtual void Dispose(bool disposing)
        {
            if (!disposedValue)
            {
                if (disposing)
                {
                    this.Font?.Dispose();
                    this.HitTestPath?.Dispose();
                    this.LastRenderedGlobalHitTestPath?.Dispose();
                    this.Paint?.Dispose();
                    this.Path?.Dispose();
                }

                disposedValue = true;
            }
        }

        /// <inheritdoc cref="IDisposable.Dispose"/>
        public void Dispose()
        {
            Dispose(disposing: true);
            GC.SuppressFinalize(this);
        }
    }


    internal static class SKTypefaceCache
    {
        private static readonly object LockObject = new object();
        private static readonly Dictionary<string, SKTypeface> Typefaces = new Dictionary<string, SKTypeface>();
        public static SKTypeface GetSKTypeface(FontFamily family)
        {

            lock (LockObject)
            {
                if (Typefaces.TryGetValue(family.FileName, out SKTypeface tbr))
                {
                    return tbr;
                }
                else
                {
                    try
                    {
                        System.IO.MemoryStream fontStream = new System.IO.MemoryStream((int)family.TrueTypeFile.FontStream.Length);
                        family.TrueTypeFile.FontStream.Seek(0, System.IO.SeekOrigin.Begin);
                        family.TrueTypeFile.FontStream.CopyTo(fontStream);
                        fontStream.Seek(0, System.IO.SeekOrigin.Begin);

                        SKTypeface typeface = SKTypeface.FromData(SKData.Create(fontStream));

                        Typefaces[family.FileName] = typeface;

                        return typeface;
                    }
                    catch
                    {
                        SKTypeface typeface = SKTypeface.Default;

                        Typefaces[family.FileName] = typeface;

                        return typeface;
                    }
                }
            }
        }
    }

    /// <summary>
    /// Represents a page that has been prepared for fast rendering using the SkiaSharp renderer.
    /// </summary>
    public class SKRenderContext
    {
```

```
        internal virtual Dictionary<string, (SKBitmap, bool)> Images { get; set; }
        internal virtual List<SKRenderAction> SKRenderActions { get; set; }
    }


    internal class SKRenderContextImpl : SKRenderContext, IGraphicsContext, IDisposable
    {
        public Dictionary<string, Func<SKRenderAction, IEnumerable<SKRenderAction>> TaggedActions { get; set; } = new Dictionary<string, Func<SKRenderAction, IEnumerable<SKRenderAction>>();

        private readonly bool removeTaggedActions = true;

        public string Tag { get; set; }

        private readonly AvaloniaContextInterpreter.TextOptions _textOption;

        internal override Dictionary<string, (SKBitmap, bool)> Images { get; set; }

        private readonly FilterOption _filterOption;

        public SKRenderContextImpl(double width, double height, bool removeTaggedActionsAfterExecution, AvaloniaContextInterpreter.TextOptions textOption, Dictionary<string, (SKBitmap, bool)> images, FilterOption filterOption)
        {
            this.Images = images;

            currentPath = null;
            figureInitialised = false;

            SKRenderActions = new List<SKRenderAction>();
            removeTaggedActions = removeTaggedActionsAfterExecution;

            Width = width;
            Height = height;

            _textOption = textOption;
            _filterOption = filterOption;
        }

        internal override List<SKRenderAction> SKRenderActions { get; set; }

        public double Width { get; private set; }
        public double Height { get; private set; }

        private void AddAction(SKRenderAction act)
        {
            if (!string.IsNullOrEmpty(Tag))
            {
                if (TaggedActions.ContainsKey(Tag))
                {
                    IEnumerable<SKRenderAction> actions = TaggedActions[Tag](act);

                    foreach (SKRenderAction action in actions)
                    {
                        SKRenderActions.Add(action);
                    }

                    if (removeTaggedActions)
                    {
                        TaggedActions.Remove(Tag);
                    }
                }
                else
                {
                    SKRenderActions.Add(act);
                }
            }
            else if (TaggedActions.ContainsKey(""))
            {
                IEnumerable<SKRenderAction> actions = TaggedActions[""](act);

                foreach (SKRenderAction action in actions)
                {
                    SKRenderActions.Add(action);
                }
            }
            else
            {
                SKRenderActions.Add(act);
            }
        }

        public void Translate(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            SKRenderAction act = SKRenderAction.TransformAction(SKMatrix.CreateTranslation((float)x,
```

```
(float)y), Tag);
            AddAction(act);

            currentPath = null;
            figureInitialised = false;
        }

        public TextBaselines TextBaseline { get; set; }

        private void PathText(string text, double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            GraphicsPath textPath = new GraphicsPath().AddText(x, y, text, Font, TextBaseline);

            for (int j = 0; j < textPath.Segments.Count; j++)
            {
                switch (textPath.Segments[j].Type)
                {
                    case VectSharp.SegmentType.Move:
                        this.MoveTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.Line:
                        this.LineTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.CubicBezier:
                        this.CubicBezierTo(textPath.Segments[j].Points[0].X,
textPath.Segments[j].Points[0].Y, textPath.Segments[j].Points[1].X, textPath.Segments[j].Points[1].Y,
textPath.Segments[j].Points[2].X, textPath.Segments[j].Points[2].Y);
                        break;
                    case VectSharp.SegmentType.Close:
                        this.Close();
                        break;
                }
            }
        }

        public void StrokeSimpleText(string text, double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if ((_textOption == AvaloniaContextInterpreter.TextOptions.NeverConvert || (_textOption ==
AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary && (Font.FontFamily.IsStandardFamily ||
Font.FontFamily.TrueTypeFile?.FontStream != null))) &&
!System.Runtime.InteropServices.RuntimeInformation.IsOSPlatform(System.Runtime.InteropServices.OSPlatform.Linux))
            {
                SKTypeface typeface = SKTypefaceCache.GetSKTypeface(Font.FontFamily);

                double top = y;
                double left = x;

                Font.DetailedFontMetrics metrics = Font.MeasureTextAdvanced(text);

                if (TextBaseline == TextBaselines.Top)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        left -= metrics.LeftSideBearing;
                        top += metrics.Top;
                    }
                }
                else if (TextBaseline == TextBaselines.Middle)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        left -= metrics.LeftSideBearing;
                        top += (metrics.Top + metrics.Bottom) * 0.5;
                    }
                }
                else if (TextBaseline == TextBaselines.Baseline)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        left -= metrics.LeftSideBearing;
                    }
                }
                else if (TextBaseline == TextBaselines.Bottom)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        left -= metrics.LeftSideBearing;
                        top += metrics.Bottom;
                    }
                }

                SKPaint stroke = new SKPaint() { IsStroke = true, IsAntialias = true, Style =
SKPaintStyle.Stroke, StrokeWidth = (float)LineWidth, SubpixelText = true };
```

```

                if (this.StrokeStyle is SolidColourBrush solid)
                {
                    stroke.Color = new SKColor((byte)(solid.R * 255), (byte)(solid.G * 255), (byte)(solid.B * 255), StrokeAlpha);
                }
                else if (this.StrokeStyle is LinearGradientBrush linearGradient)
                {
                    stroke.Shader = linearGradient.ToSKShader();
                }
                else if (this.StrokeStyle is RadialGradientBrush radialGradient)
                {
                    stroke.Shader = radialGradient.ToSKShader();
                }

                stroke.PathEffect = SKPathEffect.CreateDash(new float[] { (float)LineDash[0], (float)LineDash[1] }, (float)LineDash[2]);

                switch (LineCap)
                {
                    case LineCaps.Butt:
                        stroke.StrokeCap = SKStrokeCap.Butt;
                        break;
                    case LineCaps.Round:
                        stroke.StrokeCap = SKStrokeCap.Round;
                        break;
                    case LineCaps.Square:
                        stroke.StrokeCap = SKStrokeCap.Square;
                        break;
                }

                switch (LineJoin)
                {
                    case LineJoins.Bevel:
                        stroke.StrokeJoin = SKStrokeJoin.Bevel;
                        break;
                    case LineJoins.Round:
                        stroke.StrokeJoin = SKStrokeJoin.Round;
                        break;
                    case LineJoins.Miter:
                        stroke.StrokeJoin = SKStrokeJoin.Miter;
                        break;
                }

                SKRenderAction act = SKRenderAction.TextAction(text, (float)left, (float)top, new SKFont(typeface, (float)Font.FontSize), stroke, Tag);

                AddAction(act);
            }
            else
            {
                PathText(text, x, y);
                Stroke();
            }
        }

        public void FillText(string text, double x, double y)
        {
            if (!Font.EnableKerning)
            {
                FillSimpleText(text, x, y);
            }
            else
            {
                List<(string, Point)> tSpans = new List<(string, Point)>();

                System.Text.StringBuilder currentRun = new System.Text.StringBuilder();
                Point currentKerning = new Point();

                Point currentGlyphPlacementDelta = new Point();
                Point currentGlyphAdvanceDelta = new Point();
                Point nextGlyphPlacementDelta = new Point();
                Point nextGlyphAdvanceDelta = new Point();

                for (int i = 0; i < text.Length; i++)
                {
                    if (i < text.Length - 1)
                    {
                        currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                        currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                        nextGlyphAdvanceDelta = new Point();
                        nextGlyphPlacementDelta = new Point();

                        TrueTypeFile.PairKerning kerning = Font.FontFamily.TrueTypeFile.Get1000EmKerning(text[i], text[i + 1]);

                        if (kerning != null)
```

```
                        {
                            currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                            currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                            nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                            nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                        }
                    }

                    if (currentGlyphPlacementDelta.X != 0 || currentGlyphPlacementDelta.Y != 0 || currentGlyphAdvanceDelta.X != 0 || currentGlyphAdvanceDelta.Y != 0)
                    {
                        if (currentRun.Length > 0)
                        {
                            tSpans.Add((currentRun.ToString(), currentKerning));

                            tSpans.Add((text[i].ToString(), new Point(currentGlyphPlacementDelta.X * Font.FontSize / 1000, currentGlyphPlacementDelta.Y * Font.FontSize / 1000)));

                            currentRun.Clear();
                            currentKerning = new Point((currentGlyphAdvanceDelta.X - currentGlyphPlacementDelta.X) * Font.FontSize / 1000, (currentGlyphAdvanceDelta.Y - currentGlyphPlacementDelta.Y) * Font.FontSize / 1000);
                        }
                        else
                        {
                            tSpans.Add((text[i].ToString(), new Point(currentGlyphPlacementDelta.X * Font.FontSize / 1000 + currentKerning.X, currentGlyphPlacementDelta.Y * Font.FontSize / 1000 + currentKerning.Y)));

                            currentRun.Clear();
                            currentKerning = new Point((currentGlyphAdvanceDelta.X - currentGlyphPlacementDelta.X) * Font.FontSize / 1000, (currentGlyphAdvanceDelta.Y - currentGlyphPlacementDelta.Y) * Font.FontSize / 1000);
                        }
                    }
                    else
                    {
                        currentRun.Append(text[i]);
                    }
                }

                if (currentRun.Length > 0)
                {
                    tSpans.Add((currentRun.ToString(), currentKerning));
                }

                double currX = x;
                double currY = y;

                Font.DetailedFontMetrics fullMetrics = Font.MeasureTextAdvanced(text);

                if (TextBaseline == TextBaselines.Top)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        currY += fullMetrics.Top;
                    }
                }
                else if (TextBaseline == TextBaselines.Middle)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        currY += (fullMetrics.Top + fullMetrics.Bottom) * 0.5;
                    }
                }
                else if (TextBaseline == TextBaselines.Bottom)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        currY += fullMetrics.Bottom;
                    }
                }

                TextBaseline = TextBaselines.Baseline;

                for (int i = 0; i < tSpans.Count; i++)
                {
                    Font.DetailedFontMetrics metrics = Font.MeasureTextAdvanced(tSpans[i].Item1);

                    if (i == 0)
                    {
                        FillSimpleText(tSpans[i].Item1, currX + tSpans[i].Item2.X, currY +
```

```
tSpans[i].Item2.Y);
                    }
                    else
                    {
                        FillSimpleText(tSpans[i].Item1, currX + metrics.LeftSideBearing - fullMetrics.LeftSideBearing + tSpans[i].Item2.X, currY + tSpans[i].Item2.Y);
                    }


                    currX += metrics.AdvanceWidth + tSpans[i].Item2.X;
                    currY += tSpans[i].Item2.Y;
                }
            }
        }

        public void StrokeText(string text, double x, double y)
        {
            if (!Font.EnableKerning)
            {
                StrokeSimpleText(text, x, y);
            }
            else
            {
                List<(string, Point)> tSpans = new List<(string, Point)>();

                System.Text.StringBuilder currentRun = new System.Text.StringBuilder();
                Point currentKerning = new Point();

                Point currentGlyphPlacementDelta = new Point();
                Point currentGlyphAdvanceDelta = new Point();
                Point nextGlyphPlacementDelta = new Point();
                Point nextGlyphAdvanceDelta = new Point();

                for (int i = 0; i < text.Length; i++)
                {
                    if (i < text.Length - 1)
                    {
                        currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                        currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                        nextGlyphAdvanceDelta = new Point();
                        nextGlyphPlacementDelta = new Point();

                        TrueTypeFile.PairKerning kerning = Font.FontFamily.TrueTypeFile.Get1000EmKerning(text[i], text[i + 1]);

                        if (kerning != null)
                        {
                            currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                            currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                            nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                            nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                        }
                    }

                    if (currentGlyphPlacementDelta.X != 0 || currentGlyphPlacementDelta.Y != 0 || currentGlyphAdvanceDelta.X != 0 || currentGlyphAdvanceDelta.Y != 0)
                    {
                        if (currentRun.Length > 0)
                        {
                            tSpans.Add((currentRun.ToString(), currentKerning));

                            tSpans.Add((text[i].ToString(), new Point(currentGlyphPlacementDelta.X * Font.FontSize / 1000, currentGlyphPlacementDelta.Y * Font.FontSize / 1000)));

                            currentRun.Clear();
                            currentKerning = new Point((currentGlyphAdvanceDelta.X - currentGlyphPlacementDelta.X) * Font.FontSize / 1000, (currentGlyphAdvanceDelta.Y - currentGlyphPlacementDelta.Y) * Font.FontSize / 1000);
                        }
                        else
                        {
                            tSpans.Add((text[i].ToString(), new Point(currentGlyphPlacementDelta.X * Font.FontSize / 1000 + currentKerning.X, currentGlyphPlacementDelta.Y * Font.FontSize / 1000 + currentKerning.Y)));

                            currentRun.Clear();
                            currentKerning = new Point((currentGlyphAdvanceDelta.X - currentGlyphPlacementDelta.X) * Font.FontSize / 1000, (currentGlyphAdvanceDelta.Y - currentGlyphPlacementDelta.Y) * Font.FontSize / 1000);
                        }
                    }
                    else
```

```
                    {
                        currentRun.Append(text[i]);
                    }
                }

                if (currentRun.Length > 0)
                {
                    tSpans.Add((currentRun.ToString(), currentKerning));
                }

                double currX = x;
                double currY = y;

                Font.DetailedFontMetrics fullMetrics = Font.MeasureTextAdvanced(text);

                for (int i = 0; i < tSpans.Count; i++)
                {
                    Font.DetailedFontMetrics metrics = Font.MeasureTextAdvanced(tSpans[i].Item1);

                    if (i == 0)
                    {
                        StrokeSimpleText(tSpans[i].Item1, currX + tSpans[i].Item2.X, currY + tSpans[i].Item2.Y);
                    }
                    else
                    {
                        StrokeSimpleText(tSpans[i].Item1, currX + metrics.LeftSideBearing - fullMetrics.LeftSideBearing + tSpans[i].Item2.X, currY + tSpans[i].Item2.Y);
                    }


                    currX += metrics.AdvanceWidth + tSpans[i].Item2.X;
                    currY += tSpans[i].Item2.Y;
                }
            }
        }


        public void FillSimpleText(string text, double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if ((_textOption == AvaloniaContextInterpreter.TextOptions.NeverConvert || (_textOption == AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary && (Font.FontFamily.IsStandardFamily || Font.FontFamily.TrueTypeFile?.FontStream != null))) && !System.Runtime.InteropServices.RuntimeInformation.IsOSPlatform(System.Runtime.InteropServices.OSPlatform.Linux))
            {
                SKTypeface typeface = SKTypefaceCache.GetSKTypeface(Font.FontFamily);

                double top = y;
                double left = x;

                Font.DetailedFontMetrics metrics = Font.MeasureTextAdvanced(text);

                if (TextBaseline == TextBaselines.Top)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        left -= metrics.LeftSideBearing;
                        top += metrics.Top;
                    }
                }
                else if (TextBaseline == TextBaselines.Middle)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        left -= metrics.LeftSideBearing;
                        top += (metrics.Top + metrics.Bottom) * 0.5;
                    }
                }
                else if (TextBaseline == TextBaselines.Baseline)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        left -= metrics.LeftSideBearing;
                    }
                }
                else if (TextBaseline == TextBaselines.Bottom)
                {
                    if (Font.FontFamily.TrueTypeFile != null)
                    {
                        left -= metrics.LeftSideBearing;
                        top += metrics.Bottom;
                    }
                }

                SKPaint fill = new SKPaint() { IsStroke = false, IsAntialias = true, Style =
```

```
SKPaintStyle.Fill, SubpixelText = true };

                if (this.FillStyle is SolidColourBrush solid)
                {
                    fill.Color = new SKColor((byte)(solid.R * 255), (byte)(solid.G * 255), (byte)(solid.B * 255), FillAlpha);
                }
                else if (this.FillStyle is LinearGradientBrush linearGradient)
                {
                    fill.Shader = linearGradient.ToSKShader();
                }
                else if (this.FillStyle is RadialGradientBrush radialGradient)
                {
                    fill.Shader = radialGradient.ToSKShader();
                }

                SKRenderAction act = SKRenderAction.TextAction(text, (float)left, (float)top, new SKFont(typeface, (float)Font.FontSize), fill, Tag);

                AddAction(act);
            }
            else
            {
                PathText(text, x, y);
                Fill();
            }
        }

        public Brush StrokeStyle { get; private set; } = Colour.FromRgb(0, 0, 0);
        private byte StrokeAlpha = 255;

        public Brush FillStyle { get; private set; } = Colour.FromRgb(0, 0, 0);
        private byte FillAlpha = 255;

        public void SetFillStyle((int r, int g, int b, double a) style)
        {
            FillStyle = Colour.FromRgba(style.r, style.g, style.b, (int)(style.a * 255));
            FillAlpha = (byte)(style.a * 255);
        }

        public void SetFillStyle(Brush style)
        {
            FillStyle = style;

            if (style is SolidColourBrush solid)
            {
                FillAlpha = (byte)(solid.A * 255);
            }
            else
            {
                FillAlpha = 255;
            }
        }

        public void SetStrokeStyle((int r, int g, int b, double a) style)
        {
            StrokeStyle = Colour.FromRgba(style.r, style.g, style.b, (int)(style.a * 255));
            StrokeAlpha = (byte)(style.a * 255);
        }

        public void SetStrokeStyle(Brush style)
        {
            StrokeStyle = style;

            if (style is SolidColourBrush solid)
            {
                StrokeAlpha = (byte)(solid.A * 255);
            }
            else
            {
                StrokeAlpha = 255;
            }
        }

        private double[] LineDash;

        public void SetLineDash(LineDash dash)
        {
            LineDash = new double[] { dash.UnitsOn, dash.UnitsOff, dash.Phase };
        }

        public void Rotate(double angle)
        {
            Utils.CoerceNaNAndInfinityToZero(ref angle);

            SKRenderAction act = SKRenderAction.TransformAction(SKMatrix.CreateRotation((float)angle), Tag);
```

```
            AddAction(act);

            currentPath = null;
            figureInitialised = false;
        }

        public void Transform(double a, double b, double c, double d, double e, double f)
        {
            Utils.CoerceNaNAndInfinityToZero(ref a, ref b, ref c, ref d, ref e, ref f);

            SKRenderAction act = SKRenderAction.TransformAction(new SKMatrix((float)a, (float)c,
    (float)e, (float)b, (float)d, (float)f, 0, 0, 1), Tag);
            AddAction(act);

            currentPath = null;
            figureInitialised = false;
        }

        public void Scale(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            SKRenderAction act = SKRenderAction.TransformAction(SKMatrix.CreateScale((float)x,
    (float)y), Tag);
            AddAction(act);

            currentPath = null;
            figureInitialised = false;
        }

        public void Save()
        {
            SKRenderAction act = SKRenderAction.SaveAction(Tag);
            AddAction(act);

            currentPath = null;
            figureInitialised = false;
        }

        public void Restore()
        {
            SKRenderAction act = SKRenderAction.RestoreAction(Tag);
            AddAction(act);

            currentPath = null;
            figureInitialised = false;
        }

        public double LineWidth { get; set; }
        public LineCaps LineCap { get; set; }
        public LineJoins LineJoin { get; set; }

        public Font Font { get; set; }

        private SKPath currentPath;

        private bool figureInitialised = false;
        private bool disposedValue;

        public void MoveTo(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if (currentPath == null)
            {
                currentPath = new SKPath() { FillType = SKPathFillType.EvenOdd };
            }

            currentPath.MoveTo((float)x, (float)y);
            figureInitialised = true;
        }

        public void LineTo(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if (currentPath == null)
            {
                currentPath = new SKPath() { FillType = SKPathFillType.EvenOdd };
            }

            if (!figureInitialised)
            {
                figureInitialised = true;
                currentPath.MoveTo((float)x, (float)y);
            }
            else
```

```
            {
                currentPath.LineTo((float)x, (float)y);
            }
        }

        public void Rectangle(double x0, double y0, double width, double height)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x0, ref y0, ref width, ref height);

            if (currentPath == null)
            {
                currentPath = new SKPath() { FillType = SKPathFillType.EvenOdd };
            }

            currentPath.MoveTo((float)x0, (float)y0);
            currentPath.LineTo((float)(x0 + width), (float)y0);
            currentPath.LineTo((float)(x0 + width), (float)(y0 + height));
            currentPath.LineTo((float)x0, (float)(y0 + height));

            currentPath.Close();
            figureInitialised = false;
        }

        public void CubicBezierTo(double p1X, double p1Y, double p2X, double p2Y, double p3X, double p3Y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref p1X, ref p1Y, ref p2X, ref p2Y, ref p3X, ref p3Y);

            if (currentPath == null)
            {
                currentPath = new SKPath() { FillType = SKPathFillType.EvenOdd };
            }

            if (figureInitialised)
            {
                currentPath.CubicTo((float)p1X, (float)p1Y, (float)p2X, (float)p2Y, (float)p3X, (float)p3Y);
            }
            else
            {
                currentPath.MoveTo((float)p1X, (float)p1Y);
                figureInitialised = true;
            }
        }

        public void Close()
        {
            currentPath.Close();

            figureInitialised = false;
        }

        public void Stroke()
        {
            SKPaint stroke = new SKPaint() { IsStroke = true, IsAntialias = true, Style = SKPaintStyle.Stroke, StrokeWidth = (float)LineWidth };

            if (this.StrokeStyle is SolidColourBrush solid)
            {
                stroke.Color = new SKColor((byte)(solid.R * 255), (byte)(solid.G * 255), (byte)(solid.B * 255), StrokeAlpha);
            }
            else if (this.StrokeStyle is LinearGradientBrush linearGradient)
            {
                stroke.Shader = linearGradient.ToSKShader();
            }
            else if (this.StrokeStyle is RadialGradientBrush radialGradient)
            {
                stroke.Shader = radialGradient.ToSKShader();
            }

            stroke.PathEffect = SKPathEffect.CreateDash(new float[] { (float)LineDash[0], (float)LineDash[1] }, (float)LineDash[2]);

            switch (LineCap)
            {
                case LineCaps.Butt:
                    stroke.StrokeCap = SKStrokeCap.Butt;
                    break;
                case LineCaps.Round:
                    stroke.StrokeCap = SKStrokeCap.Round;
                    break;
                case LineCaps.Square:
                    stroke.StrokeCap = SKStrokeCap.Square;
                    break;
            }

```

```
            switch (LineJoin)
            {
                case LineJoins.Bevel:
                    stroke.StrokeJoin = SKStrokeJoin.Bevel;
                    break;
                case LineJoins.Round:
                    stroke.StrokeJoin = SKStrokeJoin.Round;
                    break;
                case LineJoins.Miter:
                    stroke.StrokeJoin = SKStrokeJoin.Miter;
                    break;
            }

            SKRenderAction act = SKRenderAction.PathAction(currentPath, stroke, Tag);

            AddAction(act);

            currentPath = null;
            figureInitialised = false;
        }

        public void Fill()
        {
            SKPaint fill = new SKPaint() { IsStroke = false, IsAntialias = true, Style = SKPaintStyle.Fill };

            if (this.FillStyle is SolidColourBrush solid)
            {
                fill.Color = new SKColor((byte)(solid.R * 255), (byte)(solid.G * 255), (byte)(solid.B * 255), FillAlpha);
            }
            else if (this.FillStyle is LinearGradientBrush linearGradient)
            {
                fill.Shader = linearGradient.ToSKShader();
            }
            else if (this.FillStyle is RadialGradientBrush radialGradient)
            {
                fill.Shader = radialGradient.ToSKShader();
            }

            SKRenderAction act = SKRenderAction.PathAction(currentPath, fill, Tag);

            AddAction(act);

            currentPath = null;
            figureInitialised = false;
        }

        public void SetClippingPath()
        {
            SKRenderAction act = SKRenderAction.ClipAction(currentPath, Tag);

            AddAction(act);

            currentPath = null;
            figureInitialised = false;
        }

        public void DrawRasterImage(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image)
        {
            Utils.CoerceNaNAndInfinityToZero(ref destinationX, ref destinationY, ref destinationWidth, ref destinationHeight);

            if (!this.Images.ContainsKey(image.Id))
            {
                SKBitmap bmp = SKBitmap.Decode(image.PNGStream);
                this.Images.Add(image.Id, (bmp, image.Interpolate));
            }

            SKRenderAction act = SKRenderAction.ImageAction(image.Id, new SKRect(sourceX, sourceY, sourceX + sourceWidth, sourceY + sourceHeight), new SKRect((float)destinationX, (float)destinationY, (float)(destinationX + destinationWidth), (float)(destinationY + destinationHeight)), Tag);

            AddAction(act);
        }

        public void DrawFilteredGraphics(Graphics graphics, IFilter filter)
        {
            if (this._filterOption.Operation == FilterOption.FilterOperations.RasteriseAllWithSkia)
            {
                double scale = this._filterOption.RasterisationResolution;

                Rectangle bounds = graphics.GetBounds();

                bounds = new Rectangle(bounds.Location.X - filter.TopLeftMargin.X, bounds.Location.Y -
```

```
filter.TopLeftMargin.Y, bounds.Size.Width + filter.TopLeftMargin.X + filter.BottomRightMargin.X, bounds.Size.Height + filter.TopLeftMargin.Y + filter.BottomRightMargin.Y);

                if (bounds.Size.Width > 0 && bounds.Size.Height > 0)
                {
                    if (!this._filterOption.RasterisationResolutionRelative)
                    {
                        scale = scale / Math.Min(bounds.Size.Width, bounds.Size.Height);
                    }

                    RasterImage rasterised = SKRenderContextInterpreter.Rasterise(graphics, bounds, scale, true);
                    RasterImage filtered = null;

                    if (filter is IFilterWithRasterisableParameter filterWithRastParam)
                    {
                        filterWithRastParam.RasteriseParameter(SKRenderContextInterpreter.Rasterise, scale);
                    }

                    if (filter is ILocationInvariantFilter locInvFilter)
                    {
                        filtered = locInvFilter.Filter(rasterised, scale);
                    }
                    else if (filter is IFilterWithLocation filterWithLoc)
                    {
                        filtered = filterWithLoc.Filter(rasterised, bounds, scale);
                    }

                    if (filtered != null)
                    {
                        rasterised.Dispose();

                        DrawRasterImage(0, 0, filtered.Width, filtered.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, filtered);
                    }
                }
            }
            else if (this._filterOption.Operation == FilterOption.FilterOperations.RasteriseAllWithVectSharp)
            {
                double scale = this._filterOption.RasterisationResolution;

                Rectangle bounds = graphics.GetBounds();

                bounds = new Rectangle(bounds.Location.X - filter.TopLeftMargin.X, bounds.Location.Y - filter.TopLeftMargin.Y, bounds.Size.Width + filter.TopLeftMargin.X + filter.BottomRightMargin.X, bounds.Size.Height + filter.TopLeftMargin.Y + filter.BottomRightMargin.Y);

                if (bounds.Size.Width > 0 && bounds.Size.Height > 0)
                {
                    if (!this._filterOption.RasterisationResolutionRelative)
                    {
                        scale = scale / Math.Min(bounds.Size.Width, bounds.Size.Height);
                    }

                    if (graphics.TryRasterise(bounds, scale, true, out RasterImage rasterised))
                    {
                        RasterImage filtered = null;

                        if (filter is IFilterWithRasterisableParameter filterWithRastParam)
                        {
                            filterWithRastParam.RasteriseParameter(SKRenderContextInterpreter.Rasterise, scale);
                        }

                        if (filter is ILocationInvariantFilter locInvFilter)
                        {
                            filtered = locInvFilter.Filter(rasterised, scale);
                        }
                        else if (filter is IFilterWithLocation filterWithLoc)
                        {
                            filtered = filterWithLoc.Filter(rasterised, bounds, scale);
                        }

                        if (filtered != null)
                        {
                            rasterised.Dispose();

                            DrawRasterImage(0, 0, filtered.Width, filtered.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, filtered);
                        }
                    }
                    else
                    {
                        throw new NotImplementedException(@"The filter could not be rasterised! You can avoid this error by doing one of the following:
```

```
• Add a reference to VectSharp.Raster or VectSharp.Raster.ImageSharp (you may also need to add a using directive somewhere to force the assembly to be loaded).
• Provide your own implementation of Graphics.RasterisationMethod.
• Set the FilterOption.Operation to ""RasteriseAllWithSkia"", ""IgnoreAll"" or ""SkipAll"".");
                    }
                }
            }
            else if (this._filterOption.Operation == FilterOption.FilterOperations.IgnoreAll)
            {
                graphics.CopyToIGraphicsContext(this);
            }
            else
            {

            }
        }

        protected virtual void Dispose(bool disposing)
        {
            if (!disposedValue)
            {
                if (disposing)
                {
                    this.currentPath?.Dispose();
                }

                disposedValue = true;
            }
        }

        public void Dispose()
        {
            Dispose(disposing: true);
            GC.SuppressFinalize(this);
        }
    }

    /// <summary>
    /// Determines how and whether image filters are rasterised.
    /// </summary>
    public class FilterOption
    {
        /// <summary>
        /// Defines whether image filters should be rasterised or not.
        /// </summary>
        public enum FilterOperations
        {
            /// <summary>
            /// Image filters will always be rasterised using the SkiaSharp backend.
            /// </summary>
            RasteriseAllWithSkia,

            /// <summary>
            /// Image filters will always be rasterised using the VectSharp.Raster or VectSharp.Raster.ImageSharp. This option requires a reference to VectSharp.Raster or to VectSharp.Raster.ImageSharp to be added.
            /// </summary>
            RasteriseAllWithVectSharp,

            /// <summary>
            /// All image filters will be ignored.
            /// </summary>
            IgnoreAll,

            /// <summary>
            /// All the images that should be drawn with a filter will be ignored.
            /// </summary>
            SkipAll
        }

        /// <summary>
        /// Defines whether image filters should be rasterised or not.
        /// </summary>
        public FilterOperations Operation { get; } = FilterOperations.RasteriseAllWithSkia;

        /// <summary>
        /// The resolution that will be used to rasterise image filters. Depending on the value of <see cref="RasterisationResolutionRelative"/>, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.
        /// </summary>
        public double RasterisationResolution { get; } = 1;

        /// <summary>
        /// Determines whether the value of <see cref="RasterisationResolution"/> is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).
        /// </summary>
```

```
        public bool RasterisationResolutionRelative { get; } = true;

        /// <summary>
        /// The default options for image filter rasterisation.
        /// </summary>
        public static FilterOption Default = new FilterOption(FilterOperations.RasteriseAllWithSkia, 1, true);

        /// <summary>
        /// Create a new <see cref="FilterOption"/> object.
        /// </summary>
        /// <param name="operation">Defines whether image filters should be rasterised or not.</param>
        /// <param name="rasterisationResolution">The resolution that will be used to rasterise image filters. Depending on the value of <see cref="RasterisationResolutionRelative"/>, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.</param>
        /// <param name="rasterisationResolutionRelative">Determines whether the value of <see cref="RasterisationResolution"/> is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).</param>
        public FilterOption(FilterOperations operation, double rasterisationResolution, bool rasterisationResolutionRelative)
        {
            this.Operation = operation;
            this.RasterisationResolution = rasterisationResolution;
            this.RasterisationResolutionRelative = rasterisationResolutionRelative;
        }
    }

    /// <summary>
    /// Contains methods to render a <see cref="Page"/> to an <see cref="Avalonia.Controls.Canvas"/> using the SkiaSharp renderer.
    /// </summary>
    public static class SKRenderContextInterpreter
    {
        internal static SKColor ToSKColor(this Colour colour)
        {
            return new SKColor((byte)(255 * colour.R), (byte)(255 * colour.G), (byte)(255 * colour.B), (byte)(255 * colour.A));
        }

        internal static SKShader ToSKShader(this LinearGradientBrush brush)
        {
            return SKShader.CreateLinearGradient(new SKPoint((float)brush.StartPoint.X, (float)brush.StartPoint.Y), new SKPoint((float)brush.EndPoint.X, (float)brush.EndPoint.Y), (from el in brush.GradientStops select el.Colour.ToSKColor()).ToArray(), (from el in brush.GradientStops select (float)el.Offset).ToArray(), SKShaderTileMode.Clamp);
        }

        internal static SKShader ToSKShader(this RadialGradientBrush brush)
        {
            return SKShader.CreateTwoPointConicalGradient(new SKPoint((float)brush.FocalPoint.X, (float)brush.FocalPoint.Y), 0, new SKPoint((float)brush.Centre.X, (float)brush.Centre.Y), (float)brush.Radius, (from el in brush.GradientStops select el.Colour.ToSKColor()).ToArray(), (from el in brush.GradientStops select (float)el.Offset).ToArray(), SKShaderTileMode.Clamp);
        }

        /// <summary>
        /// Render a <see cref="Document"/> to an <see cref="Avalonia.Controls.Canvas"/> using the SkiaSharp renderer. Each page corresponds to a layer in the image.
        /// </summary>
        /// <param name="document">The <see cref="Document"/> to render.</param>
        /// <param name="width">The width of the document. If this is <see langword="null" />, the width of the largest page is used.</param>
        /// <param name="height">The height of the document. If this is <see langword="null" />, the height of the largest page is used.</param>
        /// <param name="background">The background colour of the document. If this is <see langword="null" />, a transparent background is used.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered graphics objects.</returns>
        public static SKMultiLayerRenderCanvas PaintToSKCanvas(this Document document, double? width = null, double? height = null, Colour? background = null, AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            return new SKMultiLayerRenderCanvas((from el in document.Pages select el.CopyToSKRenderContext(textOption, filterOption)).ToList(), (from el in document.Pages select SKRenderAction.TransformAction(SKMatrix.Identity)).ToList(), background ?? Colour.FromRgba(0, 0, 0, 0), width ?? (from el in document.Pages select el.Width).Max(), height ?? (from el in document.Pages select el.Height).Max());
        }

        /// <summary>
        /// Render a <see cref="Document"/> to an <see cref="Avalonia.Controls.Canvas"/> using the
```

```
SkiaSharp renderer. Each page corresponds to a layer in the image.
        /// </summary>
        /// <param name="document">The <see cref="Document"/> to render.</param>
        /// <param name="taggedActions">A Dictionary containing the actions that will be performed on items with the corresponding tag.
        /// These should be functions that accept one parameter of type <see cref="SKRenderAction"/> and return an <see cref="IEnumerable{SKRenderAction}"/> of the render actions that will actually be added to the plot.</param>
        /// <param name="removeTaggedActionsAfterExecution">Whether the actions should be removed from <paramref name="taggedActions"/> after their execution. Set to false if the same action should be performed on multiple items with the same tag.</param>
        /// <param name="width">The width of the document. If this is <see langword="null" />, the width of the largest page is used.</param>
        /// <param name="height">The height of the document. If this is <see langword="null" />, the height of the largest page is used.</param>
        /// <param name="background">The background colour of the document. If this is <see langword="null" />, a transparent background is used.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered graphics objects.</returns>
        public static SKMultiLayerRenderCanvas PaintToSKCanvas(this Document document, Dictionary<string, Func<SKRenderAction, IEnumerable<SKRenderAction>> taggedActions, bool removeTaggedActionsAfterExecution = true, double? width = null, double? height = null, Colour? background = null, AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            return new SKMultiLayerRenderCanvas((from el in document.Pages select el.CopyToSKRenderContext(taggedActions, removeTaggedActionsAfterExecution, textOption, filterOption)).ToList(), (from el in document.Pages select SKRenderAction.TransformAction(SKMatrix.Identity)).ToList(), background ?? Colour.FromRgba(0, 0, 0, 0), width ?? (from el in document.Pages select el.Width).Max(), height ?? (from el in document.Pages select el.Height).Max());
        }

        /// <summary>
        /// Render a <see cref="Document"/> to an <see cref="Avalonia.Controls.Canvas"/> using the SkiaSharp renderer. Each page corresponds to a layer in the image.
        /// </summary>
        /// <param name="document">The <see cref="Document"/> to render.</param>
        /// <param name="taggedActions">A Dictionary containing the actions that will be performed on items with the corresponding tag.
        /// These should be functions that accept one parameter of type <see cref="SKRenderAction"/> and return an <see cref="IEnumerable{SKRenderAction}"/> of the render actions that will actually be added to the plot.</param>
        /// <param name="images">A dictionary that associates to each raster image path (or data URL) the image rendered as a <see cref="SKBitmap"/> and a boolean value indicating whether it should be drawn as "pixelated" or not. This will be populated automatically as the page is rendered.
        /// If you are rendering multiple <see cref="Page"/>s (or you are rendering the same page multiple times), it will be beneficial to keep a reference to this dictionary and pass it again on further rendering requests; otherwise, you can just pass an empty dictionary.</param>
        /// <param name="removeTaggedActionsAfterExecution">Whether the actions should be removed from <paramref name="taggedActions"/> after their execution. Set to false if the same action should be performed on multiple items with the same tag.</param>
        /// <param name="width">The width of the document. If this is <see langword="null" />, the width of the largest page is used.</param>
        /// <param name="height">The height of the document. If this is <see langword="null" />, the height of the largest page is used.</param>
        /// <param name="background">The background colour of the document. If this is <see langword="null" />, a transparent background is used.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered graphics objects.</returns>
        public static SKMultiLayerRenderCanvas PaintToSKCanvas(this Document document, Dictionary<string, Func<SKRenderAction, IEnumerable<SKRenderAction>> taggedActions, Dictionary<string, (SKBitmap, bool)> images, bool removeTaggedActionsAfterExecution = true, double? width = null, double? height = null, Colour? background = null, AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            return new SKMultiLayerRenderCanvas((from el in document.Pages select el.CopyToSKRenderContext(taggedActions, images, removeTaggedActionsAfterExecution, textOption, filterOption)).ToList(), (from el in document.Pages select SKRenderAction.TransformAction(SKMatrix.Identity)).ToList(), background ?? Colour.FromRgba(0, 0, 0, 0), width ?? (from el in document.Pages select el.Width).Max(), height ?? (from el in document.Pages select el.Height).Max());
        }

        /// <summary>
        /// Render a <see cref="Page"/> to an <see cref="Avalonia.Controls.Canvas"/> using the SkiaSharp renderer.
        /// </summary>
```

```
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered graphics objects.</returns>
        public static SKMultiLayerRenderCanvas PaintToSKCanvas(this Page page, AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            return new SKMultiLayerRenderCanvas(new List<SKRenderContext>() { page.CopyToSKRenderContext(textOption, filterOption) }, new List<SKRenderAction>() { SKRenderAction.TransformAction(SKMatrix.Identity) }, page.Background, page.Width, page.Height);
        }

        /// <summary>
        /// Render a <see cref="Page"/> to an <see cref="Avalonia.Controls.Canvas"/> using the SkiaSharpRenderer.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="taggedActions">A Dictionary containing the actions that will be performed on items with the corresponding tag.
        /// These should be functions that accept one parameter of type <see cref="SKRenderAction"/> and return an <see cref="IEnumerable{SKRenderAction}"/> of the render actions that will actually be added to the plot.</param>
        /// <param name="removeTaggedActionsAfterExecution">Whether the actions should be removed from <paramref name="taggedActions"/> after their execution. Set to false if the same action should be performed on multiple items with the same tag.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered graphics objects.</returns>
        public static SKMultiLayerRenderCanvas PaintToSKCanvas(this Page page, Dictionary<string, Func<SKRenderAction, IEnumerable<SKRenderAction>>> taggedActions, bool removeTaggedActionsAfterExecution = true, AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            return new SKMultiLayerRenderCanvas(new List<SKRenderContext>() { page.CopyToSKRenderContext(taggedActions, removeTaggedActionsAfterExecution, textOption, filterOption) }, new List<SKRenderAction>() { SKRenderAction.TransformAction(SKMatrix.Identity) }, page.Background, page.Width, page.Height);
        }

        /// <summary>
        /// Render a <see cref="Page"/> to an <see cref="Avalonia.Controls.Canvas"/> using the SkiaSharpRenderer.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="taggedActions">A Dictionary containing the actions that will be performed on items with the corresponding tag.
        /// These should be functions that accept one parameter of type <see cref="SKRenderAction"/> and return an <see cref="IEnumerable{SKRenderAction}"/> of the render actions that will actually be added to the plot.</param>
        /// <param name="images">A dictionary that associates to each raster image path (or data URL) the image rendered as a <see cref="SKBitmap"/> and a boolean value indicating whether it should be drawn as "pixelated" or not. This will be populated automatically as the page is rendered.
        /// If you are rendering multiple <see cref="Page"/>s (or you are rendering the same page multiple times), it will be beneficial to keep a reference to this dictionary and pass it again on further rendering requests; otherwise, you can just pass an empty dictionary.</param>
        /// <param name="removeTaggedActionsAfterExecution">Whether the actions should be removed from <paramref name="taggedActions"/> after their execution. Set to false if the same action should be performed on multiple items with the same tag.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="Avalonia.Controls.Canvas"/> containing the rendered graphics objects.</returns>
        public static SKMultiLayerRenderCanvas PaintToSKCanvas(this Page page, Dictionary<string, Func<SKRenderAction, IEnumerable<SKRenderAction>>> taggedActions, Dictionary<string, (SKBitmap, bool)> images, bool removeTaggedActionsAfterExecution = true, AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            return new SKMultiLayerRenderCanvas(new List<SKRenderContext>() { page.CopyToSKRenderContext(taggedActions, images, removeTaggedActionsAfterExecution, textOption, filterOption) }, new List<SKRenderAction>() { SKRenderAction.TransformAction(SKMatrix.Identity) }, page.Background, page.Width, page.Height);
        }

        /// <summary>
        /// Render a <see cref="Page"/> to a <see cref="SKRenderContext"/>. This can be drawn using the SkiaSharpRenderer by adding it to a <see cref="SKMultiLayerRenderCanvas"/>.
        /// </summary>
```

```
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>A <see cref="SKRenderContext"/> containing the rendered graphics objects.</returns>
        public static SKRenderContext CopyToSKRenderContext(this Page page, AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            return CopyToSKRenderContext(page, new Dictionary<string, Func<SKRenderAction, IEnumerable<SKRenderAction>>>(), new Dictionary<string, (SKBitmap, bool)>(), textOption: textOption, filterOption: filterOption);
        }

        /// <summary>
        /// Render a <see cref="Page"/> to a <see cref="SKRenderContext"/>. This can be drawn using the SkiaSharpRenderer by adding it to a <see cref="SKMultiLayerRenderCanvas"/>.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="taggedActions">A Dictionary containing the actions that will be performed on items with the corresponding tag.
        /// These should be functions that accept one parameter of type <see cref="SKRenderAction"/> and return an <see cref="IEnumerable{SKRenderAction}"/> of the render actions that will actually be added to the plot.</param>
        /// <param name="removeTaggedActionsAfterExecution">Whether the actions should be removed from <paramref name="taggedActions"/> after their execution. Set to false if the same action should be performed on multiple items with the same tag.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>A <see cref="SKRenderContext"/> containing the rendered graphics objects.</returns>
        public static SKRenderContext CopyToSKRenderContext(this Page page, Dictionary<string, Func<SKRenderAction, IEnumerable<SKRenderAction>>> taggedActions, bool removeTaggedActionsAfterExecution = true, AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            return CopyToSKRenderContext(page, taggedActions, new Dictionary<string, (SKBitmap, bool)>(), removeTaggedActionsAfterExecution, textOption, filterOption);
        }

        /// <summary>
        /// Render a <see cref="Page"/> to a <see cref="SKRenderContext"/>. This can be drawn using the SkiaSharpRenderer by adding it to a <see cref="SKMultiLayerRenderCanvas"/>.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="taggedActions">A Dictionary containing the actions that will be performed on items with the corresponding tag.
        /// These should be functions that accept one parameter of type <see cref="SKRenderAction"/> and return an <see cref="IEnumerable{SKRenderAction}"/> of the render actions that will actually be added to the plot.</param>
        /// <param name="images">A dictionary that associates to each raster image path (or data URL) the image rendered as a <see cref="SKBitmap"/> and a boolean value indicating whether it should be drawn as "pixelated" or not. This will be populated automatically as the page is rendered.
        /// If you are rendering multiple <see cref="Page"/>s (or you are rendering the same page multiple times), it will be beneficial to keep a reference to this dictionary and pass it again on further rendering requests; otherwise, you can just pass an empty dictionary.</param>
        /// <param name="removeTaggedActionsAfterExecution">Whether the actions should be removed from <paramref name="taggedActions"/> after their execution. Set to false if the same action should be performed on multiple items with the same tag.</param>
        /// <param name="textOption">Defines whether text items should be converted into paths when drawing.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>A <see cref="SKRenderContext"/> containing the rendered graphics objects.</returns>
        public static SKRenderContext CopyToSKRenderContext(this Page page, Dictionary<string, Func<SKRenderAction, IEnumerable<SKRenderAction>>> taggedActions, Dictionary<string, (SKBitmap, bool)> images, bool removeTaggedActionsAfterExecution = true, AvaloniaContextInterpreter.TextOptions textOption = AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary, FilterOption filterOption = default)
        {
            if (filterOption == null)
            {
                filterOption = FilterOption.Default;
            }

            SKRenderContextImpl tbr = new SKRenderContextImpl(page.Width, page.Height, removeTaggedActionsAfterExecution, textOption, images, filterOption)
            {
                TaggedActions = taggedActions
            };
            page.Graphics.CopyToIGraphicsContext(tbr);

```

```
            return tbr;
        }


        /// <summary>
        /// Rasterise a region of a <see cref="Graphics"/> object.
        /// </summary>
        /// <param name="graphics">The <see cref="Graphics"/> object that will be rasterised.</param>
        /// <param name="region">The region of the <paramref name="graphics"/> that will be rasterised.</param>
        /// <param name="scale">The scale at which the image will be rendered.</param>
        /// <param name="interpolate">Whether the resulting image should be interpolated or not when it is drawn on another <see cref="Graphics"/> surface.</param>
        /// <returns>A <see cref="RasterImage"/> containing the rasterised graphics.</returns>
        public static RasterImage Rasterise(this Graphics graphics, Rectangle region, double scale, bool interpolate)
        {
            Page pag = new Page(1, 1);
            pag.Graphics.DrawGraphics(0, 0, graphics);
            pag.Crop(region.Location, region.Size);

            //pag.PaintToSKCanvas().RenderAtResolution((int)Math.Round(region.Size.Width * scale), (int)Math.Round(region.Size.Height * scale), null);

            int width = (int)Math.Round(region.Size.Width * scale);
            int height = (int)Math.Round(region.Size.Height * scale);

            SKRenderContext ctx = pag.CopyToSKRenderContext();

            SKBitmap bitmap = new SKBitmap(width, height, SKColorType.Rgba8888, SKAlphaType.Unpremul);

            SKCanvas canvas = new SKCanvas(bitmap);

            canvas.Save();

            for (int i = 0; i < ctx.SKRenderActions.Count; i++)
            {
                if (ctx.SKRenderActions[i].ActionType == SKRenderAction.ActionTypes.Clip)
                {
                    canvas.ClipPath(ctx.SKRenderActions[i].Path, antialias: true);
                }
                else if (ctx.SKRenderActions[i].ActionType == SKRenderAction.ActionTypes.Restore)
                {
                    canvas.Restore();
                }
                else if (ctx.SKRenderActions[i].ActionType == SKRenderAction.ActionTypes.Save)
                {
                    canvas.Save();
                }
                else if (ctx.SKRenderActions[i].ActionType == SKRenderAction.ActionTypes.Transform)
                {
                    SKMatrix mat = ctx.SKRenderActions[i].Transform.Value;
                    canvas.Concat(ref mat);
                }
                else
                {
                    if (ctx.SKRenderActions[i].ActionType == SKRenderAction.ActionTypes.Path && ctx.SKRenderActions[i].Path != null)
                    {
                        canvas.DrawPath(ctx.SKRenderActions[i].Path, ctx.SKRenderActions[i].Paint);
                    }
                    else if (ctx.SKRenderActions[i].ActionType == SKRenderAction.ActionTypes.Text)
                    {
                        canvas.DrawText(ctx.SKRenderActions[i].Text, ctx.SKRenderActions[i].TextX, ctx.SKRenderActions[i].TextY, ctx.SKRenderActions[i].Font, ctx.SKRenderActions[i].Paint);
                    }
                    else if (ctx.SKRenderActions[i].ActionType == SKRenderAction.ActionTypes.RasterImage)
                    {
                        (SKBitmap image, bool interpolateIt) = ctx.Images[ctx.SKRenderActions[i].ImageId];

                        SKPaint paint;

                        if (!interpolateIt)
                        {
                            paint = null;
                        }
                        else
                        {
                            paint = new SKPaint() { FilterQuality = SKFilterQuality.Medium };
                        }

                        canvas.DrawBitmap(image, ctx.SKRenderActions[i].ImageSource.Value, ctx.SKRenderActions[i].ImageDestination.Value, paint);

                        paint?.Dispose();
```

```
                    }
                }
            }

            canvas.Restore();

            IntPtr pixels = bitmap.GetPixels(out IntPtr length);

            IntPtr tbrData = System.Runtime.InteropServices.Marshal.AllocHGlobal(length);
            GC.AddMemoryPressure((long)length);

            unsafe
            {
                Buffer.MemoryCopy((void*)pixels, (void*)tbrData, (long)length, (long)length);
            }

            canvas.Dispose();
            bitmap.Dispose();

            DisposableIntPtr disp = new DisposableIntPtr(tbrData);
            return new RasterImage(ref disp, width, height, true, interpolate);
        }
    }
}
```

8.6 Nimbus.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, version 3.
    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU General Public License for more details.
    You should have received a copy of the GNU General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

namespace VectSharp.Fonts
{
    /// <summary>
    /// Contains an <see cref="IFontLibrary"/> providing access to the Nimbus family of standard fonts (used e.g. by MuPDF).
    /// </summary>
    public class Nimbus
    {
        /// <summary>
        /// The font library.
        /// </summary>
        public static IFontLibrary Library { get; }

        static Nimbus()
        {
            Library = new SimpleFontLibrary(
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusRomNo9L-Reg.t
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Roman No9 L"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusRomNo9L-Med.t
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Roman No9 L"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusRomNo9L-RegIt
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Roman No9 L"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusRomNo9L-MedIt
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Roman No9 L"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusSanL-Reg.ttf
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Sans L"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusSanL-Bol.ttf
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Sans L"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusSanL-RegIta.t
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Sans L"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusSanL-BolIta.t
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Sans L"),
```

```
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusMono-Regular
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Mono"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusMono-Bold.tt
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Mono"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusMono-Oblique
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Mono"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.NimbusMono-BoldObl
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#Nimbus Mono"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.StandardSymbolsPS.t
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#StandardSymbolsPS"),
                new ResourceFontFamily(typeof(Nimbus).Assembly.GetManifestResourceStream("VectSharp.Fonts.Nimbus.Nimbus.D050000L.ttf"),
                "resm:VectSharp.Fonts.Nimbus.Nimbus.?assembly=VectSharp.Fonts.Nimbus#D050000L"));
        }
    }
}
```

8.7 HtmlTag.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.IO;
using System.Net;
using System.Text;
using System.Threading.Tasks;

namespace VectSharp.Markdown
{
    internal class HtmlTag
    {
        public Dictionary<string, string> Attributes { get; }
        public string Tag { get; }

        private HtmlTag(string tag, Dictionary<string, string> attributes)
        {
            this.Tag = tag;
            this.Attributes = attributes;
        }

        private static HtmlTag ParseTag(StringReader reader)
        {
            StringBuilder tagBuilder = new StringBuilder();

            int character = reader.Read();

            while (character >= 0 && (char)character != '<')
            {
                character = reader.Read();
            }

            if ((char)character == '<')
            {
                character = reader.Read();
            }

            while (character >= 0 && char.IsWhiteSpace((char)character))
            {
                character = reader.Read();
            }

            while (character >= 0 && !char.IsWhiteSpace((char)character) && (char)character != '>')
```

```
            {
                tagBuilder.Append((char)character);
                character = reader.Read();
            }

            string tag = tagBuilder.ToString();

            Dictionary<string, string> attributes = new Dictionary<string, string>();

            (string, string)? attribute = ReadAttribute(reader, ref character);

            while (attribute != null && character >= 0)
            {
                attributes[attribute.Value.Item1] = attribute.Value.Item2;
                attribute = ReadAttribute(reader, ref character);
            }

            return new HtmlTag(tag, attributes);
        }

        public static IEnumerable<HtmlTag> ParseTagsUntil(StringReader reader, string targetTag)
        {
            HtmlTag tag = ParseTag(reader);

            while (tag.Tag != targetTag && reader.Peek() >= 0)
            {
                if (tag.Tag.Equals("p", StringComparison.OrdinalIgnoreCase))
                {
                    foreach (HtmlTag nestedTag in ParseTagsUntil(reader, "/p"))
                    {
                        if (nestedTag.Tag != "/p")
                        {
                            foreach (KeyValuePair<string, string> kvp in tag.Attributes)
                            {
                                if (!nestedTag.Attributes.ContainsKey(kvp.Key))
                                {
                                    nestedTag.Attributes[kvp.Key] = kvp.Value;
                                }
                            }

                            yield return nestedTag;
                        }
                    }
                }
                else
                {
                    yield return tag;
                }


                tag = ParseTag(reader);
            }

            yield return tag;
        }

        public static IEnumerable<HtmlTag> Parse(string html)
        {
            using (StringReader reader = new StringReader(html))
            {
                while (reader.Peek() >= 0)
                {
                    HtmlTag tag = ParseTag(reader);

                    if (tag.Tag.Equals("p", StringComparison.OrdinalIgnoreCase))
                    {
                        foreach (HtmlTag nestedTag in ParseTagsUntil(reader, "/p"))
                        {
                            if (nestedTag.Tag != "/p")
                            {
                                foreach (KeyValuePair<string, string> kvp in tag.Attributes)
                                {
                                    if (!nestedTag.Attributes.ContainsKey(kvp.Key))
                                    {
                                        nestedTag.Attributes[kvp.Key] = kvp.Value;
                                    }
                                }

                                yield return nestedTag;
                            }
                        }
                    }
                    else
                    {
                        yield return tag;
                    }
                }
```

```
            }
        }

        private static (string, string)? ReadAttribute(StringReader reader, ref int character)
        {
            while (character >= 0 && char.IsWhiteSpace((char)character) && (char)character != '>')
            {
                character = reader.Read();
            }

            if ((char)character == '>')
            {
                return null;
            }
            else
            {
                StringBuilder attributeNameBuilder = new StringBuilder();

                while (character >= 0 && !char.IsWhiteSpace((char)character) && (char)character != '>' && (char)character != '=')
                {
                    attributeNameBuilder.Append((char)character);
                    character = reader.Read();
                }

                string attributeName = attributeNameBuilder.ToString();

                while (character >= 0 && char.IsWhiteSpace((char)character) && (char)character != '>' && (char)character != '=')
                {
                    character = reader.Read();
                }

                if ((char)character == '=')
                {
                    character = reader.Read();

                    while (character >= 0 && char.IsWhiteSpace((char)character) && (char)character != '>')
                    {
                        character = reader.Read();
                    }

                    if ((char)character == '>')
                    {
                        return (attributeName, null);
                    }
                    else
                    {
                        bool quoted = (char)character == '"' || (char)character == '\';

                        if (quoted)
                        {
                            char quoteChar = (char)character;

                            character = reader.Read();

                            StringBuilder attributeValueBuilder = new StringBuilder();

                            bool isEscaped = (char)character == '\\';

                            while (character >= 0 && ((char)character != quoteChar || isEscaped))
                            {
                                attributeValueBuilder.Append((char)character);
                                character = reader.Read();
                                isEscaped = (char)character == '\\' && !isEscaped;
                            }

                            string attributeValue = attributeValueBuilder.ToString();

                            return (attributeName, attributeValue);
                        }
                        else
                        {
                            StringBuilder attributeValueBuilder = new StringBuilder();

                            while (character >= 0 && !char.IsWhiteSpace((char)character) && (char)character != '>' && (char)character != '=')
                            {
                                attributeValueBuilder.Append((char)character);
                                character = reader.Read();
                            }

                            string attributeValue = attributeValueBuilder.ToString();

                            return (attributeName, attributeValue);
                        }
```

```
                    }
                }
                else
                {
                    return (attributeName, null);
                }
            }
        }

    }

    /// <summary>
    /// Contains utilities to resolve absolute and relative URIs.
    /// </summary>
    public static class HTTPUtils
    {
        /// <summary>
        /// Determines whether every file that is downloaded should be logged to the standard error stream.
        /// </summary>
        public static bool LogDownloads { get; set; } = true;

        /// <summary>
        /// Resolves an image Uri, by downloading the image file if necessary. It also takes care of ensuring that the file extension matches the format of the file.
        /// </summary>
        /// <param name="uri">The address of the image.</param>
        /// <param name="baseUriString">The base uri to use for relative uris.</param>
        /// <returns>A tuple containing the local path of the image file (either the original image, or a local copy of a remote file) and a boolean value indicating whether the image was fetched from a remote location and should be deleted after the program is done with it.</returns>
        public static (string path, bool wasDownloaded) ResolveImageURI(string uri, string baseUriString)
        {
            if (uri.StartsWith("data:"))
            {
                string tempFile = Path.GetTempFileName();
                if (File.Exists(tempFile))
                {
                    File.Delete(tempFile);
                }

                Directory.CreateDirectory(tempFile);

                if (!uri.StartsWith(""))
                {
                    VectSharp.Page pag = VectSharp.SVG.Parser.ParseImageURI(uri, true);
                    VectSharp.SVG.SVGContextInterpreter.SaveAsSVG(pag, Path.Combine(tempFile, "temp.svg"));
                }
                else
                {
                    string base64 = uri.Substring("".Length);

                    File.WriteAllBytes(Path.Combine(tempFile, "temp.svg"), Convert.FromBase64String(base64));
                }

                return (Path.Combine(tempFile, "temp.svg"), true);
            }
            else if (File.Exists(Path.Combine(baseUriString, uri)))
            {
                return (Path.Combine(baseUriString, uri), false);
            }
            else if (File.Exists(uri))
            {
                return (uri, false);
            }
            else
            {
                Uri absoluteUri;
                bool validUri;

                if (Uri.TryCreate(baseUriString, UriKind.Absolute, out Uri baseUri))
                {
                    validUri = Uri.TryCreate(baseUri, uri, out absoluteUri);
                }
                else
                {
                    validUri = Uri.TryCreate(uri, UriKind.Absolute, out absoluteUri);
                }

                if (validUri)
                {
                    string tempFile = Path.GetTempFileName();
                    File.Delete(tempFile);
                    Directory.CreateDirectory(tempFile);
```

```

                    string fileDest = Path.Combine(tempFile, Path.GetFileName(absoluteUri.LocalPath));

                    try
                    {
                        if (LogDownloads)
                        {
                            Console.Error.WriteLine();
                            Console.Error.Write("Downloading {0}...", absoluteUri);
                        }

                        using (WebClient client = new WebClient())
                        {
                            client.DownloadFile(absoluteUri, fileDest);
                        }

                        if (LogDownloads)
                        {
                            Console.Error.WriteLine(" Done.");
                        }

                        string newName = FixFileExtensionBasedOnContent(fileDest);

                        File.Move(fileDest, newName);
                        fileDest = newName;

                        return (fileDest, true);
                    }
                    catch (Exception ex)
                    {
                        if (LogDownloads)
                        {
                            Console.Error.WriteLine(" Failed!");
                            Console.Error.WriteLine(ex.Message);
                        }

                        Directory.Delete(tempFile, true);
                        return (null, false);
                    }
                }
                else
                {
                    return (null, false);
                }
            }
        }

        private static string FixFileExtensionBasedOnContent(string fileName)
        {
            using (FileStream fileStream = File.OpenRead(fileName))
            {
                bool isSvg = false;

                try
                {
                    using (var xmlReader = System.Xml.XmlReader.Create(fileStream))
                    {
                        isSvg = xmlReader.MoveToContent() == System.Xml.XmlNodeType.Element &&
"svg".Equals(xmlReader.Name, StringComparison.OrdinalIgnoreCase);
                    }
                }
                catch
                {
                    isSvg = false;
                }

                if (isSvg)
                {
                    return fileName + ".svg";
                }
                else
                {
                    fileStream.Seek(0, SeekOrigin.Begin);
                    byte[] header = new byte[8];

                    for (int i = 0; i < header.Length; i++)
                    {
                        header[i] = (byte)fileStream.ReadByte();
                    }

                    if (header[0] == 0x42 && header[1] == 0x4D)
                    {
                        return fileName + ".bmp";
                    }
                    else if (header[0] == 0x47 && header[1] == 0x49 && header[2] == 0x46 && header[3]
== 0x38)
                    {
```

```
                        return fileName + ".gif";
                    }
                    else if (header[0] == 0xFF && header[1] == 0xD8 && header[2] == 0xFF && (header[3] == 0xDB || header[3] == 0xE0 || header[3] == 0xEE || header[3] == 0xE1))
                    {
                        return fileName + ".jpg";
                    }
                    else if (header[0] == 0x25 && header[1] == 0x50 && header[2] == 0x44 && header[3] == 0x46 && header[4] == 0x2D)
                    {
                        return fileName + ".pdf";
                    }
                    else if (header[0] == 0x89 && header[1] == 0x50 && header[2] == 0x4E && header[3] == 0x47 && header[4] == 0x0D && header[5] == 0x0A && header[6] == 0x1A && header[7] == 0x0A)
                    {
                        return fileName + ".png";
                    }
                    else if ((header[0] == 0x49 && header[1] == 0x49 && header[2] == 0x2A && header[3] == 0x00) || (header[0] == 0x4D && header[1] == 0x4D && header[2] == 0x00 && header[3] == 0x2A))
                    {
                        return fileName + ".tif";
                    }
                    else
                    {
                        return fileName;
                    }
                }
            }
        }
    }
}
```

8.8 Line.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Text;

namespace VectSharp.Markdown
{
    internal abstract class LineFragment
    {
        public abstract void Translate(double deltaX, double deltaY);
        public string Tag { get; protected set; }
    }

    internal class TextFragment : LineFragment
    {
        public string Text { get; }
        public Point Position { get; private set; }
        public Font Font { get; }
        public Colour Colour { get; }

        public TextFragment(Point position, string text, Font font, Colour colour, string tag)
        {
            this.Position = position;
            this.Text = text;
            this.Font = font;
            this.Colour = colour;
            this.Tag = tag;
        }

        public override void Translate(double deltaX, double deltaY)
        {
            this.Position = new Point(this.Position.X + deltaX, this.Position.Y + deltaY);
        }
```

```
    }

    internal class UnderlineFragment : LineFragment
    {
        public Point Start { get; private set; }
        public Point End { get; private set; }
        public Colour Colour { get; }
        public double Thickness { get; }

        public UnderlineFragment(Point start, Point end, Colour colour, double thickness, string tag)
        {
            this.Start = start;
            this.End = end;
            this.Colour = colour;
            this.Thickness = thickness;
            this.Tag = tag;
        }

        public override void Translate(double deltaX, double deltaY)
        {
            this.Start = new Point(this.Start.X + deltaX, this.Start.Y + deltaY);
            this.End = new Point(this.End.X + deltaX, this.End.Y + deltaY);
        }
    }

    internal class RectangleFragment : LineFragment
    {
        public Point TopLeft { get; private set; }
        public Size Size { get; }
        public Colour Colour { get; }

        public RectangleFragment(Point topLeft, Size size, Colour colour, string tag)
        {
            this.TopLeft = topLeft;
            this.Size = size;
            this.Colour = colour;
            this.Tag = tag;
        }

        public override void Translate(double deltaX, double deltaY)
        {
            this.TopLeft = new Point(this.TopLeft.X + deltaX, this.TopLeft.Y + deltaY);
        }
    }

    internal class GraphicsFragment : LineFragment
    {
        public Point Origin { get; private set; }
        public Graphics Graphics { get; }
        public double Ascent { get; }

        public GraphicsFragment(Point origin, Graphics graphics, double ascent)
        {
            this.Origin = origin;
            this.Graphics = graphics;
            this.Ascent = ascent;
        }

        public override void Translate(double deltaX, double deltaY)
        {
            this.Origin = new Point(this.Origin.X + deltaX, this.Origin.Y + deltaY);
        }
    }

    internal class Line
    {
        public List<LineFragment> Fragments { get; }
        public double InitialAscent { get; }

        public Line(double initialAscent)
        {
            this.InitialAscent = initialAscent;
            this.Fragments = new List<LineFragment>();
        }

        public void Render(ref Graphics graphics, ref MarkdownContext context,
MarkdownRenderer.NewPageAction newPageAction, double pageMaxY)
        {
            double deltaY = 0;
            double maxY = 0;

            foreach (LineFragment fragment in this.Fragments)
            {
                if (fragment is TextFragment text)
                {
                    deltaY = Math.Max(deltaY, text.Font.Ascent - this.InitialAscent);
                    maxY = Math.Max(maxY, text.Position.Y - text.Font.Descent);
```

```
                }
                else if (fragment is UnderlineFragment underline)
                {
                    maxY = Math.Max(maxY, Math.Max(underline.Start.Y, underline.End.Y));
                }
                else if (fragment is RectangleFragment rectangle)
                {
                    maxY = Math.Max(maxY, rectangle.TopLeft.Y + rectangle.Size.Height);
                }
                else if (fragment is GraphicsFragment graphicsFragment)
                {
                    deltaY = Math.Max(deltaY, graphicsFragment.Ascent - this.InitialAscent);
                    maxY = Math.Max(maxY, graphicsFragment.Origin.Y);
                }
            }

            maxY += deltaY;

            if (maxY > pageMaxY)
            {
                double currCursY = context.Cursor.Y;

                newPageAction(ref context, ref graphics);

                double currDelta = deltaY;

                context.Cursor = new Point(context.Cursor.X, context.Cursor.Y + this.InitialAscent + currDelta);

                deltaY -= currCursY - context.Cursor.Y + currDelta;

                context.Cursor = new Point(context.Cursor.X, context.Cursor.Y);
            }
            else
            {
                context.Cursor = new Point(context.Cursor.X, context.Cursor.Y + deltaY);
            }


            for (int i = 0; i < this.Fragments.Count; i++)
            {
                LineFragment fragment = this.Fragments[i];
                if (fragment is TextFragment text)
                {
                    Size size = text.Font.MeasureText(text.Text);
                    context.BottomRight = new Point(Math.Max(context.BottomRight.X, size.Width + text.Position.X + context.Translation.X), Math.Max(context.BottomRight.Y, text.Position.Y + size.Height + deltaY + context.Translation.Y));
                    graphics.FillText(text.Position.X, text.Position.Y + deltaY, text.Text, text.Font, text.Colour, TextBaselines.Baseline, tag: fragment.Tag);
                }
                else if (fragment is UnderlineFragment underline)
                {
                    graphics.StrokePath(new GraphicsPath().MoveTo(underline.Start.X, underline.Start.Y + deltaY).LineTo(underline.End.X, underline.End.Y + deltaY), underline.Colour, underline.Thickness, tag: fragment.Tag);
                }
                else if (fragment is RectangleFragment rectangle)
                {
                    graphics.FillRectangle(rectangle.TopLeft.X, rectangle.TopLeft.Y + deltaY, rectangle.Size.Width, rectangle.Size.Height, rectangle.Colour, tag: fragment.Tag);
                }
                else if (fragment is GraphicsFragment graphicsFragment)
                {
                    graphics.DrawGraphics(graphicsFragment.Origin.X, graphicsFragment.Origin.Y + deltaY, graphicsFragment.Graphics);
                }
            }
        }
    }
}
```

8.9 MarkdownContext.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
```

```
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Text;

namespace VectSharp.Markdown
{
    internal class MarkdownContext
    {
        public Font Font { get; set; }

        private Point cursor;
        public Point Cursor
        {
            get
            {
                return cursor;
            }

            set
            {
                cursor = value;

                double maxX = Math.Max(BottomRight.X, value.X + Translation.X);
                double maxY = Math.Max(BottomRight.Y, value.Y + Translation.Y);

                this.BottomRight = new Point(maxX, maxY);
            }
        }

        public Colour Colour { get; set; }
        public bool Underline { get; set; }
        public bool StrikeThrough { get; set; }

        private Point translation;
        public Point Translation
        {
            get
            {
                return translation;
            }

            set
            {
                translation = value;

                double maxX = Math.Max(BottomRight.X, cursor.X + value.X);
                double maxY = Math.Max(BottomRight.Y, cursor.Y + value.Y);

                this.BottomRight = new Point(maxX, maxY);
            }
        }

        public Point MarginBottomRight { get; set; }

        public Line CurrentLine { get; set; }
        public int ListDepth { get; set; }
        public List<(double MaxX, double MinY, double MaxY)> ForbiddenAreasRight { get; set; }
        public List<(double MinX, double MinY, double MaxY)> ForbiddenAreasLeft { get; set; }
        public Page CurrentPage { get; set; }
        public Point BottomRight { get; set; }
        public string Tag { get; set; } = null;
        public Dictionary<string, string> LinkDestinations { get; set; } = new Dictionary<string,
    string>();
        public Dictionary<string, string> InternalAnchors { get; set; } = new Dictionary<string,
    string>();

        public MarkdownContext()
        {
            this.ForbiddenAreasRight = new List<(double MaxX, double MinY, double MaxY)>();
            this.ForbiddenAreasLeft = new List<(double MinX, double MinY, double MaxY)>();
        }

        public MarkdownContext Clone()
        {
            return new MarkdownContext()
            {
                Font = this.Font,
                Cursor = this.Cursor,
                Colour = this.Colour,
```

```
                Underline = this.Underline,
                StrikeThrough = this.StrikeThrough,
                Translation = this.Translation,
                CurrentLine = this.CurrentLine,
                ListDepth = this.ListDepth,
                ForbiddenAreasRight = this.ForbiddenAreasRight,
                ForbiddenAreasLeft = this.ForbiddenAreasLeft,
                CurrentPage = this.CurrentPage,
                Tag = this.Tag,
                LinkDestinations = this.LinkDestinations,
                InternalAnchors = this.InternalAnchors,
                MarginBottomRight = this.MarginBottomRight
            };
        }

        public double GetMaxX(double y, double pageMaxX)
        {
            y = y + Translation.Y;

            double maxX = pageMaxX;

            foreach ((double MaxX, double MinY, double MaxY) in ForbiddenAreasRight)
            {
                if (MinY <= y && MaxY >= y)
                {
                    maxX = Math.Min(maxX, MaxX - this.Translation.X);
                }
            }

            return maxX;
        }

        public double GetMaxX(double y0, double y1, double pageMaxX)
        {
            y0 = y0 + Translation.Y;
            y1 = y1 + Translation.Y;

            double maxX = pageMaxX;

            foreach ((double MaxX, double MinY, double MaxY) in ForbiddenAreasRight)
            {
                if (!((MinY < y0 && MaxY < y0) || (MinY > y1 && MaxY > y1)))
                {
                    maxX = Math.Min(maxX, MaxX - this.Translation.X);
                }
            }

            return maxX;
        }

        public double GetMinX(double y)
        {
            y = y + Translation.Y;

            double minX = 0;

            foreach ((double MinX, double MinY, double MaxY) in ForbiddenAreasLeft)
            {
                if (MinY <= y && MaxY >= y)
                {
                    minX = Math.Max(minX, MinX - Translation.X);
                }
            }

            return minX;
        }

        public double GetMinX(double y0, double y1)
        {

            y0 = y0 + Translation.Y;
            y1 = y1 + Translation.Y;

            double minX = 0;

            foreach ((double MinX, double MinY, double MaxY) in ForbiddenAreasLeft)
            {
                if (!((MinY < y0 && MaxY < y0) || (MinY > y1 && MaxY > y1)))
                {
                    minX = Math.Max(minX, MinX - Translation.X);
                }
            }

            return minX;
        }
    }

```

```
    /// <summary>
    /// Represents the margins of a page.
    /// </summary>
    public class Margins
    {
        /// <summary>
        /// The left margin.
        /// </summary>
        public double Left { get; }

        /// <summary>
        /// The right margin.
        /// </summary>
        public double Right { get; }

        /// <summary>
        /// The top margin.
        /// </summary>
        public double Top { get; }

        /// <summary>
        /// The bottom margin.
        /// </summary>
        public double Bottom { get; }

        /// <summary>
        /// Creates a new <see cref="Margins"/> instance.
        /// </summary>
        /// <param name="left">The left margin.</param>
        /// <param name="top">The top margin.</param>
        /// <param name="right">The right margin.</param>
        /// <param name="bottom">The bottom margin.</param>
        public Margins(double left, double top, double right, double bottom)
        {
            this.Left = left;
            this.Right = right;
            this.Top = top;
            this.Bottom = bottom;
        }
    }
}
```

8.10 MarkdownRenderer.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using VectSharp.Markdown.CSharpMath.VectSharp;
using Markdig;
using Markdig.Extensions;
using Markdig.Extensions.Tables;
using Markdig.Helpers;
using Markdig.Renderers.Html;
using Markdig.Syntax;
using Markdig.Syntax.Inlines;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

namespace VectSharp.Markdown
{
    /// <summary>
    /// Renders Markdown documents into VectSharp graphics objects.
    /// </summary>
    public class MarkdownRenderer
    {
        /// <summary>
```

```
        /// The base font size to use when rendering the document. This will be the size of regular elements, and the size of header elements will be expressed as a multiple of this.
        /// </summary>
        public double BaseFontSize { get; set; } = 9.71424;

        /// <summary>
        /// Scaling factor for the font size to use when rendering math.
        /// </summary>
        public double MathFontScalingFactor { get; set; } = 0.85;

        /// <summary>
        /// The font size for elements at each header level. The values in this array will be multiplied by the <see cref="BaseFontSize"/>.
        /// </summary>
        public double[] HeaderFontSizeMultipliers { get; } = new double[]
        {
            28 / 12.0, 22 / 12.0, 16 / 12.0, 14 / 12.0, 13 / 12.0, 12 / 12.0
        };

        /// <summary>
        /// The thickness of the separator line after a header of each level. A value of 0 disables the line after headers of that level.
        /// </summary>
        public double[] HeaderLineThicknesses { get; } = new double[] { 1, 1, 0, 0, 0, 0 };

        /// <summary>
        /// The thickness of thematic break lines.
        /// </summary>
        public double ThematicBreakThickness { get; set; } = 2;

        /// <summary>
        /// The font family for regular text.
        /// </summary>
        public FontFamily RegularFontFamily { get; set; } = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica);

        /// <summary>
        /// The font family for bold text.
        /// </summary>
        public FontFamily BoldFontFamily { get; set; } = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBold);

        /// <summary>
        /// The font family for italic text.
        /// </summary>
        public FontFamily ItalicFontFamily { get; set; } = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaOblique);

        /// <summary>
        /// The font family for bold italic text.
        /// </summary>
        public FontFamily BoldItalicFontFamily { get; set; } = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBoldOblique);

        /// <summary>
        /// The font family for code elements.
        /// </summary>
        public FontFamily CodeFont { get; set; } = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Courier);

        /// <summary>
        /// The font family for bold code elements.
        /// </summary>
        public FontFamily CodeFontBold { get; set; } = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBold);

        /// <summary>
        /// The font family for italic code elements.
        /// </summary>
        public FontFamily CodeFontItalic { get; set; } = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierOblique);

        /// <summary>
        /// The font family for bold italic code elements.
        /// </summary>
        public FontFamily CodeFontBoldItalic { get; set; } = FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBoldOblique);

        /// <summary>
        /// The thickness of underlines. This value will be multiplied by the font size of the element being underlined.
        /// </summary>
        public double UnderlineThickness { get; set; } = 0.075;

        /// <summary>
        /// The thickness of underlines for bold text. This value will be multiplied by the font size of the element being underlined.
```

```
        /// </summary>
        public double BoldUnderlineThickness { get; set; } = 0.15;

        /// <summary>
        /// The margins of the page.
        /// </summary>
        public Margins Margins { get; set; } = new Margins(55, 55, 55, 55);

        /// <summary>
        /// The margins for table cells.
        /// </summary>
        public Margins TableCellMargins { get; set; } = new Margins(5, 0, 5, 0);

        /// <summary>
        /// Defines the options for the vertical alignment of table cells.
        /// </summary>
        public enum VerticalAlignment
        {
            /// <summary>
            /// Table cells will be aligned at the top of their row.
            /// </summary>
            Top,

            /// <summary>
            /// Table cells will be aligned in the middle of their row.
            /// </summary>
            Middle,

            /// <summary>
            /// Table cells will be aligned at the bottom of their row.
            /// </summary>
            Bottom
        }

        /// <summary>
        /// The vertical alignment of table cells.
        /// </summary>
        public VerticalAlignment TableVAlign { get; set; } = VerticalAlignment.Middle;

        /// <summary>
        /// The size of the page.
        /// </summary>
        public Size PageSize { get; set; } = new Size(595, 842);

        /// <summary>
        /// The space before each text paragraph. This value will be multiplied by the <see cref="BaseFontSize"/>.
        /// </summary>
        public double SpaceBeforeParagaph { get; set; } = 0;

        /// <summary>
        /// The space after each text paragraph. This value will be multiplied by the <see cref="BaseFontSize"/>.
        /// </summary>
        public double SpaceAfterParagraph { get; set; } = 0.75;

        /// <summary>
        /// The space after each line of text. This value will be multiplied by the <see cref="BaseFontSize"/>.
        /// </summary>
        public double SpaceAfterLine { get; set; } = 0.25;

        /// <summary>
        /// The space before each heading. This value will be multiplied by the font size of the heading.
        /// </summary>
        public double SpaceBeforeHeading { get; set; } = 0.25;

        /// <summary>
        /// The space after each heading. This value will be multiplied by the font size of the heading.
        /// </summary>
        public double SpaceAfterHeading { get; set; } = 0.25;

        /// <summary>
        /// The margin at the left and right of code inlines. This value will be multiplied by the current font size.
        /// </summary>
        public double CodeInlineMargin { get; set; } = 0.25;

        /// <summary>
        /// The indentation width used for list items.
        /// </summary>
        public double IndentWidth { get; set; } = 40;

        /// <summary>
        /// The indentation width used for block quotes.
```

```
        /// </summary>
        public double QuoteBlockIndentWidth { get; set; } = 30;

        /// <summary>
        /// The thickness of the bar to the left of block quotes.
        /// </summary>
        public double QuoteBlockBarWidth { get; set; } = 5;

        /// <summary>
        /// The font size for subscripts and superscripts. This value will be multiplied by the current font size.
        /// </summary>
        public double SubSuperscriptFontSize { get; set; } = 0.7;

        /// <summary>
        /// The upwards shift in the baseline for superscript elements. This value will be multiplied by the current font size.
        /// </summary>
        public double SuperscriptShift { get; set; } = 0.33;

        /// <summary>
        /// The downwards shift in the baseline for subscript elements. This value will be multiplied by the current font size.
        /// </summary>
        public double SubscriptShift { get; set; } = 0.14;

        /// <summary>
        /// The base uri for resolving relative image addresses.
        /// </summary>
        public string BaseImageUri { get; set; } = "";

        /// <summary>
        /// A method used to resolve (possibly remote) image uris into local file paths. The first argument of the method should be the image uri and the second argument the base uri used to resolve relative links. The method should return a tuple containing the path of the local file and a boolean value indicating whether the file has been fetched from a remote location and should be deleted after the program has finished using it.
        /// </summary>
        public Func<string, string, (string, bool)> ImageUriResolver { get; set; } = HTTPUtils.ResolveImageURI;

        /// <summary>
        /// The base uri for resolving links.
        /// </summary>
        public Uri BaseLinkUri { get; set; } = new Uri("about:blank");

        /// <summary>
        /// A method used to resolve link addresses. The argument of the method should be the absolute link, and the method should return the resolved address. This can be used to "redirect" links to a different target.
        /// </summary>
        public Func<string, string> LinkUriResolver { get; set; } = a => a;

        /// <summary>
        /// A method used to a load raster image from a local file. The argument of the method should be the path of a local image file, and the method should return a RasterImage representing that file. For example, this can be achieved using the <c>RasterImageFile</c> class from the <c>VectSharp.MuPDFUtils</c> package. If this is <see langword="null" />, only SVG images will be included in the document.
        /// </summary>
        public Func<string, RasterImage> RasterImageLoader { get; set; } = null;

        /// <summary>
        /// The size of images (as defined in the image's width and height attributes) will be multiplied by this value to determine the actual size of the image on the page. This has no effect on images without a width or height attribute.
        /// </summary>
        public double ImageUnitMultiplier { get; set; } = 0.60714;

        /// <summary>
        /// The size of images will be multiplied by this value to determine the actual size of the image on the page. For images that have a width or height attribute, this will be applied in addition to the <see cref="ImageUnitMultiplier"/>. For images without width and height, only this multiplier will be applied.
        /// </summary>
        public double ImageMultiplier { get; set; } = 1;

        /// <summary>
        /// The margin on the right of left-aligned images and on the left of right-aligned images.
        /// </summary>
        public double ImageSideMargin { get; set; } = 10;

        /// <summary>
        /// Images will be allowed to extend into the page bottom margin area by this amount before triggering a page break. This should be smaller than the bottom margin, otherwise images risk being cut off by the page boundary.
        /// </summary>
```

```
        public double ImageMarginTolerance { get; set; } = 25;

        /// <summary>
        /// A method used for syntax highlighting. The first argument should be the source code to highlight, while the second parameter is the name of the language to use for the highlight. The method should return a list of lists of <see cref="FormattedString"/>s, with each list of <see cref="FormattedString"/>s representing a line. For each code block, if the method returns <see langword="null" />, no syntax highlighting is used.
        /// </summary>
        public Func<string, string, List<List<FormattedString>> SyntaxHighlighter { get; set; } = VectSharp.Markdown.SyntaxHighlighter.GetSyntaxHighlightedLines;

        /// <summary>
        /// Bullet points used for unordered lists. Each element of this list corresponds to the bullet for each level of list indentation. If the list indentation is greater than the number of elements in this list, the bullet points will be reused cyclically.
        /// Each element of this list is a method taking two arguments: the first is the <see cref="Graphics"/> object on which the bullet point should be drawn, while the second is the colour in which it should be painted. The method should draw the bullet point centered around the origin. The size of the bullet point will be multiplied by the current font size.
        /// </summary>
        public List<Action<Graphics, Colour> Bullets { get; } = new List<Action<Graphics, Colour>()
        {
            (graphics, colour) =>
            {
                graphics.FillPath(new GraphicsPath().Arc(-0.5, 0, 0.25, 0, 2 * Math.PI), colour);
            },

            (graphics, colour) =>
            {
                graphics.StrokePath(new GraphicsPath().Arc(-0.5, 0, 0.25, 0, 2 * Math.PI), colour, 0.1);
            },

            (graphics, colour) =>
            {
                graphics.StrokeRectangle(-0.75, -0.25, 0.5, 0.5, colour, 0.1);
            },
        };

        /// <summary>
        /// The foreground colour for text elements.
        /// </summary>
        public Colour ForegroundColour { get; set; } = Colours.Black;

        /// <summary>
        /// The background colour for the page.
        /// </summary>
        public Colour BackgroundColour { get; set; } = Colours.White;

        /// <summary>
        /// The colour of the line below headers.
        /// </summary>
        public Colour HeaderLineColour { get; set; } = Colour.FromRgb(180, 180, 180);

        /// <summary>
        /// The colour for thematic break lines.
        /// </summary>
        public Colour ThematicBreakLineColour { get; set; } = Colour.FromRgb(180, 180, 200);

        /// <summary>
        /// The colour for hypertext links-
        /// </summary>
        public Colour LinkColour { get; set; } = Colour.FromRgb(25, 140, 191);

        /// <summary>
        /// The background colour for code inlines.
        /// </summary>
        public Colour CodeInlineBackgroundColour { get; set; } = Colour.FromRgb(240, 240, 240);

        /// <summary>
        /// The background colour for code blocks.
        /// </summary>
        public Colour CodeBlockBackgroundColour { get; set; } = Colour.FromRgb(240, 240, 245);

        /// <summary>
        /// The colour for the bar to the left of block quotes.
        /// </summary>
        public Colour QuoteBlockBarColour { get; set; } = Colour.FromRgb(75, 152, 220);

        /// <summary>
        /// The background colour for block quotes.
        /// </summary>
        public Colour QuoteBlockBackgroundColour { get; set; } = Colour.FromRgb(240, 240, 255);

        /// <summary>
        /// The colour for text that has been styled as "inserted".
```

```
        /// </summary>
        public Colour InsertedColour { get; set; } = Colour.FromRgb(0, 158, 115);

        /// <summary>
        /// The colour for text that has been styled as "marked".
        /// </summary>
        public Colour MarkedColour { get; set; } = Colour.FromRgb(213, 94, 0);

        /// <summary>
        /// The colour for the line separating the table header row from normal rows.
        /// </summary>
        public Colour TableHeaderRowSeparatorColour { get; set; } = Colours.Black;

        /// <summary>
        /// The colour for lines separating table rows from each other.
        /// </summary>
        public Colour TableRowSeparatorColour { get; set; } = Colour.FromRgb(180, 180, 180);

        /// <summary>
        /// The thickness of the line separating the table header row from normal rows.
        /// </summary>
        public double TableHeaderRowSeparatorThickness { get; set; } = 2;

        /// <summary>
        /// The thickness of lines separating table rows from each other.
        /// </summary>
        public double TableHeaderSeparatorThickness { get; set; } = 1;

        /// <summary>
        /// The bullet used for unchecked task list items.
        /// </summary>
        public Graphics TaskListUncheckedBullet { get; set; } = new Func<Graphics>(() =>
        {
            Graphics tbr = new Graphics();

            GraphicsPath checkboxPath = new GraphicsPath().MoveTo(-0.7, -0.4).LineTo(-0.3, -0.4).Arc(-0.3, -0.2, 0.2, 3 * Math.PI / 2, 2 * Math.PI).LineTo(-0.1, 0.2).Arc(-0.3, 0.2, 0.2, 0, Math.PI / 2).LineTo(-0.7, 0.4).Arc(-0.7, 0.2, 0.2, Math.PI / 2, Math.PI).LineTo(-0.9, -0.2).Arc(-0.7, -0.2, 0.2, Math.PI, 3 * Math.PI / 2).Close();
            tbr.FillPath(checkboxPath, Colour.FromRgb(240, 246, 249));
            tbr.StrokePath(checkboxPath, Colour.FromRgb(0, 114, 178), 0.075);

            return tbr;
        })();


        /// <summary>
        /// The bullet used for checked task list items.
        /// </summary>
        public Graphics TaskListCheckedBullet { get; set; } = new Func<Graphics>(() =>
        {
            Graphics tbr = new Graphics();

            GraphicsPath checkboxPath = new GraphicsPath().MoveTo(-0.7, -0.4).LineTo(-0.3, -0.4).Arc(-0.3, -0.2, 0.2, 3 * Math.PI / 2, 2 * Math.PI).LineTo(-0.1, 0.2).Arc(-0.3, 0.2, 0.2, 0, Math.PI / 2).LineTo(-0.7, 0.4).Arc(-0.7, 0.2, 0.2, Math.PI / 2, Math.PI).LineTo(-0.9, -0.2).Arc(-0.7, -0.2, 0.2, Math.PI, 3 * Math.PI / 2).Close();
            tbr.FillPath(checkboxPath, Colour.FromRgb(240, 246, 249));
            tbr.StrokePath(checkboxPath, Colour.FromRgb(0, 114, 178), 0.075);

            GraphicsPath tickpath = new GraphicsPath().MoveTo(-0.75, -0.1).LineTo(-0.5, 0.15).LineTo(-0.1, -0.4);

            tbr.StrokePath(new GraphicsPath().MoveTo(-0.5, 0.15).LineTo(-0.1, -0.4), Colour.FromRgb(240, 246, 249), 0.3, LineCaps.Round);
            tbr.StrokePath(tickpath, Colour.FromRgb(0, 158, 115), 0.2, LineCaps.Round);

            return tbr;
        })();

        /// <summary>
        /// Determines whether page breaks should be treated as such in the source.
        /// </summary>
        public bool AllowPageBreak { get; set; } = true;

        internal MarkdownRenderer Clone()
        {
            return (MarkdownRenderer)this.MemberwiseClone();
        }

        private MathPainter MathPainter = new MathPainter() { DisplayErrorInline = false };

        /// <summary>
        /// Parses the supplied <paramref name="markdownSource"/> using all the supported extensions and renders the resulting document. Page breaks are disabled, and the document is rendered as a single page with the specified <paramref name="width"/>. The page will be cropped at the appropriate height to contain the entire document.
```

```
        /// </summary>
        /// <param name="markdownSource">The markdown source to parse.</param>
        /// <param name="width">The width of the page.</param>
        /// <param name="linkDestinations">When this method returns, this value will contain a dictionary used to associate graphic action tags to hyperlinks. This can be used to enable such links when rendering the <see cref="Page"/> to a file.</param>
        /// <returns>A <see cref="Page"/> containing a rendering of the supplied markdown document.</returns>
        public Page RenderSinglePage(string markdownSource, double width, out Dictionary<string, string> linkDestinations)
        {
            MarkdownDocument document = Markdig.Markdown.Parse(markdownSource, new MarkdownPipelineBuilder().UseGridTables().UsePipeTables().UseEmphasisExtras().UseGenericAttributes().UseAutoIdentifiers

            return this.RenderSinglePage(document, width, out linkDestinations);
        }

        /// <summary>
        /// Renders the supplied <paramref name="markdownDocument"/>. Page breaks are disabled, and the document is rendered as a single page with the specified <paramref name="width"/>. The page will be cropped at the appropriate height to contain the entire document.
        /// </summary>
        /// <param name="markdownDocument">The markdown document to render.</param>
        /// <param name="width">The width of the page.</param>
        /// <param name="linkDestinations">When this method returns, this value will contain a dictionary used to associate graphic action tags to hyperlinks. This can be used to enable such links when rendering the <see cref="Page"/> to a file.</param>
        /// <returns>A <see cref="Page"/> containing a rendering of the supplied markdown document.</returns>
        public Page RenderSinglePage(MarkdownDocument markdownDocument, double width, out Dictionary<string, string> linkDestinations)
        {
            Size prevPageSize = this.PageSize;
            bool allowPageBreak = this.AllowPageBreak;

            this.PageSize = new Size(width, double.PositiveInfinity);
            this.AllowPageBreak = false;

            Page pag = new Page(PageSize.Width, PageSize.Height) { Background = BackgroundColour };

            Graphics graphics = pag.Graphics;

            graphics.Save();

            graphics.Translate(Margins.Left, Margins.Top);

            void newPageAction(ref MarkdownContext mdContext, ref Graphics pageGraphics)
            {

            }

            MarkdownContext context = new MarkdownContext()
            {
                Font = new Font(RegularFontFamily, BaseFontSize),
                Cursor = new Point(0, 0),
                Colour = ForegroundColour,
                Underline = false,
                Translation = new Point(this.Margins.Left, this.Margins.Top),
                CurrentLine = null,
                ListDepth = 0,
                CurrentPage = pag
            };

            int index = 0;
            foreach (Block block in markdownDocument)
            {
                RenderBlock(block, ref context, ref graphics, newPageAction, index > 0, index < markdownDocument.Count - 1);
                index++;
            }

            if (context.CurrentLine != null)
            {
                context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
            }

            graphics.Restore();

            pag.Crop(new Point(0, 0), new Size(width, context.BottomRight.Y + this.Margins.Bottom));

            linkDestinations = context.LinkDestinations;
            return pag;
        }

        /// <summary>
        /// Parses the supplied <paramref name="markdownSource"/> using all the supported extensions
```

```
and renders the resulting document. The <see cref="Document"/> produced consists of one or more pages of the size specified in the <see cref="PageSize"/> of the current instance.
        /// </summary>
        /// <param name="markdownSource">The markdown source to parse.</param>
        /// <param name="linkDestinations">When this method returns, this value will contain a dictionary used to associate graphic action tags to hyperlinks. This can be used to enable such links when rendering the <see cref="Document"/> to a file.</param>
        /// <returns>A <see cref="Document"/> containing a rendering of the supplied markdown document, consisting of one or more pages of the size specified in the <see cref="PageSize"/> of the current instance.</returns>
        public Document Render(string markdownSource, out Dictionary<string, string> linkDestinations)
        {
            MarkdownDocument document = Markdig.Markdown.Parse(markdownSource, new MarkdownPipelineBuilder().UseGridTables().UsePipeTables().UseEmphasisExtras().UseGenericAttributes().UseAutoIdentifiers

            return this.Render(document, out linkDestinations);
        }

        /// <summary>
        /// Renders the supplied <paramref name="mardownDocument"/>. The <see cref="Document"/> produced consists of one or more pages of the size specified in the <see cref="PageSize"/> of the current instance.
        /// </summary>
        /// <param name="mardownDocument">The markdown document to render.</param>
        /// <param name="linkDestinations">When this method returns, this value will contain a dictionary used to associate graphic action tags to hyperlinks. This can be used to enable such links when rendering the <see cref="Document"/> to a file.</param>
        /// <returns>A <see cref="Document"/> containing a rendering of the supplied markdown document, consisting of one or more pages of the size specified in the <see cref="PageSize"/> of the current instance.</returns>
        public Document Render(MarkdownDocument mardownDocument, out Dictionary<string, string> linkDestinations)
        {
            Document doc = new Document();
            Page pag = new Page(PageSize.Width, PageSize.Height) { Background = BackgroundColour };
            doc.Pages.Add(pag);

            Graphics graphics = pag.Graphics;

            graphics.Translate(Margins.Left, Margins.Top);

            void newPageAction(ref MarkdownContext mdContext, ref Graphics pageGraphics)
            {
                Page newPag = new Page(PageSize.Width, PageSize.Height) { Background = BackgroundColour };
                doc.Pages.Add(newPag);

                newPag.Graphics.Translate(mdContext.Translation);
                mdContext.Cursor = new Point(0, 0);
                mdContext.ForbiddenAreasLeft.Clear();
                mdContext.ForbiddenAreasRight.Clear();

                pageGraphics = newPag.Graphics;
                mdContext.CurrentPage = newPag;
            }

            MarkdownContext context = new MarkdownContext()
            {
                Font = new Font(RegularFontFamily, BaseFontSize),
                Cursor = new Point(0, 0),
                Colour = ForegroundColour,
                Underline = false,
                Translation = new Point(Margins.Left, Margins.Top),
                CurrentLine = null,
                ListDepth = 0,
                CurrentPage = pag
            };

            int index = 0;

            foreach (Block block in mardownDocument)
            {
                RenderBlock(block, ref context, ref graphics, newPageAction, index > 0, index < mardownDocument.Count - 1);
                index++;
            }

            if (context.CurrentLine != null)
            {
                context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
            }

            linkDestinations = context.LinkDestinations;

            return doc;
        }
```

```

        private Document RenderSubDocument(ContainerBlock document, ref MarkdownContext context)
        {
            Document doc = new Document();
            Page pag = new Page(PageSize.Width, PageSize.Height) { Background = BackgroundColour };
            doc.Pages.Add(pag);

            Graphics graphics = pag.Graphics;

            graphics.Save();

            graphics.Translate(Margins.Left, Margins.Top);

            void newPageAction(ref MarkdownContext mdContext, ref Graphics pageGraphics)
            {
                pageGraphics.Restore();
                Page newPag = new Page(PageSize.Width, PageSize.Height) { Background = BackgroundColour };
                doc.Pages.Add(newPag);

                newPag.Graphics.Save();
                newPag.Graphics.Translate(mdContext.Translation);
                mdContext.Cursor = new Point(0, 0);
                mdContext.ForbiddenAreasLeft.Clear();
                mdContext.ForbiddenAreasRight.Clear();

                pageGraphics = newPag.Graphics;
                mdContext.CurrentPage = newPag;
            }

            context.Translation = new Point(context.Translation.X + Margins.Left, context.Translation.Y + Margins.Top);
            context.CurrentPage = pag;
            context.ListDepth = 0;

            int index = 0;
            foreach (Block block in document)
            {
                RenderBlock(block, ref context, ref graphics, newPageAction, index > 0, index < document.Count - 1);
                index++;
            }

            if (context.CurrentLine != null)
            {
                context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
            }

            graphics.Restore();

            return doc;
        }

        internal delegate void NewPageAction(ref MarkdownContext context, ref Graphics graphics);

        private void RenderBlock(Block block, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction, bool spaceBefore, bool spaceAfter)
        {
            HtmlAttributes attributes = block.TryGetAttributes();

            if (attributes != null && !string.IsNullOrEmpty(attributes.Id))
            {
                Point cursor = context.Cursor;
                NewPageAction reversibleNewPageAction = (ref MarkdownContext currContext, ref Graphics currGraphics) =>
                {
                    newPageAction(ref currContext, ref currGraphics);
                    cursor = currContext.Cursor;
                };

                RenderHTMLBlock("<a name=\"" + attributes.Id + "\"></a>", false, ref context, ref graphics, reversibleNewPageAction, spaceBefore, spaceAfter);
                context.Cursor = cursor;
            }

            if (block is LeafBlock leaf)
            {
                if (leaf is HeadingBlock heading)
                {
                    RenderHeadingBlock(heading, ref context, ref graphics, newPageAction, spaceBefore, spaceAfter);
                }
                else if (leaf is ParagraphBlock paragraph)
                {
                    RenderParagraphBlock(paragraph, ref context, ref graphics, newPageAction, spaceBefore, spaceAfter);
```

```
                }
                else if (leaf is CodeBlock code)
                {
                    if (block is Markdig.Extensions.Mathematics.MathBlock math)
                    {
                        StringBuilder mathBuilder = new StringBuilder();
                        foreach (StringLine line in math.Lines)
                        {
                            mathBuilder.Append(line.ToString());
                            mathBuilder.Append("\n");
                        }

                        byte[] svgData;

                        using (MemoryStream ms = new MemoryStream())
                        {
                            MathPainter.LineStyle = global::CSharpMath.Atom.LineStyle.Display;
                            MathPainter.FontSize = (float)(context.Font.FontSize * MathFontScalingFactor / ImageMultiplier);
                            MathPainter.LaTeX = mathBuilder.ToString();
                            Page pag = MathPainter.DrawToPage();
                            VectSharp.SVG.SVGContextInterpreter.SaveAsSVG(pag, ms);
                            svgData = ms.ToArray();
                        }

                        string imageUri = "<img align=\"center\" src=\"" + Convert.ToBase64String(svgData) + "\">";
                        RenderHTMLBlock(imageUri, false, ref context, ref graphics, newPageAction, true, true);
                    }
                    else if (leaf is FencedCodeBlock fenced)
                    {
                        if (!string.IsNullOrEmpty(fenced.Info))
                        {
                            RenderFencedCodeBlock(fenced, ref context, ref graphics, newPageAction, spaceBefore, spaceAfter);
                        }
                        else
                        {
                            RenderCodeBlock(code, ref context, ref graphics, newPageAction, spaceBefore, spaceAfter);
                        }

                    }
                    else
                    {
                        RenderCodeBlock(code, ref context, ref graphics, newPageAction, spaceBefore, spaceAfter);
                    }
                }
                else if (leaf is HtmlBlock html)
                {
                    RenderHTMLBlock(html.Lines.ToString(), false, ref context, ref graphics, newPageAction, spaceBefore, spaceAfter);
                }
                else if (leaf is ThematicBreakBlock thematicBreak)
                {
                    RenderThematicBreakBlock(thematicBreak, ref context, ref graphics, newPageAction, spaceBefore, spaceAfter);
                }
                else if (leaf is LinkReferenceDefinition link)
                {
                    // Nothing to do (the links are correctly referenced by the parser)
                }

            }
            else if (block is ContainerBlock)
            {
                if (block is ListBlock list)
                {
                    RenderListBlock(list, ref context, ref graphics, newPageAction);
                }
                else if (block is ListItemBlock listItem)
                {
                    RenderListItemBlock(listItem, ref context, ref graphics, newPageAction);
                }
                else if (block is QuoteBlock quote)
                {
                    RenderQuoteBlock(quote, ref context, ref graphics, newPageAction);
                }
                else if (block is LinkReferenceDefinitionGroup linkGroup)
                {
                    // Nothing to render here
                }
                else if (block is Table table)
                {
                    RenderTable(table, ref context, ref graphics, newPageAction);
```

```
                }
            }
            else if (block is BlankLineBlock)
            {
                // Nothing to render here
            }
        }

        private void RenderHeadingBlock(HeadingBlock heading, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction, bool spaceBefore, bool spaceAfter)
        {
            MarkdownContext prevContext = context.Clone();

            context.Font = new Font(this.RegularFontFamily, BaseFontSize * HeaderFontSizeMultipliers[heading.Level - 1]);

            double minX = context.GetMinX(context.Cursor.Y + SpaceBeforeHeading * context.Font.FontSize, context.Cursor.Y + context.Font.Ascent + SpaceBeforeHeading * context.Font.FontSize - context.Font.Descent);

            if (spaceBefore)
            {
                context.Cursor = new Point(minX, context.Cursor.Y + context.Font.Ascent + SpaceBeforeHeading * context.Font.FontSize);
            }
            else
            {
                context.Cursor = new Point(minX, context.Cursor.Y + context.Font.Ascent);
            }

            if (context.CurrentLine == null)
            {
                context.CurrentLine = new Line(context.Font.Ascent);
            }
            else
            {
                double delta = context.Cursor.Y - (prevContext.Cursor.Y + prevContext.Font.Ascent + SpaceBeforeParagaph * prevContext.Font.FontSize);

                for (int i = 0; i < context.CurrentLine.Fragments.Count; i++)
                {
                    context.CurrentLine.Fragments[i].Translate(0, delta);
                }
            }

            foreach (Inline inline in heading.Inline)
            {
                RenderInline(inline, ref context, ref graphics, newPageAction);
            }

            if (this.HeaderLineThicknesses[heading.Level - 1] > 0)
            {
                double lineY = context.Cursor.Y + context.Font.FontSize * 0.3;
                context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(minX, context.Cursor.Y + context.Font.FontSize * 0.3), new Point(context.GetMaxX(lineY, PageSize.Width - Margins.Right - context.Translation.X), lineY), this.HeaderLineColour, this.HeaderLineThicknesses[heading.Level - 1], context.Tag));
            }

            context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
            context.CurrentLine = null;

            if (this.HeaderLineThicknesses[heading.Level - 1] > 0)
            {
                double lineY = context.Cursor.Y + context.Font.FontSize * 0.3;
                context.Cursor = new Point(context.Cursor.X, lineY + this.HeaderLineThicknesses[heading.Level - 1]);
            }

            context.Cursor = new Point(0, context.Cursor.Y - context.Font.Descent + SpaceAfterLine * context.Font.FontSize);

            if (spaceAfter)
            {
                context.Cursor = new Point(0, context.Cursor.Y + SpaceAfterHeading * context.Font.FontSize);
            }

            prevContext.Cursor = context.Cursor;
            prevContext.BottomRight = context.BottomRight;
            prevContext.CurrentPage = context.CurrentPage;
            prevContext.CurrentLine = context.CurrentLine;

            context = prevContext;
        }

```

```
        private void RenderParagraphBlock(ParagraphBlock paragraph, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction, bool spaceBefore, bool spaceAfter)
        {
            double minX = context.GetMinX(context.Cursor.Y + SpaceBeforeParagaph * context.Font.FontSize, context.Cursor.Y + context.Font.Ascent + SpaceBeforeParagaph * context.Font.FontSize - context.Font.Descent);

            if (spaceBefore)
            {
                context.Cursor = new Point(minX, context.Cursor.Y + context.Font.Ascent + SpaceBeforeParagaph * context.Font.FontSize);
            }
            else
            {
                context.Cursor = new Point(minX, context.Cursor.Y + context.Font.Ascent);
            }

            if (context.CurrentLine == null)
            {
                context.CurrentLine = new Line(context.Font.Ascent);
            }

            foreach (Inline inline in paragraph.Inline)
            {
                RenderInline(inline, ref context, ref graphics, newPageAction);
            }

            context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
            context.CurrentLine = null;

            context.Cursor = new Point(0, context.Cursor.Y - context.Font.Descent + SpaceAfterLine * context.Font.FontSize);

            if (spaceAfter)
            {
                context.Cursor = new Point(0, context.Cursor.Y + SpaceAfterParagraph * context.Font.FontSize);
            }
        }

        private void RenderFencedCodeBlock(FencedCodeBlock codeBlock, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction, bool spaceBefore, bool spaceAfter)
        {
            string info = codeBlock.Info;

            if (string.IsNullOrEmpty(info) || codeBlock.Lines.Count == 0)
            {
                RenderCodeBlock(codeBlock, ref context, ref graphics, newPageAction, spaceBefore, spaceAfter);
                return;
            }

            StringBuilder code = new StringBuilder();

            foreach (StringLine line in codeBlock.Lines)
            {
                code.Append(line.ToString());
                code.Append('\n');
            }

            List<List<FormattedString>> lines = this.SyntaxHighlighter(code.ToString(0, code.Length - 1), info);

            if (lines == null)
            {
                RenderCodeBlock(codeBlock, ref context, ref graphics, newPageAction, spaceBefore, spaceAfter);
                return;
            }

            MarkdownContext prevContext = context.Clone();

            context.Font = new Font(this.CodeFont, context.Font.FontSize);

            if (spaceBefore)
            {
                context.Cursor = new Point(0, context.Cursor.Y + SpaceBeforeParagaph * context.Font.FontSize);
            }

            int index = 0;

            if (codeBlock.Lines.Count > 0)
            {
                foreach (List<FormattedString> line in lines)
                {
```

```
                    if (index < codeBlock.Lines.Count)
                    {
                        if (context.CurrentLine == null)
                        {
                            context.CurrentLine = new Line(context.Font.Ascent);
                        }

                        double maxX = context.GetMaxX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X);

                        double minX = context.GetMinX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent);

                        context.Cursor = new Point(minX + context.Font.FontSize, context.Cursor.Y + context.Font.YMax);

                        if (index == 0)
                        {
                            context.CurrentLine.Fragments.Insert(0, new RectangleFragment(new Point(minX, context.Cursor.Y - context.Font.YMax - this.SpaceAfterLine * context.Font.FontSize), new Size(maxX - minX, this.SpaceAfterLine * context.Font.FontSize * 2), CodeBlockBackgroundColour, context.Tag));
                        }

                        foreach (FormattedString item in line)
                        {
                            context.Colour = item.Colour;

                            if (!item.IsBold && !item.IsItalic)
                            {
                                context.Font = new Font(this.CodeFont, context.Font.FontSize);
                            }
                            else if (item.IsBold && !item.IsItalic)
                            {
                                context.Font = new Font(this.CodeFontBold, context.Font.FontSize);
                            }
                            else if (item.IsBold && item.IsItalic)
                            {
                                context.Font = new Font(this.CodeFontBoldItalic, context.Font.FontSize);
                            }
                            else if (!item.IsBold && item.IsItalic)
                            {
                                context.Font = new Font(this.CodeFontItalic, context.Font.FontSize);
                            }

                            RenderCodeBlockLine(item.Text, ref context, ref graphics, newPageAction);
                        }


                        context.Colour = Colours.Black;

                        context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
                        context.CurrentLine = null;

                        context.Cursor = new Point(0, context.Cursor.Y - context.Font.YMin);
                        index++;
                    }
                    else
                    {
                        break;
                    }
                }
            }

            context.Cursor = new Point(0, context.Cursor.Y + SpaceAfterLine * context.Font.FontSize);

            if (spaceAfter)
            {
                context.Cursor = new Point(0, context.Cursor.Y + SpaceAfterParagraph * context.Font.FontSize);
            }

            prevContext.Cursor = context.Cursor;
            prevContext.BottomRight = context.BottomRight;
            prevContext.CurrentPage = context.CurrentPage;
            prevContext.CurrentLine = context.CurrentLine;

            context = prevContext;
        }


        private void RenderCodeBlock(CodeBlock codeBlock, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction, bool spaceBefore, bool spaceAfter)
        {
```

```
            MarkdownContext prevContext = context.Clone();

            context.Font = new Font(this.CodeFont, context.Font.FontSize);

            if (spaceBefore)
            {
                context.Cursor = new Point(0, context.Cursor.Y + SpaceBeforeParagaph * context.Font.FontSize);
            }

            int index = 0;

            if (codeBlock.Lines.Count > 0)
            {
                foreach (StringLine line in codeBlock.Lines)
                {
                    if (index < codeBlock.Lines.Count)
                    {
                        if (context.CurrentLine == null)
                        {
                            context.CurrentLine = new Line(context.Font.Ascent);
                        }

                        double maxX = context.GetMaxX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X);

                        double minX = context.GetMinX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent);

                        context.Cursor = new Point(minX + context.Font.FontSize, context.Cursor.Y + context.Font.YMax);

                        if (index == 0)
                        {
                            context.CurrentLine.Fragments.Insert(0, new RectangleFragment(new Point(minX, context.Cursor.Y - context.Font.YMax - this.SpaceAfterLine * context.Font.FontSize), new Size(maxX - minX, this.SpaceAfterLine * context.Font.FontSize * 2), CodeBlockBackgroundColour, context.Tag));
                        }


                        RenderCodeBlockLine(line.ToString(), ref context, ref graphics, newPageAction);


                        context.Colour = Colours.Black;

                        context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
                        context.CurrentLine = null;

                        context.Cursor = new Point(0, context.Cursor.Y - context.Font.YMin);
                        index++;
                    }
                    else
                    {
                        break;
                    }
                }
            }

            context.Cursor = new Point(0, context.Cursor.Y + SpaceAfterLine * context.Font.FontSize);

            if (spaceAfter)
            {
                context.Cursor = new Point(0, context.Cursor.Y + SpaceAfterParagraph * context.Font.FontSize);
            }

            prevContext.Cursor = context.Cursor;
            prevContext.BottomRight = context.BottomRight;
            prevContext.CurrentPage = context.CurrentPage;
            prevContext.CurrentLine = context.CurrentLine;

            context = prevContext;
        }

        private void RenderThematicBreakBlock(ThematicBreakBlock thematicBreak, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction, bool spaceBefore, bool spaceAfter)
        {
            if (context.CurrentLine == null)
            {
                context.CurrentLine = new Line(0);
            }

            if (spaceBefore)
```

```
            {
                context.Cursor = new Point(context.Cursor.X, context.Cursor.Y + SpaceBeforeParagaph * context.Font.FontSize + this.ThematicBreakThickness * 0.5);
            }
            else
            {
                context.Cursor = new Point(context.Cursor.X, context.Cursor.Y + this.ThematicBreakThickness * 0.5);
            }


            double maxX = context.GetMaxX(context.Cursor.Y, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X);
            double minX = context.GetMinX(context.Cursor.Y);

            context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(minX, context.Cursor.Y), new Point(maxX, context.Cursor.Y), this.ThematicBreakLineColour, this.ThematicBreakThickness, context.Tag));

            context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
            context.CurrentLine = null;

            if (spaceAfter)
            {
                context.Cursor = new Point(context.Cursor.X, context.Cursor.Y + SpaceAfterParagraph * context.Font.FontSize + this.ThematicBreakThickness * 0.5);
            }
            else
            {
                context.Cursor = new Point(context.Cursor.X, context.Cursor.Y + this.ThematicBreakThickness * 0.5);
            }
        }

        private void RenderInline(Inline inline, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            HtmlAttributes attributes = inline.TryGetAttributes();

            if (attributes != null && !string.IsNullOrEmpty(attributes.Id))
            {
                Point cursor = context.Cursor;
                RenderHTMLBlock("<a name=\"" + attributes.Id + "\"></a>", true, ref context, ref graphics, newPageAction, false, false);
                context.Cursor = cursor;
            }

            if (inline is LeafInline)
            {
                if (inline is AutolinkInline autoLink)
                {
                    LinkInline link = new LinkInline((autoLink.IsEmail ? "mailto:" : "") + autoLink.Url, "");
                    link.AppendChild(new LiteralInline(autoLink.Url));

                    RenderLinkInline(link, ref context, ref graphics, newPageAction);
                }
                else if (inline is CodeInline code)
                {
                    RenderCodeInline(code, ref context, ref graphics, newPageAction);
                }
                else if (inline is HtmlEntityInline htmlEntity)
                {
                    RenderLiteralInline(new LiteralInline(htmlEntity.Transcoded), ref context, ref graphics, newPageAction);
                }
                else if (inline is HtmlInline html)
                {
                    RenderHTMLBlock(html.Tag, true, ref context, ref graphics, newPageAction, true, true);
                }
                else if (inline is LineBreakInline lineBreak)
                {
                    RenderLineBreakInline(lineBreak.IsHard, false, ref context, ref graphics, newPageAction);
                }
                else if (inline is LiteralInline literal)
                {
                    RenderLiteralInline(literal, ref context, ref graphics, newPageAction);
                }
                else if (inline is Markdig.Extensions.Mathematics.MathInline math)
                {
                    byte[] svgData;

                    using (MemoryStream ms = new MemoryStream())
                    {
```

```
                        if (math.DelimiterCount == 1)
                        {
                            MathPainter.LineStyle = global::CSharpMath.Atom.LineStyle.Text;
                        }
                        else
                        {
                            MathPainter.LineStyle = global::CSharpMath.Atom.LineStyle.Display;
                        }

                        MathPainter.FontSize = (float)(context.Font.FontSize * MathFontScalingFactor / ImageMultiplier);

                        MathPainter.LaTeX = math.Content.ToString();
                        Page pag = MathPainter.DrawToPage();
                        VectSharp.SVG.SVGContextInterpreter.SaveAsSVG(pag, ms);
                        svgData = ms.ToArray();
                    }

                    string imageUri = "<img src=\"" + Convert.ToBase64String(svgData) + "\">";
                    RenderHTMLBlock(imageUri, true, ref context, ref graphics, newPageAction, true, true);
                }
                else if (inline is Markdig.Extensions.SmartyPants.SmartyPant smartyPant)
                {
                    switch (smartyPant.Type)
                    {
                        case Markdig.Extensions.SmartyPants.SmartyPantType.LeftDoubleQuote:
                            RenderLiteralInline(new LiteralInline("“"), ref context, ref graphics, newPageAction);
                            break;
                        case Markdig.Extensions.SmartyPants.SmartyPantType.RightDoubleQuote:
                            RenderLiteralInline(new LiteralInline("”"), ref context, ref graphics, newPageAction);
                            break;
                        case Markdig.Extensions.SmartyPants.SmartyPantType.LeftQuote:
                            RenderLiteralInline(new LiteralInline("‘"), ref context, ref graphics, newPageAction);
                            break;
                        case Markdig.Extensions.SmartyPants.SmartyPantType.RightQuote:
                            RenderLiteralInline(new LiteralInline("’"), ref context, ref graphics, newPageAction);
                            break;
                        case Markdig.Extensions.SmartyPants.SmartyPantType.Dash2:
                            RenderLiteralInline(new LiteralInline("–"), ref context, ref graphics, newPageAction);
                            break;
                        case Markdig.Extensions.SmartyPants.SmartyPantType.Dash3:
                            RenderLiteralInline(new LiteralInline("—"), ref context, ref graphics, newPageAction);
                            break;
                        case Markdig.Extensions.SmartyPants.SmartyPantType.DoubleQuote:
                            RenderLiteralInline(new LiteralInline("\""), ref context, ref graphics, newPageAction);
                            break;
                        case Markdig.Extensions.SmartyPants.SmartyPantType.Ellipsis:
                            RenderLiteralInline(new LiteralInline("..."), ref context, ref graphics, newPageAction);
                            break;
                        case Markdig.Extensions.SmartyPants.SmartyPantType.LeftAngleQuote:
                            RenderLiteralInline(new LiteralInline("«"), ref context, ref graphics, newPageAction);
                            break;
                        case Markdig.Extensions.SmartyPants.SmartyPantType.Quote:
                            RenderLiteralInline(new LiteralInline("’"), ref context, ref graphics, newPageAction);
                            break;
                        case Markdig.Extensions.SmartyPants.SmartyPantType.RightAngleQuote:
                            RenderLiteralInline(new LiteralInline("»"), ref context, ref graphics, newPageAction);
                            break;
                    }
                }
                else if (inline is Markdig.Extensions.TaskLists.TaskList)
                {
                    // Nothing to render here (the checkbox has already been rendered)
                }
            }
            else if (inline is ContainerInline)
            {
                if (inline is DelimiterInline)
                {
                    // Nothing to render here
                }
                else if (inline is EmphasisInline emphasis)
                {
                    RenderEmphasisInline(emphasis, ref context, ref graphics, newPageAction);
```

```
                }
                else if (inline is LinkInline link)
                {
                    RenderLinkInline(link, ref context, ref graphics, newPageAction);
                }
            }
        }

        private void RenderLineBreakInline(bool isHard, bool isPageBreak, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            if (isHard)
            {
                context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
                context.CurrentLine = new Line(context.Font.Ascent);
                context.Cursor = new Point(0, context.Cursor.Y - context.Font.Descent + SpaceAfterLine * context.Font.FontSize + context.Font.Ascent);

                double minX = context.GetMinX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent);

                context.Cursor = new Point(minX, context.Cursor.Y);

                if (isPageBreak)
                {
                    newPageAction(ref context, ref graphics);
                }
            }
            else
            {
                double spaceWidth = context.Font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(' ') / 1000.0 * context.Font.FontSize;
                context.Cursor = new Point(context.Cursor.X + spaceWidth, context.Cursor.Y);
            }
        }

        private void RenderLiteralInline(LiteralInline literal, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            string text = literal.Content.ToString();

            double spaceWidth = context.Font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(' ') / 1000.0 * context.Font.FontSize;

            List<Word> words = Word.GetWords(text, context.Font, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X).ToList();

            double underlineStart = context.Cursor.X;
            double underlineEnd = context.Cursor.X;

            double currLineMaxX = context.GetMaxX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X);

            bool ignoreNextWhitespace = false;

            bool broken = false;

            for (int i = 0; i < words.Count; i++)
            {
                Word w = words[i];

                if (!string.IsNullOrEmpty(w.Text))
                {
                    if (!ignoreNextWhitespace)
                    {
                        context.Cursor = new Point(context.Cursor.X + spaceWidth * w.WhitespaceCount * (w.PrecedingWhitespace == '\t' ? 4 : 1), context.Cursor.Y);
                    }
                    else
                    {
                        ignoreNextWhitespace = false;
                    }

                    Font.DetailedFontMetrics wordMetrics = w.Metrics;

                    double finalX = context.Cursor.X + wordMetrics.Width + wordMetrics.RightSideBearing + wordMetrics.LeftSideBearing;

                    if (finalX <= currLineMaxX || broken)
                    {
                        context.CurrentLine.Fragments.Add(new TextFragment(new Point(context.Cursor.X + wordMetrics.LeftSideBearing, context.Cursor.Y), w.Text, context.Font, context.Colour, context.Tag));
                        context.Cursor = new Point(context.Cursor.X + wordMetrics.Width + wordMetrics.RightSideBearing + wordMetrics.LeftSideBearing, context.Cursor.Y);
```

```
                        broken = false;

                        if (context.Underline || context.StrikeThrough)
                        {
                            underlineEnd = context.Cursor.X;
                        }
                    }
                    else
                    {
                        if (context.Underline && underlineStart != underlineEnd)
                        {
                            context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y + context.Font.FontSize * 0.2), new Point(underlineEnd, context.Cursor.Y + context.Font.FontSize * 0.2), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
                        }
                        else if (context.StrikeThrough && underlineStart != underlineEnd)
                        {
                            context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), new Point(underlineEnd, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
                        }

                        context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);



                        context.CurrentLine = new Line(context.Font.Ascent);
                        context.Cursor = new Point(0, context.Cursor.Y - context.Font.Descent + SpaceAfterLine * context.Font.FontSize + context.Font.Ascent);
                        currLineMaxX = context.GetMaxX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X);

                        double minX = context.GetMinX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent);

                        context.Cursor = new Point(minX, context.Cursor.Y);

                        underlineStart = minX;
                        underlineEnd = minX;

                        i--;
                        ignoreNextWhitespace = true;
                        broken = true;
                    }
                }
                else
                {
                    context.Cursor = new Point(context.Cursor.X + spaceWidth * w.WhitespaceCount * (w.PrecedingWhitespace == '\t' ? 4 : 1), context.Cursor.Y);
                }
            }

            if (context.Underline && underlineStart != underlineEnd)
            {
                context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y + context.Font.FontSize * 0.2), new Point(underlineEnd, context.Cursor.Y + context.Font.FontSize * 0.2), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
            }
            else if (context.StrikeThrough && underlineStart != underlineEnd)
            {
                context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), new Point(underlineEnd, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
            }
        }

        private void RenderCodeInline(CodeInline code, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            MarkdownContext prevContext = context.Clone();

            context.Font = new Font(this.CodeFont, context.Font.FontSize);

            double spaceWidth = context.Font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(' ') / 1000.0 * context.Font.FontSize;

            string text = code.Content.ToString();
```

```
            List<Word> words = Word.GetWords(text, context.Font, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X).ToList();

            double currLineMaxX = context.GetMaxX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X);

            context.Cursor = new Point(context.Cursor.X + this.CodeInlineMargin * context.Font.FontSize, context.Cursor.Y);


            double underlineStart = context.Cursor.X;
            double underlineEnd = context.Cursor.X;

            bool broken = false;

            for (int i = 0; i < words.Count; i++)
            {
                Word w = words[i];

                if (!string.IsNullOrEmpty(w.Text))
                {
                    context.Cursor = new Point(context.Cursor.X + spaceWidth * w.WhitespaceCount * (w.PrecedingWhitespace == '\t' ? 4 : 1), context.Cursor.Y);

                    Font.DetailedFontMetrics wordMetrics = w.Metrics;

                    double finalX = context.Cursor.X + wordMetrics.Width + wordMetrics.RightSideBearing + wordMetrics.LeftSideBearing;

                    if (finalX <= currLineMaxX || broken)
                    {
                        if (i == 0)
                        {
                            context.CurrentLine.Fragments.Add(new RectangleFragment(new Point(context.Cursor.X - this.CodeInlineMargin * context.Font.FontSize, context.Cursor.Y - context.Font.YMax), new Size(this.CodeInlineMargin * context.Font.FontSize, context.Font.YMax - context.Font.YMin), CodeInlineBackgroundColour, context.Tag));
                        }

                        context.CurrentLine.Fragments.Add(new RectangleFragment(new Point(context.Cursor.X - spaceWidth * w.WhitespaceCount * (w.PrecedingWhitespace == '\t' ? 4 : 1), context.Cursor.Y - context.Font.YMax), new Size(wordMetrics.Width + wordMetrics.LeftSideBearing * 2 + wordMetrics.RightSideBearing + spaceWidth * w.WhitespaceCount * (w.PrecedingWhitespace == '\t' ? 4 : 1), context.Font.YMax - context.Font.YMin), CodeInlineBackgroundColour, context.Tag));

                        context.CurrentLine.Fragments.Add(new TextFragment(new Point(context.Cursor.X + wordMetrics.LeftSideBearing, context.Cursor.Y), w.Text, context.Font, context.Colour, context.Tag));
                        context.Cursor = new Point(context.Cursor.X + wordMetrics.Width + wordMetrics.RightSideBearing + wordMetrics.LeftSideBearing, context.Cursor.Y);

                        broken = false;

                        if (context.Underline || context.StrikeThrough)
                        {
                            underlineEnd = context.Cursor.X;
                        }
                    }
                    else
                    {
                        if (context.Underline && underlineStart != underlineEnd)
                        {
                            context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y + context.Font.FontSize * 0.2), new Point(underlineEnd, context.Cursor.Y + context.Font.FontSize * 0.2), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
                        }
                        else if (context.StrikeThrough && underlineStart != underlineEnd)
                        {
                            context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), new Point(underlineEnd, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
                        }

                        context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);

                        context.CurrentLine = new Line(prevContext.Font.Ascent);

                        context.Cursor = new Point(0, context.Cursor.Y - prevContext.Font.Descent + SpaceAfterLine * prevContext.Font.FontSize + prevContext.Font.Ascent);


```

```
                        double minX = context.GetMinX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent);

                        context.Cursor = new Point(minX, context.Cursor.Y);

                        if (i == 0)
                        {
                            context.Cursor = new Point(context.Cursor.X + this.CodeInlineMargin * context.Font.FontSize, context.Cursor.Y);
                        }

                        underlineStart = minX;
                        underlineEnd = minX;

                        i--;
                        broken = true;
                    }
                }
            }


            if (context.Underline && underlineStart != underlineEnd)
            {
                context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y + context.Font.FontSize * 0.2), new Point(underlineEnd, context.Cursor.Y + context.Font.FontSize * 0.2), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
            }
            else if (context.StrikeThrough && underlineStart != underlineEnd)
            {
                context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), new Point(underlineEnd, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
            }

            context.CurrentLine.Fragments.Add(new RectangleFragment(new Point(context.Cursor.X, context.Cursor.Y - context.Font.YMax), new Size(this.CodeInlineMargin * context.Font.FontSize, context.Font.YMax - context.Font.YMin), CodeInlineBackgroundColour, context.Tag));

            context.Cursor = new Point(context.Cursor.X + this.CodeInlineMargin * context.Font.FontSize, context.Cursor.Y);

            prevContext.Cursor = context.Cursor;
            prevContext.BottomRight = context.BottomRight;
            prevContext.CurrentPage = context.CurrentPage;
            prevContext.CurrentLine = context.CurrentLine;

            context = prevContext;
        }

        private void RenderCodeBlockLine(string text, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            double spaceWidth = context.Font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(' ') / 1000.0 * context.Font.FontSize;

            List<Word> words = Word.GetWords(text, context.Font, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.Font.FontSize * 2 - context.MarginBottomRight.X).ToList();

            double underlineStart = context.Cursor.X;
            double underlineEnd = context.Cursor.X;

            double minX = context.GetMinX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent);

            double currLineMaxX = context.GetMaxX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X) - context.Font.FontSize;

            bool broken = false;

            for (int i = 0; i < words.Count; i++)
            {
                Word w = words[i];
                if (!string.IsNullOrEmpty(w.Text))
                {
                    context.Cursor = new Point(context.Cursor.X + spaceWidth * w.WhitespaceCount * (w.PrecedingWhitespace == '\t' ? 4 : 1), context.Cursor.Y);
                    Font.DetailedFontMetrics wordMetrics = w.Metrics;

                    double finalX = context.Cursor.X + wordMetrics.Width + wordMetrics.RightSideBearing + wordMetrics.LeftSideBearing;

                    double effW = wordMetrics.Width + wordMetrics.RightSideBearing +
```

```
wordMetrics.LeftSideBearing;

                    double maxW = this.PageSize.Width - this.Margins.Right - context.Translation.X - context.Font.FontSize * 2 - spaceWidth * w.WhitespaceCount * (w.PrecedingWhitespace == '\t' ? 4 : 1) - context.MarginBottomRight.X;

                    if (finalX <= currLineMaxX || broken)
                    {
                        broken = false;
                        context.CurrentLine.Fragments.Add(new TextFragment(new Point(context.Cursor.X + wordMetrics.LeftSideBearing, context.Cursor.Y), w.Text, context.Font, context.Colour, context.Tag));
                        context.Cursor = new Point(context.Cursor.X + wordMetrics.Width + wordMetrics.RightSideBearing + wordMetrics.LeftSideBearing, context.Cursor.Y);

                        if (context.Underline || context.StrikeThrough)
                        {
                            underlineEnd = context.Cursor.X;
                        }
                    }
                    else
                    {
                        context.CurrentLine.Fragments.Insert(0, new RectangleFragment(new Point(minX, context.Cursor.Y - context.Font.YMax), new Size(currLineMaxX + context.Font.FontSize - minX, context.Font.YMax - context.Font.YMin + this.SpaceAfterLine * context.Font.FontSize), CodeBlockBackgroundColour, context.Tag));

                        if (context.Underline && underlineStart != underlineEnd)
                        {
                            context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y + context.Font.FontSize * 0.2), new Point(underlineEnd, context.Cursor.Y + context.Font.FontSize * 0.2), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
                        }
                        else if (context.StrikeThrough && underlineStart != underlineEnd)
                        {
                            context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), new Point(underlineEnd, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
                        }

                        context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);

                        underlineStart = 0;
                        underlineEnd = 0;

                        context.CurrentLine = new Line(context.Font.Ascent);

                        context.Cursor = new Point(0, context.Cursor.Y - context.Font.Descent + SpaceAfterLine * context.Font.FontSize + context.Font.Ascent);

                        currLineMaxX = context.GetMaxX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X) - context.Font.FontSize;

                        minX = context.GetMinX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent);

                        context.Cursor = new Point(minX + context.Font.FontSize, context.Cursor.Y);

                        i--;
                        broken = true;
                    }
                }
                else
                {
                    context.Cursor = new Point(context.Cursor.X + spaceWidth * w.WhitespaceCount * (w.PrecedingWhitespace == '\t' ? 4 : 1), context.Cursor.Y);
                }
            }

            if (context.Underline && underlineStart != underlineEnd)
            {
                context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y + context.Font.FontSize * 0.2), new Point(underlineEnd, context.Cursor.Y + context.Font.FontSize * 0.2), context.Colour, context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
            }
            else if (context.StrikeThrough && underlineStart != underlineEnd)
            {
                context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(underlineStart, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), new Point(underlineEnd, context.Cursor.Y - context.Font.Ascent * 0.5 - context.Font.Descent * 0.5), context.Colour,
```

```
context.Font.FontSize * (context.Font.FontFamily.IsBold ? this.BoldUnderlineThickness : this.UnderlineThickness), context.Tag));
            }

            context.CurrentLine.Fragments.Insert(0, new RectangleFragment(new Point(minX, context.Cursor.Y - context.Font.YMax), new Size(currLineMaxX + context.Font.FontSize - minX, context.Font.YMax - context.Font.YMin + this.SpaceAfterLine * context.Font.FontSize), CodeBlockBackgroundColour, context.Tag));
        }

        private void RenderEmphasisInline(EmphasisInline emphasis, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            MarkdownContext prevContext = context.Clone();

            Point translationToUndo = new Point(0, 0);

            switch (emphasis.DelimiterChar)
            {
                case '*':
                case '_':
                    if (emphasis.DelimiterCount == 2)
                    {
                        if (context.Font.FontFamily == this.ItalicFontFamily)
                        {
                            context.Font = new Font(this.BoldItalicFontFamily, context.Font.FontSize);
                        }
                        else if (context.Font.FontFamily == this.BoldFontFamily)
                        {
                            context.Font = new Font(this.RegularFontFamily, context.Font.FontSize);
                        }
                        else if (context.Font.FontFamily == this.BoldItalicFontFamily)
                        {
                            context.Font = new Font(this.ItalicFontFamily, context.Font.FontSize);
                        }
                        else
                        {
                            context.Font = new Font(this.BoldFontFamily, context.Font.FontSize);
                        }
                    }
                    else if (emphasis.DelimiterCount == 3)
                    {
                        if (context.Font.FontFamily == this.ItalicFontFamily)
                        {
                            context.Font = new Font(this.BoldFontFamily, context.Font.FontSize);
                        }
                        else if (context.Font.FontFamily == this.BoldFontFamily)
                        {
                            context.Font = new Font(this.ItalicFontFamily, context.Font.FontSize);
                        }
                        else if (context.Font.FontFamily == this.BoldItalicFontFamily)
                        {
                            context.Font = new Font(this.RegularFontFamily, context.Font.FontSize);
                        }
                        else
                        {
                            context.Font = new Font(this.BoldItalicFontFamily, context.Font.FontSize);
                        }
                    }
                    else
                    {
                        if (context.Font.FontFamily == this.ItalicFontFamily)
                        {
                            context.Font = new Font(this.RegularFontFamily, context.Font.FontSize);
                        }
                        else if (context.Font.FontFamily == this.BoldFontFamily)
                        {
                            context.Font = new Font(this.BoldItalicFontFamily, context.Font.FontSize);
                        }
                        else if (context.Font.FontFamily == this.BoldItalicFontFamily)
                        {
                            context.Font = new Font(this.BoldFontFamily, context.Font.FontSize);
                        }
                        else
                        {
                            context.Font = new Font(this.ItalicFontFamily, context.Font.FontSize);
                        }
                    }
                    break;
                case '"':
                    if (emphasis.DelimiterCount == 2)
                    {
                        if (context.Font.FontFamily == this.ItalicFontFamily)
                        {
                            context.Font = new Font(this.RegularFontFamily, context.Font.FontSize);
                        }
                        else if (context.Font.FontFamily == this.BoldFontFamily)
```

```
                        {
                            context.Font = new Font(this.BoldItalicFontFamily, context.Font.FontSize);
                        }
                        else if (context.Font.FontFamily == this.BoldItalicFontFamily)
                        {
                            context.Font = new Font(this.BoldFontFamily, context.Font.FontSize);
                        }
                        else
                        {
                            context.Font = new Font(this.ItalicFontFamily, context.Font.FontSize);
                        }
                    }
                    break;
                case '~':
                    if (emphasis.DelimiterCount == 1)
                    {
                        //subscript;
                        context.Cursor = new Point(context.Cursor.X, context.Cursor.Y + context.Font.FontSize * this.SubscriptShift);
                        translationToUndo = new Point(translationToUndo.X, translationToUndo.Y + context.Font.FontSize * this.SubscriptShift);
                        context.Font = new Font(context.Font.FontFamily, context.Font.FontSize * this.SubSuperscriptFontSize);
                    }
                    else
                    {
                        //strikethrough
                        context.StrikeThrough = true;
                    }
                    break;
                case '^':
                    if (emphasis.DelimiterCount == 1)
                    {
                        //superscript
                        context.Cursor = new Point(context.Cursor.X, context.Cursor.Y - context.Font.FontSize * this.SuperscriptShift);
                        translationToUndo = new Point(translationToUndo.X, translationToUndo.Y - context.Font.FontSize * this.SuperscriptShift);
                        context.Font = new Font(context.Font.FontFamily, context.Font.FontSize * this.SubSuperscriptFontSize);
                    }
                    break;
                case '+':
                    context.Colour = this.InsertedColour;
                    break;
                case '=':
                    context.Colour = this.MarkedColour;
                    break;
            }

            foreach (Inline innerInline in emphasis)
            {
                RenderInline(innerInline, ref context, ref graphics, newPageAction);
            }

            if (translationToUndo.X != 0 || translationToUndo.Y != 0)
            {
                context.Cursor = new Point(context.Cursor.X - translationToUndo.X, context.Cursor.Y - translationToUndo.Y);
            }


            prevContext.Cursor = context.Cursor;
            prevContext.BottomRight = context.BottomRight;
            prevContext.CurrentPage = context.CurrentPage;
            prevContext.CurrentLine = context.CurrentLine;

            context = prevContext;
        }

        private void RenderLinkInline(LinkInline link, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            if (!link.IsImage)
            {
                MarkdownContext prevContext = context.Clone();

                context.Colour = this.LinkColour;
                context.Underline = true;
                string tag = Guid.NewGuid().ToString("N");

                if (!link.Url.StartsWith("#"))
                {
                    if (Uri.TryCreate(this.BaseLinkUri, link.Url, out Uri uri))
                    {
                        context.LinkDestinations[tag] = this.LinkUriResolver(uri.ToString());
                    }
```

```
                    else
                    {
                        context.LinkDestinations[tag] = this.LinkUriResolver(link.Url);
                    }
                }
                else
                {
                    if (!context.InternalAnchors.TryGetValue(link.Url, out string anchor))
                    {
                        anchor = Guid.NewGuid().ToString("N");
                        context.InternalAnchors[link.Url] = anchor;
                    }

                    context.LinkDestinations[tag] = "#" + anchor;
                }

                context.Tag = tag;

                foreach (Inline innerInline in link)
                {
                    RenderInline(innerInline, ref context, ref graphics, newPageAction);
                }

                prevContext.Cursor = context.Cursor;
                prevContext.BottomRight = context.BottomRight;
                prevContext.CurrentPage = context.CurrentPage;
                prevContext.CurrentLine = context.CurrentLine;

                context = prevContext;
            }
            else
            {
                HtmlTag tag = HtmlTag.Parse("<img src=\"" + link.Url.Replace("\"", "\\\"") + "\">").FirstOrDefault();

                RenderHTMLImage(tag, true, ref context, ref graphics, newPageAction);
            }
        }

        private void RenderListBlock(ListBlock list, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            MarkdownContext prevContext = context.Clone();

            context.ListDepth++;

            double minX = context.GetMinX(context.Cursor.Y + SpaceBeforeParagaph * context.Font.FontSize, context.Cursor.Y - context.Font.Descent + context.Font.Ascent + SpaceBeforeParagaph * context.Font.FontSize);

            if (context.CurrentLine != null)
            {
                foreach (LineFragment fragment in context.CurrentLine.Fragments)
                {
                    fragment.Translate(-minX - this.IndentWidth, 0);
                }
            }

            graphics.Translate(minX + this.IndentWidth, 0);
            context.Translation = new Point(context.Translation.X + minX + this.IndentWidth, context.Translation.Y);

            foreach (Block block in list)
            {
                RenderBlock(block, ref context, ref graphics, newPageAction, true, true);
            }

            graphics.Translate(-minX - this.IndentWidth, 0);
            context.Translation = new Point(context.Translation.X - minX - this.IndentWidth, context.Translation.Y);

            prevContext.Cursor = context.Cursor;
            prevContext.BottomRight = context.BottomRight;
            prevContext.CurrentPage = context.CurrentPage;
            prevContext.CurrentLine = context.CurrentLine;

            context = prevContext;
        }

        private void RenderQuoteBlock(QuoteBlock quote, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            MarkdownContext prevContext = context.Clone();

            double minX = context.GetMinX(context.Cursor.Y + SpaceBeforeParagaph * context.Font.FontSize, context.Cursor.Y - context.Font.Descent + context.Font.Ascent + SpaceBeforeParagaph * context.Font.FontSize);
```

```

            Graphics quoteGraphics = new Graphics();

            quoteGraphics.Translate(minX + this.QuoteBlockIndentWidth, 0);
            context.Translation = new Point(minX + this.QuoteBlockIndentWidth, 0);
            context.MarginBottomRight = new Point(context.MarginBottomRight.X + prevContext.Translation.X, context.MarginBottomRight.Y + prevContext.Translation.Y);

            double maxX = this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X;

            double startY = context.Cursor.Y + context.Font.Ascent - context.Font.YMax;

            Point currTranslation = context.Translation;

            Graphics parentGraphics = graphics;

            NewPageAction newPageActionWithBlockquotes = (ref MarkdownContext currContext, ref Graphics currGraphics) =>
            {
                double currEndY = currContext.Cursor.Y;

                double currMaxX = maxX;

                Graphics currBackgroundGraphics = new Graphics();

                currBackgroundGraphics.Save();

                currBackgroundGraphics.Translate(currContext.Translation);
                currBackgroundGraphics.FillRectangle(new Point(-this.QuoteBlockIndentWidth + this.QuoteBlockBarWidth, startY), new Size(currMaxX + this.QuoteBlockIndentWidth - this.QuoteBlockBarWidth, currEndY - startY), this.QuoteBlockBackgroundColour, tag: currContext.Tag);
                currBackgroundGraphics.FillRectangle(new Point(-this.QuoteBlockIndentWidth, startY), new Size(this.QuoteBlockBarWidth, currEndY - startY), this.QuoteBlockBarColour, tag: currContext.Tag);

                currBackgroundGraphics.Restore();

                currBackgroundGraphics.DrawGraphics(0, 0, currGraphics);

                parentGraphics.DrawGraphics(0, 0, currBackgroundGraphics);

                Point currContextTranslation = currContext.Translation;

                currContext.Translation = prevContext.Translation;

                newPageAction(ref currContext, ref parentGraphics);

                currContext.Translation = currContextTranslation;

                currGraphics = new Graphics();
                currGraphics.Translate(minX + this.QuoteBlockIndentWidth, 0);

                startY = currContext.Cursor.Y + currContext.Font.Ascent - currContext.Font.YMax;
            };

            int index = 0;

            foreach (Block block in quote)
            {
                RenderBlock(block, ref context, ref quoteGraphics, newPageActionWithBlockquotes, true, index < quote.Count - 1);
                index++;
            }

            double endY = context.Cursor.Y;

            Graphics backgroundGraphics = new Graphics();

            backgroundGraphics.Save();

            backgroundGraphics.Translate(context.Translation);
            backgroundGraphics.FillRectangle(new Point(-this.QuoteBlockIndentWidth + this.QuoteBlockBarWidth, startY), new Size(maxX + this.QuoteBlockIndentWidth - this.QuoteBlockBarWidth, endY - startY), this.QuoteBlockBackgroundColour, tag: context.Tag);
            backgroundGraphics.FillRectangle(new Point(-this.QuoteBlockIndentWidth, startY), new Size(this.QuoteBlockBarWidth, endY - startY), this.QuoteBlockBarColour, tag: context.Tag);

            backgroundGraphics.Restore();

            backgroundGraphics.DrawGraphics(0, 0, quoteGraphics);

            parentGraphics.DrawGraphics(0, 0, backgroundGraphics);

            graphics = parentGraphics;

            context.Translation = prevContext.Translation;
```

```

            if (!(quote.Parent is QuoteBlock) || quote.Parent.LastChild != quote)
            {
                context.Cursor = new Point(context.Cursor.X, context.Cursor.Y + SpaceAfterParagraph * context.Font.FontSize);
            }

            prevContext.Cursor = context.Cursor;
            prevContext.BottomRight = context.BottomRight;
            prevContext.CurrentPage = context.CurrentPage;
            prevContext.CurrentLine = context.CurrentLine;

            context = prevContext;
        }

        private void RenderListItemBlock(ListItemBlock listItem, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            MarkdownContext prevContext = context.Clone();

            bool isLoose = false;

            double startX = context.Cursor.X;
            double startY = context.Cursor.Y;

            if (listItem.Parent is ListBlock list)
            {
                if (list.IsOrdered)
                {
                    if (context.CurrentLine == null)
                    {
                        context.CurrentLine = new Line(context.Font.Ascent);
                    }

                    string bullet;

                    switch (list.BulletType)
                    {
                        case 'a':
                        case 'A':
                            bullet = ((char)((int)list.DefaultOrderedStart[0] + listItem.Order - 1)).ToString() + list.OrderedDelimiter;
                            break;
                        case 'i':
                            bullet = GetRomanNumeral(listItem.Order).ToLower() + list.OrderedDelimiter;
                            break;
                        case 'I':
                            bullet = GetRomanNumeral(listItem.Order) + list.OrderedDelimiter;
                            break;
                        default:
                            bullet = listItem.Order.ToString() + list.OrderedDelimiter;
                            break;
                    }

                    if (list.IsLoose)
                    {
                        isLoose = true;
                        context.CurrentLine.Fragments.Add(new TextFragment(new Point(context.Cursor.X - context.Font.MeasureText(bullet).Width - this.IndentWidth * 0.15, context.Cursor.Y + context.Font.Ascent + SpaceBeforeParagaph * context.Font.FontSize), bullet, context.Font, context.Colour, context.Tag));
                    }
                    else
                    {
                        isLoose = false;
                        context.CurrentLine.Fragments.Add(new TextFragment(new Point(context.Cursor.X - context.Font.MeasureText(bullet).Width - this.IndentWidth * 0.15, context.Cursor.Y + context.Font.Ascent), bullet, context.Font, context.Colour, context.Tag));
                    }
                }
                else
                {
                    if (listItem.Count > 0 && listItem[0] is ParagraphBlock paragraph && paragraph.Inline?.FirstChild is Markdig.Extensions.TaskLists.TaskList task)
                    {
                        if (context.CurrentLine == null)
                        {
                            context.CurrentLine = new Line(context.Font.Ascent);
                        }

                        Graphics bullet = new Graphics();
                        bullet.Scale(context.Font.FontSize, context.Font.FontSize);

                        if (task.Checked)
                        {
                            bullet.DrawGraphics(0, 0, this.TaskListCheckedBullet);
```

```
                        }
                        else
                        {
                            bullet.DrawGraphics(0, 0, this.TaskListUncheckedBullet);
                        }

                        if (list.IsLoose)
                        {
                            isLoose = true;
                            context.CurrentLine.Fragments.Add(new GraphicsFragment(new Point(context.Cursor.X - this.IndentWidth * 0.15, context.Cursor.Y + context.Font.Ascent + SpaceBeforeParagaph * context.Font.FontSize - context.Font.Descent * 0.5 - (context.Font.Ascent - context.Font.Descent) * 0.5), bullet, 0));
                        }
                        else
                        {
                            isLoose = false;
                            context.CurrentLine.Fragments.Add(new GraphicsFragment(new Point(context.Cursor.X - this.IndentWidth * 0.15, context.Cursor.Y + context.Font.Ascent - context.Font.Descent * 0.5 - (context.Font.Ascent - context.Font.Descent) * 0.5), bullet, 0));
                        }
                    }
                    else
                    {
                        if (context.CurrentLine == null)
                        {
                            context.CurrentLine = new Line(context.Font.Ascent);
                        }

                        Graphics bullet = new Graphics();
                        bullet.Scale(context.Font.FontSize, context.Font.FontSize);
                        this.Bullets[(context.ListDepth - 1) % this.Bullets.Count](bullet, context.Colour);

                        if (list.IsLoose)
                        {
                            isLoose = true;
                            context.CurrentLine.Fragments.Add(new GraphicsFragment(new Point(context.Cursor.X - this.IndentWidth * 0.15, context.Cursor.Y + context.Font.Ascent + SpaceBeforeParagaph * context.Font.FontSize - context.Font.Descent * 0.5 - (context.Font.Ascent - context.Font.Descent) * 0.5), bullet, 0));
                        }
                        else
                        {
                            isLoose = false;
                            context.CurrentLine.Fragments.Add(new GraphicsFragment(new Point(context.Cursor.X - this.IndentWidth * 0.15, context.Cursor.Y + context.Font.Ascent - context.Font.Descent * 0.5 - (context.Font.Ascent - context.Font.Descent) * 0.5), bullet, 0));
                        }
                    }
                }
            }

            foreach (Block block in listItem)
            {
                RenderBlock(block, ref context, ref graphics, newPageAction, isLoose || listItem == listItem.Parent[0], isLoose || listItem == listItem.Parent.LastChild);
            }

            prevContext.Cursor = context.Cursor;
            prevContext.BottomRight = context.BottomRight;
            prevContext.CurrentPage = context.CurrentPage;
            prevContext.CurrentLine = context.CurrentLine;

            context = prevContext;
        }

        private void RenderHTMLBlock(string html, bool isInline, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction, bool spaceBefore, bool spaceAfter)
        {
            if (!isInline)
            {
                double minX = context.GetMinX(context.Cursor.Y + SpaceBeforeParagaph * context.Font.FontSize, context.Cursor.Y + context.Font.Ascent + SpaceBeforeParagaph * context.Font.FontSize - context.Font.Descent);

                context.Cursor = new Point(minX, context.Cursor.Y + context.Font.Ascent + SpaceBeforeParagaph * context.Font.FontSize);

                if (context.CurrentLine == null)
                {
                    context.CurrentLine = new Line(context.Font.Ascent);
                }
            }

            foreach (HtmlTag tag in HtmlTag.Parse(html))
            {
```

```
                if (tag.Tag.Equals("img", StringComparison.OrdinalIgnoreCase) || tag.Tag.Equals("image", StringComparison.OrdinalIgnoreCase))
                {
                    RenderHTMLImage(tag, isInline, ref context, ref graphics, newPageAction);
                }
                else if (tag.Tag.Equals("br", StringComparison.OrdinalIgnoreCase))
                {
                    RenderLineBreakInline(true, tag.Attributes.TryGetValue("type", out string typeValue) && typeValue.Equals("page", StringComparison.OrdinalIgnoreCase) && this.AllowPageBreak, ref context, ref graphics, newPageAction);
                }
                else if (tag.Tag.Equals("a"))
                {
                    if (tag.Attributes.TryGetValue("name", out string anchorName))
                    {
                        if (!context.InternalAnchors.TryGetValue("#" + anchorName, out string anchor))
                        {
                            anchor = Guid.NewGuid().ToString("N");
                            context.InternalAnchors["#" + anchorName] = anchor;
                        }

                        if (context.CurrentLine == null)
                        {
                            context.CurrentLine = new Line(0);
                        }

                        double anchorHeight = this.BaseFontSize * Math.Max(1, this.HeaderFontSizeMultipliers.Max());

                        context.CurrentLine.Fragments.Add(new RectangleFragment(new Point(context.Cursor.X, context.Cursor.Y - anchorHeight), new Size(10, anchorHeight), Colour.FromRgba(0, 0, 0, 0), anchor));
                    }
                }
                else
                {

                }
            }

            if (!isInline)
            {
                if (context.CurrentLine != null)
                {
                    context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
                    context.CurrentLine = null;

                    context.Cursor = new Point(0, context.Cursor.Y - context.Font.Descent + SpaceAfterLine * context.Font.FontSize);

                    context.Cursor = new Point(0, context.Cursor.Y + SpaceAfterParagraph * context.Font.FontSize);
                }
            }
        }

        private void RenderHTMLImage(HtmlTag imgTag, bool isInline, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            if (imgTag != null)
            {
                if (imgTag.Attributes.TryGetValue("src", out string imageSrc))
                {
                    (string imageFile, bool wasDownloaded) = this.ImageUriResolver(imageSrc, this.BaseImageUri);

                    Page imagePage = null;

                    if (imageFile != null)
                    {
                        if (System.IO.Path.GetExtension(imageFile) == ".svg")
                        {
                            try
                            {
                                imagePage = VectSharp.SVG.Parser.FromFile(imageFile);
                            }
                            catch
                            {
                                imagePage = null;
                            }
                        }
                        else if (RasterImageLoader != null)
                        {
                            try
                            {
                                RasterImage raster = RasterImageLoader(imageFile);
```

```
                                imagePage = new Page(raster.Width, raster.Height);
                                imagePage.Graphics.DrawRasterImage(0, 0, raster, tag: context.Tag);
                            }
                            catch
                            {
                                imagePage = null;
                            }
                        }


                        if (wasDownloaded)
                        {
                            System.IO.File.Delete(imageFile);
                            System.IO.Directory.Delete(System.IO.Path.GetDirectoryName(imageFile));
                        }
                    }
                    else if (imageSrc.StartsWith("data:"))
                    {
                        try
                        {
                            imagePage = VectSharp.SVG.Parser.ParseImageURI(imageSrc, false);
                        }
                        catch
                        {

                            imagePage = null;
                        }
                    }

                    if (imagePage != null)
                    {
                        double scaleX = 1;
                        double scaleY = 1;

                        bool hasWidth = false;
                        bool hasHeight = false;

                        if (imgTag.Attributes.TryGetValue("width", out string widthString) && double.TryParse(widthString, System.Globalization.NumberStyles.Any, System.Globalization.CultureInfo.InvariantCulture, out double width))
                        {
                            hasWidth = true;
                            scaleX = width * this.ImageUnitMultiplier / imagePage.Width;
                        }

                        if (imgTag.Attributes.TryGetValue("height", out string heightString) && double.TryParse(heightString, System.Globalization.NumberStyles.Any, System.Globalization.CultureInfo.InvariantCulture, out double height))
                        {
                            hasHeight = true;
                            scaleY = height * this.ImageUnitMultiplier / imagePage.Height;
                        }

                        if (hasWidth && !hasHeight)
                        {
                            scaleY = scaleX;
                        }
                        else if (hasHeight && !hasWidth)
                        {
                            scaleX = scaleY;
                        }

                        scaleX *= this.ImageMultiplier;
                        scaleY *= this.ImageMultiplier;

                        if (!isInline)
                        {
                            string alignValue;

                            if (!imgTag.Attributes.TryGetValue("align", out alignValue))
                            {
                                alignValue = null;
                            }

                            if (alignValue == "center")
                            {
                                if (context.CurrentLine != null)
                                {
                                    context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
                                    context.CurrentLine = null;
                                }

                                double minX = context.GetMinX(context.Cursor.Y + context.Translation.Y, context.Cursor.Y + context.Translation.Y + scaleY * imagePage.Height);
                                double maxX = context.GetMaxX(context.Cursor.Y +
```

```
context.Translation.Y, context.Cursor.Y + context.Translation.Y + scaleY * imagePage.Height, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X);

                                if (scaleX * imagePage.Width > maxX - minX)
                                {
                                    scaleX = (maxX - minX) / imagePage.Width;
                                    scaleY = scaleX;
                                }

                                double finalY = context.Cursor.Y + scaleY * imagePage.Height;

                                if (finalY + context.Translation.Y > this.PageSize.Height - this.Margins.Bottom + this.ImageMarginTolerance - context.MarginBottomRight.Y)
                                {
                                    newPageAction(ref context, ref graphics);
                                    finalY = context.Cursor.Y + scaleY * imagePage.Height;
                                }

                                graphics.Save();
                                graphics.Translate((minX + maxX - scaleX * imagePage.Width) * 0.5, context.Cursor.Y);
                                graphics.Scale(scaleX, scaleY);
                                graphics.SetClippingPath(0, 0, imagePage.Width, imagePage.Height);
                                graphics.DrawGraphics(0, 0, imagePage.Graphics);

                                graphics.Restore();

                                context.Cursor = new Point(0, finalY + SpaceAfterParagraph * context.Font.FontSize + SpaceAfterLine * context.Font.FontSize);
                            }
                            else if (alignValue == "right")
                            {
                                if (context.CurrentLine != null)
                                {
                                    context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
                                    context.CurrentLine = null;
                                }

                                double finalY = context.Cursor.Y + scaleY * imagePage.Height;

                                if (finalY + context.Translation.Y > this.PageSize.Height - this.Margins.Bottom + this.ImageMarginTolerance - context.MarginBottomRight.Y)
                                {
                                    newPageAction(ref context, ref graphics);
                                    finalY = context.Cursor.Y + scaleY * imagePage.Height;
                                }

                                graphics.Save();
                                graphics.Translate(this.PageSize.Width - this.Margins.Right - context.Translation.X - scaleX * imagePage.Width - context.MarginBottomRight.X, context.Cursor.Y);
                                graphics.Scale(scaleX, scaleY);
                                graphics.SetClippingPath(0, 0, imagePage.Width, imagePage.Height);

                                graphics.DrawGraphics(0, 0, imagePage.Graphics);

                                graphics.Restore();

                                context.ForbiddenAreasRight.Add((this.PageSize.Width - this.Margins.Right - scaleX * imagePage.Width - this.ImageSideMargin - context.MarginBottomRight.X, context.Cursor.Y + context.Translation.Y, context.Cursor.Y + context.Translation.Y + scaleY * imagePage.Height));
                            }
                            else if (alignValue == "left")
                            {
                                if (context.CurrentLine != null)
                                {
                                    context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
                                    context.CurrentLine = null;
                                }

                                double finalY = context.Cursor.Y + scaleY * imagePage.Height;

                                if (finalY + context.Translation.Y > this.PageSize.Height - this.Margins.Bottom + this.ImageMarginTolerance - context.MarginBottomRight.Y)
                                {
                                    newPageAction(ref context, ref graphics);
                                    finalY = context.Cursor.Y + scaleY * imagePage.Height;
                                }

                                graphics.Save();
                                graphics.Translate(0, context.Cursor.Y);
                                graphics.Scale(scaleX, scaleY);
                                graphics.SetClippingPath(0, 0, imagePage.Width, imagePage.Height);
```

```
                                graphics.DrawGraphics(0, 0, imagePage.Graphics);

                                graphics.Restore();

                                context.ForbiddenAreasLeft.Add((scaleX * imagePage.Width + context.Translation.X + this.ImageSideMargin, context.Cursor.Y + context.Translation.Y, context.Cursor.Y + context.Translation.Y + scaleY * imagePage.Height));
                            }
                            else
                            {
                                isInline = true;
                            }
                        }

                        if (isInline)
                        {
                            Graphics scaledImage = new Graphics();
                            scaledImage.Scale(scaleX, scaleY);
                            scaledImage.DrawGraphics(0, 0, imagePage.Graphics);

                            if (context.CurrentLine == null)
                            {
                                context.CurrentLine = new Line(context.Font.Ascent);

                                context.Cursor = new Point(0, context.Cursor.Y);
                                double minX = context.GetMinX(context.Cursor.Y - scaleY * imagePage.Height, context.Cursor.Y);
                                context.Cursor = new Point(minX, context.Cursor.Y);
                            }

                            double currLineMaxX = context.GetMaxX(context.Cursor.Y - scaleY * imagePage.Height, context.Cursor.Y, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X);

                            double finalX = context.Cursor.X + imagePage.Width;

                            if (finalX > currLineMaxX)
                            {
                                context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);

                                context.CurrentLine = new Line(context.Font.Ascent);
                                context.Cursor = new Point(0, context.Cursor.Y - context.Font.Descent + SpaceAfterLine * context.Font.FontSize + context.Font.Ascent);
                                currLineMaxX = context.GetMaxX(context.Cursor.Y - context.Font.Ascent, context.Cursor.Y - context.Font.Descent, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X);

                                double minX = context.GetMinX(context.Cursor.Y - scaleY * imagePage.Height * scaleY, context.Cursor.Y);

                                context.Cursor = new Point(minX, context.Cursor.Y);
                            }

                            context.CurrentLine.Fragments.Add(new GraphicsFragment(new Point(context.Cursor.X, context.Cursor.Y - scaleY * imagePage.Height), scaledImage, scaleY * imagePage.Height));
                            context.Cursor = new Point(context.Cursor.X + scaleX * imagePage.Width, context.Cursor.Y);
                        }
                    }
                }
            }
        }

        private void RenderTable(Table table, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            if (table.Count > 0)
            {
                if (!table.IsValid())
                {
                    table.NormalizeUsingMaxWidth();
                    table.NormalizeUsingHeaderRow();
                }

                if (table.IsValid() && table.ColumnDefinitions?.Count > 0)
                {
                    bool isGridTable = false;

                    foreach (TableColumnDefinition def in table.ColumnDefinitions)
                    {
                        if (def.Width > 0)
                        {
                            isGridTable = true;
                            break;
```

```
                        }
                    }

                    if (isGridTable)
                    {
                        int maxColumns = 0;

                        foreach (TableRow row in table)
                        {
                            maxColumns = Math.Max(maxColumns, ((TableCell)row.Last()).ColumnIndex + 1);
                        }

                        if (table.ColumnDefinitions.Count > maxColumns)
                        {
                            table.ColumnDefinitions.RemoveRange(maxColumns, table.ColumnDefinitions.Count - maxColumns);
                        }
                    }
                    else
                    {
                        int maxColumns = 0;

                        foreach (TableRow row in table)
                        {
                            maxColumns = Math.Max(maxColumns, row.Count);
                        }

                        if (table.ColumnDefinitions.Count > maxColumns)
                        {
                            table.ColumnDefinitions.RemoveRange(maxColumns, table.ColumnDefinitions.Count - maxColumns);
                        }
                    }

                    double[] columnWidths = new double[table.ColumnDefinitions.Count];

                    if (table.ColumnDefinitions.Count == 0)
                    {
                        int columnCount = 0;
                        foreach (TableRow row in table)
                        {
                            columnCount = Math.Max(row.Count, columnCount);
                        }

                        columnWidths = new double[columnCount];
                    }

                    int missingColumns = columnWidths.Length;

                    for (int i = 0; i < columnWidths.Length; i++)
                    {
                        columnWidths[i] = double.NaN;
                    }

                    double remainingPerc = 1;

                    for (int i = 0; i < table.ColumnDefinitions.Count; i++)
                    {
                        if (table.ColumnDefinitions[i].Width > 0)
                        {
                            missingColumns--;
                            remainingPerc -= table.ColumnDefinitions[i].Width / 100.0;
                            columnWidths[i] = table.ColumnDefinitions[i].Width / 100.0;
                        }
                    }

                    if (missingColumns > 0)
                    {
                        remainingPerc /= missingColumns;
                        for (int i = 0; i < columnWidths.Length; i++)
                        {
                            if (double.IsNaN(columnWidths[i]))
                            {
                                columnWidths[i] = remainingPerc;
                            }
                        }
                    }

                    double maxX = context.GetMaxX(context.Cursor.Y, context.Cursor.Y, this.PageSize.Width - this.Margins.Right - context.Translation.X - context.MarginBottomRight.X);

                    for (int i = 0; i < columnWidths.Length; i++)
                    {
                        columnWidths[i] *= maxX;
                    }

```

```
                    int index = 0;

                    foreach (TableRow row in table)
                    {
                        RenderTableRow(row, columnWidths, index == table.Count - 1, ref context, ref graphics, newPageAction);
                        index++;
                    }

                    context.Cursor = new Point(context.Cursor.X, context.Cursor.Y + SpaceAfterParagraph * context.Font.FontSize);
                }
            }
        }

        private void RenderTableRow(TableRow row, double[] columnWidths, bool isLastRow, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            if (context.CurrentLine == null)
            {
                context.CurrentLine = new Line(0);
            }

            int index = 0;

            foreach (TableCell cell in row)
            {
                if (cell.ColumnIndex < 0)
                {
                    cell.ColumnIndex = index;
                    index += cell.ColumnSpan;
                }
                else
                {
                    index = cell.ColumnIndex + cell.ColumnSpan;
                }
            }

            double maxHeight = 0;
            double startX = context.Cursor.X;

            MarkdownContext prevContext = context.Clone();

            if (row.IsHeader)
            {
                context.Font = new Font(this.BoldFontFamily, context.Font.FontSize);
            }


            foreach (TableCell cell in row)
            {
                double cellWidth = 0;

                for (int i = 0; i < cell.ColumnSpan && cell.ColumnIndex + i < columnWidths.Length; i++)
                {
                    cellWidth += columnWidths[cell.ColumnIndex + i];
                }

                Page cellPage = RenderTableCell(cell, cellWidth, ref context, ref graphics, newPageAction);

                double prevMaxHeight = maxHeight;
                maxHeight = Math.Max(maxHeight, cellPage.Height);

                context.CurrentLine.Fragments.Add(new GraphicsFragment(new Point(context.Cursor.X, context.Cursor.Y - cellPage.Height), cellPage.Graphics, cellPage.Height));

                context.Cursor = new Point(context.Cursor.X + cellWidth, context.Cursor.Y);
            }

            if (this.TableVAlign == VerticalAlignment.Top)
            {
                for (int i = 0; i < context.CurrentLine.Fragments.Count; i++)
                {
                    context.CurrentLine.Fragments[i].Translate(0, -maxHeight + ((GraphicsFragment)context.CurrentLine.Fragments[i]).Ascent);
                }
            }
            else if (this.TableVAlign == VerticalAlignment.Middle)
            {
                for (int i = 0; i < context.CurrentLine.Fragments.Count; i++)
                {
                    context.CurrentLine.Fragments[i].Translate(0, (-maxHeight + ((GraphicsFragment)context.CurrentLine.Fragments[i]).Ascent) * 0.5);
                }
            }
```

```

            context.CurrentLine.Fragments.Add(new UnderlineFragment(new Point(startX, context.Cursor.Y), new Point(columnWidths.Sum() + startX, context.Cursor.Y), row.IsHeader ? this.TableHeaderRowSeparatorColour : this.TableRowSeparatorColour, row.IsHeader ? this.TableHeaderRowSeparatorThickness : this.TableHeaderSeparatorThickness, context.Tag));

            context.CurrentLine.Render(ref graphics, ref context, newPageAction, this.PageSize.Height - this.Margins.Bottom - context.Translation.Y - context.MarginBottomRight.Y);
            context.CurrentLine = null;

            context.Cursor = new Point(startX, context.Cursor.Y);

            context.Cursor = new Point(startX, context.Cursor.Y + SpaceAfterLine * context.Font.FontSize + (row.IsHeader ? this.TableHeaderRowSeparatorThickness : this.TableHeaderSeparatorThickness));

            prevContext.Cursor = context.Cursor;
            prevContext.BottomRight = context.BottomRight;
            prevContext.CurrentPage = context.CurrentPage;
            prevContext.CurrentLine = context.CurrentLine;

            context = prevContext;
        }

        private Page RenderTableCell(TableCell cell, double cellWidth, ref MarkdownContext context, ref Graphics graphics, NewPageAction newPageAction)
        {
            MarkdownRenderer clonedRenderer = this.Clone();
            clonedRenderer.PageSize = new Size(cellWidth, double.PositiveInfinity);
            clonedRenderer.Margins = this.TableCellMargins;

            MarkdownContext clonedContext = context.Clone();
            clonedContext.Translation = new Point(0, 0);
            clonedContext.Cursor = new Point(0, 0);
            clonedContext.BottomRight = new Point(0, 0);
            clonedContext.CurrentLine = null;
            clonedContext.CurrentPage = null;
            clonedContext.ForbiddenAreasLeft = new List<(double MinX, double MinY, double MaxY)>();
            clonedContext.ForbiddenAreasRight = new List<(double MinX, double MinY, double MaxY)>();

            Page cellPage = clonedRenderer.RenderSubDocument(cell, ref clonedContext).Pages[0];

            cellPage.Crop(new Point(0, 0), new Size(cellWidth, clonedContext.BottomRight.Y + this.TableCellMargins.Bottom));

            return cellPage;
        }

        static (int, string)[] RomanNumbers = new (int, string)[] { (1000, "M"), (900, "CM"), (500, "D"), (400, "CD"), (100, "C"), (90, "XC"), (50, "L"), (40, "XL"), (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I") };

        private static string GetRomanNumeral(int number)
        {
            StringBuilder tbr = new StringBuilder();

            for (int i = 0; i < RomanNumbers.Length; i++)
            {
                while (number >= RomanNumbers[i].Item1)
                {
                    tbr.Append(RomanNumbers[i].Item2);
                    number -= RomanNumbers[i].Item1;
                }
            }

            return tbr.ToString();
        }
    }
}
```

8.11 SyntaxHighlighting.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
```

```
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using Highlight;
using Highlight.Engines;
using Highlight.Patterns;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;

namespace VectSharp.Markdown
{
    /// <summary>
    /// Represents a string with associated formatting information.
    /// </summary>
    public struct FormattedString
    {
        /// <summary>
        /// The text represented by this object.
        /// </summary>
        public string Text { get; }

        /// <summary>
        /// The colour of the text.
        /// </summary>
        public Colour Colour { get; }

        /// <summary>
        /// Whether the text should be rendered as bold or not.
        /// </summary>
        public bool IsBold { get; }

        /// <summary>
        /// Whether the text should be rendered as italic or not.
        /// </summary>
        public bool IsItalic { get; }

        /// <summary>
        /// Creates a new <see cref="FormattedString"/> instance.
        /// </summary>
        /// <param name="text">The text of the object.</param>
        /// <param name="colour">The colour of the text.</param>
        /// <param name="isBold">Whether the text should be rendered as bold or not.</param>
        /// <param name="isItalic">Whether the text should be rendered as italic or not.</param>
        public FormattedString(string text, Colour colour, bool isBold, bool isItalic)
        {
            this.Text = text;
            this.Colour = colour;
            this.IsBold = isBold;
            this.IsItalic = isItalic;
        }
    }

    /// <summary>
    /// Contains methods to perform syntax highlighting.
    /// </summary>
    public static class SyntaxHighlighter
    {
        private static Dictionary<string, string[]> LanguageAliases = new Dictionary<string, string[]>()
        {
            { "ASPX", new string[] { "ASP.NET", "aspx", "aspx-vb" } },
            { "C", new string[] { } },
            { "C++", new string[] { "cpp" } },
            { "C#", new string[] { "csharp" } },
            { "COBOL", new string[] { } },
            { "Eiffel", new string[] { } },
            { "Fortran", new string[] { } },
            { "Haskell", new string[] { } },
            { "HTML", new string[] { "xhtml" } },
            { "Java", new string[] { } },
            { "JavaScript", new string[] { "js", "node" } },
            { "Mercury", new string[] { } },
            { "MSIL", new string[] { } },
            { "Pascal", new string[] { "delphi", "objectpascal" } },
            { "Perl", new string[] { "cperl" } },
            { "PHP", new string[] { "inc" } },
            { "Python", new string[] { "python3", "rusthon" } },
            { "Ruby", new string[] { "jruby", "macruby", "rake", "rb", "rbx" } },
            { "SQL", new string[] { } },
            { "Visual Basic", new string[] { "vba", "vb6", "visual basic 6", "visual basic for applications" } },
```

```
            { "VBScript", new string[] { } },
            { "VB.NET", new string[] { "Visual Basic .NET", "vbnet", "vb .net" } },
            { "XML", new string[] { "rss", "xsd", "wsdl" } },
        };

        private static string GetLanguage(string language)
        {
            foreach (KeyValuePair<string, string[]> element in LanguageAliases)
            {
                if (element.Key.Equals(language, StringComparison.OrdinalIgnoreCase))
                {
                    return element.Key;
                }

                foreach (string alias in element.Value)
                {
                    if (alias.Equals(language, StringComparison.OrdinalIgnoreCase))
                    {
                        return element.Key;
                    }
                }
            }

            return null;
        }

        /// <summary>
        /// Performs syntax highlighting for a specified language on some source code.
        /// </summary>
        /// <param name="sourceCode">The source code to be highlighted.</param>
        /// <param name="language">The name of the language to use for the highlighting.</param>
        /// <returns>A list of lists of <see cref="FormattedString"/>s. Each list of <see cref="FormattedString"/>s represents a line.</returns>
        public static List<List<FormattedString» GetSyntaxHighlightedLines(string sourceCode, string language)
        {
            language = GetLanguage(language);

            if (string.IsNullOrEmpty(language))
            {
                return null;
            }

            HighlighterEngine engine = new HighlighterEngine();

            Highlighter highlighter = new Highlighter(engine);

            highlighter.Highlight(language, sourceCode);

            List<List<FormattedString» tbr = new List<List<FormattedString»();

            List<FormattedString> currentLine = new List<FormattedString>();

            for (int i = 0; i < engine.HighlightedSpans.Count; i++)
            {
                string[] split = engine.HighlightedSpans[i].Text.Replace("\r", "").Split('\n');

                for (int j = 0; j < split.Length; j++)
                {
                    currentLine.Add(new FormattedString(split[j], engine.HighlightedSpans[i].Colour, engine.HighlightedSpans[i].IsBold, engine.HighlightedSpans[i].IsItalic));

                    if (j < split.Length - 1)
                    {
                        tbr.Add(currentLine);
                        currentLine = new List<FormattedString>();
                    }
                }
            }

            tbr.Add(currentLine);

            return tbr;
        }
    }

    internal class HighlighterEngine : Engine, IEngine
    {
        private const RegexOptions DefaultRegexOptions = RegexOptions.ExplicitCapture | RegexOptions.IgnorePatternWhitespace;

        public List<FormattedString> HighlightedSpans { get; } = new List<FormattedString>();

        protected override string ProcessBlockPatternMatch(Definition definition, BlockPattern pattern, Match match)
        {
            if (!string.IsNullOrEmpty(pattern.RawRegex))
```

```
            {
                Match reMatch = Regex.Match(match.Value, pattern.RawRegex);

                if (reMatch.Groups.Count > 1)
                {
                    HighlightedSpans.Add(new FormattedString(match.Value.Substring(0, reMatch.Groups[1].Index), Colours.Black, false, false));
                    HighlightedSpans.Add(new FormattedString(reMatch.Groups[1].Value, pattern.Style.Colors.ForeColor, pattern.Style.Font.IsBold, pattern.Style.Font.IsItalic));
                    HighlightedSpans.Add(new FormattedString(match.Value.Substring(reMatch.Groups[1].Index + reMatch.Groups[1].Length), Colours.Black, false, false));
                }
                else
                {
                    HighlightedSpans.Add(new FormattedString(match.Value, pattern.Style.Colors.ForeColor, pattern.Style.Font.IsBold, pattern.Style.Font.IsItalic));
                }
            }
            else
            {
                HighlightedSpans.Add(new FormattedString(match.Value, pattern.Style.Colors.ForeColor, pattern.Style.Font.IsBold, pattern.Style.Font.IsItalic));
            }
            return match.Value;
        }

        protected override string ProcessMarkupPatternMatch(Definition definition, MarkupPattern pattern, Match match)
        {
            HighlightedSpans.Add(new FormattedString(match.Value, pattern.Style.Colors.ForeColor, pattern.Style.Font.IsBold, pattern.Style.Font.IsItalic));
            return match.Value;
        }

        protected override string ProcessWordPatternMatch(Definition definition, WordPattern pattern, Match match)
        {
            HighlightedSpans.Add(new FormattedString(match.Value, pattern.Style.Colors.ForeColor, pattern.Style.Font.IsBold, pattern.Style.Font.IsItalic));
            return match.Value;
        }

        string IEngine.Highlight(Definition definition, string input)
        {
            if (definition == null)
            {
                throw new ArgumentNullException("definition");
            }

            var output = PreHighlight(definition, input);
            output = HighlightUsingRegex(definition, output);
            output = PostHighlight(definition, output);

            return output;
        }

        private string HighlightUsingRegex(Definition definition, string input)
        {
            var regexOptions = GetRegexOptions(definition);
            var evaluator = GetMatchEvaluator(definition);
            var regexPattern = definition.GetRegexPattern();

            int currentIndex = 0;

            foreach (Match match in Regex.Matches(input, regexPattern))
            {
                if (match.Index > currentIndex)
                {
                    this.HighlightedSpans.Add(new FormattedString(input.Substring(currentIndex, match.Index - currentIndex), Colours.Black, false, false));
                }

                currentIndex = match.Index + match.Length;

                evaluator(match);
            }

            if (currentIndex < input.Length)
            {
                this.HighlightedSpans.Add(new FormattedString(input.Substring(currentIndex, input.Length - currentIndex), Colours.Black, false, false));
            }

            return null;
        }

```

```
        private RegexOptions GetRegexOptions(Definition definition)
        {
            if (definition.CaseSensitive)
            {
                return DefaultRegexOptions | RegexOptions.IgnoreCase;
            }

            return DefaultRegexOptions;
        }

        private string ElementMatchHandler(Definition definition, Match match)
        {
            if (definition == null)
            {
                throw new ArgumentNullException("definition");
            }
            if (match == null)
            {
                throw new ArgumentNullException("match");
            }

            var pattern = definition.Patterns.First(x => match.Groups[x.Key].Success).Value;
            if (pattern != null)
            {
                if (pattern is BlockPattern)
                {
                    return ProcessBlockPatternMatch(definition, (BlockPattern)pattern, match);
                }
                if (pattern is MarkupPattern)
                {
                    return ProcessMarkupPatternMatch(definition, (MarkupPattern)pattern, match);
                }
                if (pattern is WordPattern)
                {
                    return ProcessWordPatternMatch(definition, (WordPattern)pattern, match);
                }
            }

            return match.Value;
        }

        private MatchEvaluator GetMatchEvaluator(Definition definition)
        {
            return match => ElementMatchHandler(definition, match);
        }
    }
}
```

8.12 Word.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Text;

namespace VectSharp.Markdown
{
    internal class Word
    {
        public char PrecedingWhitespace { get; }
        public int WhitespaceCount { get; }
        public string Text { get; }

        public Font.DetailedFontMetrics Metrics { get; }

        private Word(char precedingWhitespace, int whitespaceCount, string text, Font font)
        {
```

```
            this.PrecedingWhitespace = precedingWhitespace;
            this.WhitespaceCount = whitespaceCount;
            this.Text = text;

            if (!string.IsNullOrEmpty(text))
            {
                this.Metrics = font.MeasureTextAdvanced(text);
            }
            else
            {
                this.Metrics = font.MeasureTextAdvanced(" ");
            }
        }

        private static Dictionary<FontFamily, double> SpaceWidths { get; } = new Dictionary<FontFamily, double>();

        private static IEnumerable<Word> SplitWord(Word word, Font font, double maxWidth)
        {
            if (!SpaceWidths.TryGetValue(font.FontFamily, out double spaceWidth))
            {
                spaceWidth = font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(' ') / 1000.0;
                SpaceWidths[font.FontFamily] = spaceWidth;
            }

            spaceWidth *= font.FontSize;

            while (!string.IsNullOrEmpty(word.Text) && word.Metrics.Width + word.Metrics.LeftSideBearing + word.Metrics.RightSideBearing + spaceWidth * word.WhitespaceCount * (word.PrecedingWhitespace == '\t' ? 4 : 1) > maxWidth)
            {
                int minIndex = 1;
                int maxIndex = word.Text.Length;

                while (maxIndex - minIndex > 1)
                {
                    int index = (minIndex + maxIndex) / 2;

                    double width = font.MeasureText(word.Text.Substring(0, index)).Width;

                    if (width > maxWidth)
                    {
                        maxIndex = index;
                    }
                    else if (width < maxWidth)
                    {
                        minIndex = index;
                    }
                    else
                    {
                        minIndex = index;
                        maxIndex = index;
                    }
                }

                Word newWord = new Word(word.PrecedingWhitespace, word.WhitespaceCount, word.Text.Substring(0, minIndex), font);
                yield return newWord;

                word = new Word('\0', 0, word.Text.Substring(minIndex), font);
            }

            yield return word;
        }

        public static IEnumerable<Word> GetWords(string text, Font font, double maxWidth)
        {
            if (!SpaceWidths.TryGetValue(font.FontFamily, out double spaceWidth))
            {
                spaceWidth = font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(' ') / 1000.0;
                SpaceWidths[font.FontFamily] = spaceWidth;
            }

            spaceWidth *= font.FontSize;

            foreach (Word w in GetWordsInternal(text, font))
            {
                if (w.Metrics.Width + w.Metrics.LeftSideBearing + w.Metrics.RightSideBearing + spaceWidth * w.WhitespaceCount * (w.PrecedingWhitespace == '\t' ? 4 : 1) > maxWidth)
                {
                    foreach (Word w2 in SplitWord(w, font, maxWidth))
                    {
                        yield return w2;
                    }
                }
                else
                {
```

```
                    yield return w;
                }
            }
        }

        private static IEnumerable<Word> GetWordsInternal(string text, Font font)
        {
            StringBuilder currWord = new StringBuilder();

            char currWhitespace = '\0';
            int whitespaceCount = 0;

            for (int i = 0; i < text.Length; i++)
            {
                if (char.IsWhiteSpace(text[i]))
                {
                    if (currWord.Length > 0)
                    {
                        yield return new Word(currWhitespace, whitespaceCount, currWord.ToString(), font);

                        currWord.Clear();
                        currWhitespace = text[i];
                        whitespaceCount = 1;
                    }
                    else if (currWhitespace == text[i])
                    {
                        whitespaceCount++;
                    }
                    else
                    {
                        yield return new Word(currWhitespace, whitespaceCount, null, font);

                        currWhitespace = text[i];
                        whitespaceCount = 1;
                    }
                }
                else
                {
                    currWord.Append(text[i]);
                }
            }

            if (currWord.Length > 0)
            {
                yield return new Word(currWhitespace, whitespaceCount, currWord.ToString(), font);

                currWord.Clear();
            }
            else if (whitespaceCount > 0)
            {
                yield return new Word(currWhitespace, whitespaceCount, null, font);
            }
        }
    }
}
```

8.13 ImageCache.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using Avalonia.Controls.ApplicationLifetimes;
using System;
using System.Collections.Generic;
using System.IO;

namespace VectSharp.MarkdownCanvas
{
```

```
    internal static class ImageCache
    {
        private const string imageCacheId = "bb5a724e-93f7-431d-87c3-53f31d8da16e";

        private static string imageCacheFolder;

        private static Dictionary<string, string> imageCache;

        static ImageCache()
        {
            imageCacheFolder = Path.Combine(Path.GetTempPath(), imageCacheId);
            Directory.CreateDirectory(imageCacheFolder);
            imageCache = new Dictionary<string, string>();
        }

        private static bool exitHandlerSet = false;
        public static void SetExitEventHandler()
        {
            if (!exitHandlerSet)
            {
                if (Avalonia.Application.Current.ApplicationLifetime is IControlledApplicationLifetime lifetime)
                {
                    lifetime.Exit += (s, e) =>
                    {
                        try
                        {
                            Directory.Delete(imageCacheFolder, true);
                        }
                        catch
                        {
                            try
                            {
                                foreach (string sr in Directory.GetFiles(imageCacheFolder, "*.*"))
                                {
                                    try
                                    {
                                        File.Delete(sr);
                                    }
                                    catch { }
                                }
                            }
                            catch { }
                        }
                    };
                }
                exitHandlerSet = true;
            }
        }

        public static (string, bool) ImageUriResolver(string imageUri, string baseUriString)
        {
            if (!imageCache.TryGetValue(baseUriString + "|||" + imageUri, out string cachedImage))
            {
                (string imagePath, bool wasDownloaded) = VectSharp.Markdown.HTTPUtils.ResolveImageURI(imageUri, baseUriString);

                if (!string.IsNullOrEmpty(imagePath) && File.Exists(imagePath))
                {
                    string id = Guid.NewGuid().ToString();

                    cachedImage = Path.Combine(imageCacheFolder, id + Path.GetExtension(imagePath));

                    if (wasDownloaded)
                    {
                        if (!Directory.Exists(imageCacheFolder))
                        {
                            Directory.CreateDirectory(imageCacheFolder);
                        }

                        File.Move(imagePath, cachedImage);
                        Directory.Delete(Path.GetDirectoryName(imagePath));
                    }
                    else
                    {
                        File.Copy(imagePath, cachedImage);
                    }

                    imageCache[baseUriString + "|||" + imageUri] = cachedImage;
                }
                else
                {
                    cachedImage = null;
                }
            }

            return (cachedImage, false);
```

```
        }
    }
}
```

8.14 MarkdownCanvas.axaml.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using Avalonia;
using Avalonia.Controls;
using Avalonia.Controls.ApplicationLifetimes;
using Avalonia.Markup.Xaml;
using Avalonia.Threading;
using Markdig;
using Markdig.Syntax;
using System;
using System.Collections.Generic;
using VectSharp.Canvas;
using VectSharp.Markdown;

namespace VectSharp.MarkdownCanvas
{
    /// <summary>
    /// A control to display a Markdown document in an Avalonia application.
    /// </summary>
    public class MarkdownCanvasControl : UserControl
    {
        /// <summary>
        /// Defines the <see cref="MaxRenderWidth"/> property.
        /// </summary>
        public static readonly StyledProperty<double> MaxRenderWidthProperty = AvaloniaProperty.Register<MarkdownCanvasControl, double>(nameof(MaxRenderWidth), double.PositiveInfinity);

        /// <summary>
        /// The maximum width for the rendered document. This will be used even if the control's client area is larger than this (the alignment of the document within the controll will depend on the control's <see cref="ContentControl.HorizontalContentAlignment"/>).
        /// </summary>
        public double MaxRenderWidth
        {
            get { return GetValue(MaxRenderWidthProperty); }
            set { SetValue(MaxRenderWidthProperty, value); }
        }

        /// <summary>
        /// Defines the <see cref="MinRenderWidth"/> property.
        /// </summary>
        public static readonly StyledProperty<double> MinRenderWidthProperty = AvaloniaProperty.Register<MarkdownCanvasControl, double>(nameof(MinRenderWidth), 200);

        /// <summary>
        /// The minimum width for the rendered document. If the control's client area is smaller than this, the horizontal scroll bar will be activated.
        /// </summary>
        public double MinRenderWidth
        {
            get { return GetValue(MinRenderWidthProperty); }
            set { SetValue(MinRenderWidthProperty, value); }
        }

        /// <summary>
        /// Defines the <see cref="MinVariation"/> property.
        /// </summary>
        public static readonly StyledProperty<double> MinVariationProperty = AvaloniaProperty.Register<MarkdownCanvasControl, double>(nameof(MinVariation), 10);

        /// <summary>
```

```
        /// The minimum width variation that triggers a document reflow. If the control is resized, but the width changes by less than this amount, the document is not re-drawn.
        /// </summary>
        public double MinVariation
        {
            get { return GetValue(MinVariationProperty); }
            set { SetValue(MinVariationProperty, value); }
        }

        /// <summary>
        /// Defines the <see cref="DocumentSource"/> property.
        /// </summary>
        public static readonly StyledProperty<string> DocumentSourceProperty = AvaloniaProperty.Register<MarkdownCanvasControl, string>(nameof(DocumentSource));

        /// <summary>
        /// Sets the currently displayed document from Markdown source.
        /// </summary>
        public string DocumentSource
        {
            set { SetValue(DocumentSourceProperty, value); }
        }

        /// <summary>
        /// Defines the <see cref="Document"/> property.
        /// </summary>
        public static readonly StyledProperty<MarkdownDocument> DocumentProperty = AvaloniaProperty.Register<MarkdownCanvasControl, MarkdownDocument>(nameof(Document));

        /// <summary>
        /// Gets or sets the currently displayed <see cref="MarkdownDocument"/>.
        /// </summary>
        public MarkdownDocument Document
        {
            get { return GetValue(DocumentProperty); }
            set { SetValue(DocumentProperty, value); }
        }

        /// <summary>
        /// Defines the <see cref="TextConversionOption"/> property.
        /// </summary>
        public static readonly StyledProperty<AvaloniaContextInterpreter.TextOptions> TextConversionOptionsProperty = AvaloniaProperty.Register<MarkdownCanvasControl, AvaloniaContextInterpreter.TextOptions>(nameof(TextConversionOption), AvaloniaContextInterpreter.TextOptions.ConvertIfNecessary);

        /// <summary>
        /// Gets or sets the value that determines whether text items should be converted into paths when drawing.
        /// Setting this to <see cref="AvaloniaContextInterpreter.TextOptions.NeverConvert"/> will improve performance if you are using custom fonts, but may cause unexpected results unless the font families being used are of type <see cref="ResourceFontFamily"/>.
        /// </summary>
        public AvaloniaContextInterpreter.TextOptions TextConversionOption
        {
            get { return GetValue(TextConversionOptionsProperty); }
            set { SetValue(TextConversionOptionsProperty, value); }
        }

        /// <summary>
        /// The <see cref="MarkdownRenderer"/> used to render the <see cref="Document"/>. You can use the properties of this object to customise the rendering. Note that setting the <see cref="Avalonia.Controls.Primitives.TemplatedControl.FontSize"/> of the <see cref="MarkdownCanvasControl"/> will propagate to the <see cref="Renderer"/>'s <see cref="MarkdownRenderer.BaseFontSize"/>.
        /// </summary>
        public MarkdownRenderer Renderer { get; }

        private double lastRenderedWidth = double.NaN;

        private bool initialized = false;

        /// <summary>
        /// Initialises a new <see cref="MarkdownCanvasControl"/>.
        /// </summary>
        public MarkdownCanvasControl()
        {
            InitializeComponent();
            this.Renderer = new MarkdownRenderer() { BaseFontSize = this.FontSize, Margins = new Margins(10, 10, 10, 10), ImageUriResolver = ImageCache.ImageUriResolver };
            this.initialized = true;
            ImageCache.SetExitEventHandler();
        }

        /// <inheritdoc/>
        protected override void OnPropertyChanged<T>(AvaloniaPropertyChangedEventArgs<T> change)
        {
```

```
            base.OnPropertyChanged(change);

            if (this.initialized)
            {
                if (change.Property == MarkdownCanvasControl.BoundsProperty || change.Property == MarkdownCanvasControl.MinRenderWidthProperty || change.Property == MarkdownCanvasControl.MaxRenderWidthProperty || change.Property == MarkdownCanvasControl.MinVariationProperty)
                {
                    Render();
                }
                else if (change.Property == MarkdownCanvasControl.DocumentProperty)
                {
                    forcedRerender = true;
                    Render();
                }
                else if (change.Property == MarkdownCanvasControl.FontSizeProperty)
                {
                    this.Renderer.BaseFontSize = this.FontSize;
                    forcedRerender = true;
                    Render();
                }
                else if (change.Property == MarkdownCanvasControl.DocumentSourceProperty)
                {
                    MarkdownDocument document = Markdig.Markdown.Parse(change.NewValue.GetValueOrDefault<string>(), new MarkdownPipelineBuilder().UseGridTables().UsePipeTables().UseEmphasisExtras().UseGenericAttributes().UseAutoIdentifiers
                    this.Document = document;
                }
                else if (change.Property == MarkdownCanvasControl.VerticalContentAlignmentProperty)
                {
                    this.FindControl<ScrollViewer>("ScrollViewer").VerticalContentAlignment = this.VerticalContentAlignment;
                }
                else if (change.Property == MarkdownCanvasControl.HorizontalContentAlignmentProperty)
                {
                    this.FindControl<ScrollViewer>("ScrollViewer").HorizontalContentAlignment = this.HorizontalContentAlignment;
                }
            }
        }

        private bool forcedRerender = false;

        private void Render()
        {
            if (Document != null)
            {
                double width = Math.Min(MaxRenderWidth, Math.Max(MinRenderWidth, this.Bounds.Width - MinVariation - 13));

                if (forcedRerender || double.IsNaN(lastRenderedWidth) || width != lastRenderedWidth && width < lastRenderedWidth - MinVariation || width > lastRenderedWidth + MinVariation)
                {
                    Page pag;
                    Dictionary<string, string> linkDestinations;

                    try
                    {
                        pag = Renderer.RenderSinglePage(this.Document, width, out linkDestinations);
                    }
                    catch
                    {
                        pag = new Page(width, 0);
                        linkDestinations = new Dictionary<string, string>();
                    }

                    Dictionary<string, Delegate> taggedActions = new Dictionary<string, Delegate>();
                    Dictionary<string, Avalonia.Point> linkDestinationPoints = new Dictionary<string, Avalonia.Point>();

                    foreach (KeyValuePair<string, string> linkDestination in linkDestinations)
                    {
                        string url = linkDestination.Value;

                        taggedActions.Add(linkDestination.Key, (Func<RenderAction, IEnumerable<RenderAction»)(act =>
                        {
                            act.PointerEnter += (s, e) =>
                            {
                                act.Parent.Cursor = new Avalonia.Input.Cursor(Avalonia.Input.StandardCursorType.Hand);
                            };

                            act.PointerLeave += (s, e) =>
                            {
                                act.Parent.Cursor = new
```

```
Avalonia.Input.Cursor(Avalonia.Input.StandardCursorType.Arrow);
                            };

                            act.PointerPressed += (s, e) =>
                            {
                                if (url.StartsWith("#"))
                                {
                                    if (linkDestinationPoints.TryGetValue(url.Substring(1), out Avalonia.Point target))
                                    {
                                        ScrollViewer scrollViewer = this.FindControl<ScrollViewer>("ScrollViewer");

                                        scrollViewer.Offset = new Vector(Math.Max(Math.Min(scrollViewer.Offset.X, target.X), target.X - scrollViewer.Viewport.Width), target.Y);
                                    }
                                }
                                else
                                {
                                    System.Diagnostics.Process.Start(new System.Diagnostics.ProcessStartInfo() { FileName = url, UseShellExecute = true });
                                }
                            };

                            if (act.ActionType == RenderAction.ActionTypes.Path)
                            {
                                linkDestinationPoints[linkDestination.Key] = act.Geometry.Bounds.TopLeft.Transform(act.Transform);
                            }
                            else if (act.ActionType == RenderAction.ActionTypes.Text)
                            {
                                linkDestinationPoints[linkDestination.Key] = act.Text.Bounds.TopLeft.Transform(act.Transform);
                            }
                            else if (act.ActionType == RenderAction.ActionTypes.RasterImage)
                            {
                                linkDestinationPoints[linkDestination.Key] = act.ImageDestination.Value.TopLeft.Transform(act.Transform);
                            }

                            return new RenderAction[] { act };
                        }));

                        if (url.StartsWith("#"))
                        {
                            if (!taggedActions.ContainsKey(url.Substring(1)))
                            {
                                taggedActions.Add(url.Substring(1), (Func<RenderAction, IEnumerable<RenderAction>>)(act =>
                                {
                                    if (act.ActionType == RenderAction.ActionTypes.Path)
                                    {
                                        linkDestinationPoints[url.Substring(1)] = act.Geometry.Bounds.TopLeft.Transform(act.Transform);
                                    }
                                    else if (act.ActionType == RenderAction.ActionTypes.Text)
                                    {
                                        linkDestinationPoints[url.Substring(1)] = act.Text.Bounds.TopLeft.Transform(act.Transform);
                                    }
                                    else if (act.ActionType == RenderAction.ActionTypes.RasterImage)
                                    {
                                        linkDestinationPoints[url.Substring(1)] = act.ImageDestination.Value.TopLeft.Transform(act.Transform);
                                    }

                                    return new RenderAction[] { act };
                                }));
                            }
                        }
                    }

                    Avalonia.Controls.Canvas can = pag.PaintToCanvas(false, taggedActions, false, this.TextConversionOption);

                    this.FindControl<ScrollViewer>("ScrollViewer").Content = can;
                    this.FindControl<ScrollViewer>("ScrollViewer").Padding = new Thickness(0, 0, 0, 0);
                    lastRenderedWidth = width;
                    forcedRerender = false;
                }
                else
                {
                    this.FindControl<ScrollViewer>("ScrollViewer").Padding = new Thickness(0, 0, width - lastRenderedWidth, 0);
                }
```

```
            }
        }

        private void InitializeComponent()
        {
            AvaloniaXamlLoader.Load(this);
        }
    }
}
```

8.15 ImageUriParser.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using SixLabors.ImageSharp;
using System;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices;

namespace VectSharp.ImageSharpUtils
{
    /// <summary>
    /// Provides a method to parse an image URI into a page.
    /// </summary>
    public static class ImageURIParser
    {
        /// <summary>
        /// Parses an image URI into a page. This is intended to replace the default image URI interpreter in <c>VectSharp.SVG.Parser.ParseImageURI</c>. To do this, use something like:
        /// <code>VectSharp.SVG.Parser.ParseImageURI = VectSharp.ImageSharpUtils.ImageURIParser.Parser(VectSharp.SVG.Parser.ParseSVGURI);</code>
        /// </summary>
        /// <param name="parseSVG">A function to parse an SVG image uri into a page. You should pass <c>VectSharp.SVG.Parser.ParseSVGURI</c> as this argument.</param>
        /// <returns>A function to parse an image URI into a page.</returns>
        public static Func<string, bool, Page> Parser(Func<string, bool, Page> parseSVG)
        {
            return (string uri, bool interpolate) =>
            {
                if (uri.StartsWith("data:"))
                {
                    string mimeType = uri.Substring(uri.IndexOf(":") + 1, uri.IndexOf(";") - uri.IndexOf(":") - 1);

                    string type = uri.Substring(uri.IndexOf(";") + 1, uri.IndexOf(",") - uri.IndexOf(";") - 1);

                    if (mimeType != "image/svg+xml")
                    {
                        int offset = uri.IndexOf(",") + 1;

                        byte[] parsed;

                        string substring = uri.Substring(offset);

                        switch (type)
                        {
                            case "base64":
                                parsed = Convert.FromBase64String(uri.Substring(offset));
                                break;
                            case "":
                                parsed = (from el in System.Web.HttpUtility.UrlDecode(uri.Substring(offset)) select (byte)el).ToArray();
                                break;
                            default:
                                throw new InvalidDataException("Unknown data stream type!");
                        }
```

```
                        GCHandle handle = GCHandle.Alloc(parsed, GCHandleType.Pinned);

                        RasterImageStream img = new RasterImageStream(handle.AddrOfPinnedObject(), parsed.Length, interpolate: interpolate);

                        handle.Free();

                        Page pag = new Page(img.Width, img.Height);

                        pag.Graphics.DrawRasterImage(0, 0, img);

                        return pag;
                    }
                    else
                    {
                        return parseSVG(uri, interpolate);
                    }
                }

                return null;
            };
        }
    }
}
```

8.16 ImageURIParser.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020  Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Affero General Public License as
    published by the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU Affero General Public License for more details.

    You should have received a copy of the GNU Affero General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>
*/

using MuPDFCore;
using System;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices;

namespace VectSharp.MuPDFUtils
{
    /// <summary>
    /// Provides a method to parse an image URI into a page.
    /// </summary>
    public static class ImageURIParser
    {
        /// <summary>
        /// Parses an image URI into a page. This is intended to replace the default image URI interpreter in <c>VectSharp.SVG.Parser.ParseImageURI</c>. To do this, use something like:
        /// <code>VectSharp.SVG.Parser.ParseImageURI = VectSharp.MuPDFUtils.ImageURIParser.Parser(VectSharp.SVG.Parser.ParseSVGURI);</code>
        /// </summary>
        /// <param name="parseSVG">A function to parse an SVG image uri into a page. You should pass <c>VectSharp.SVG.Parser.ParseSVGURI</c> as this argument.</param>
        /// <returns>A function to parse an image URI into a page.</returns>
        public static Func<string, bool, Page> Parser(Func<string, bool, Page> parseSVG)
        {
            return (string uri, bool interpolate) =>
            {
                if (uri.StartsWith("data:"))
                {
                    string mimeType = uri.Substring(uri.IndexOf(":") + 1, uri.IndexOf(";") - uri.IndexOf(":") - 1);

                    string type = uri.Substring(uri.IndexOf(";") + 1, uri.IndexOf(",") - uri.IndexOf(";") - 1);

                    if (mimeType != "image/svg+xml")
                    {
                        int offset = uri.IndexOf(",") + 1;

```

```
                        byte[] parsed;

                        bool isVector = false;

                        InputFileTypes fileType;

                        switch (mimeType)
                        {
                            case "image/png":
                                fileType = InputFileTypes.PNG;
                                break;
                            case "image/jpeg":
                            case "image/jpg":
                                fileType = InputFileTypes.JPEG;
                                break;
                            case "image/gif":
                                fileType = InputFileTypes.GIF;
                                break;
                            case "image/bmp":
                                fileType = InputFileTypes.BMP;
                                break;
                            case "image/tiff":
                            case "image/tif":
                                fileType = InputFileTypes.TIFF;
                                break;
                            case "application/oxps":
                            case "application/vnd.ms-xpsdocument":
                                fileType = InputFileTypes.XPS;
                                isVector = true;
                                break;
                            case "application/x-cbz":
                                fileType = InputFileTypes.CBZ;
                                break;
                            case "application/epub+zip":
                                fileType = InputFileTypes.EPUB;
                                isVector = true;
                                break;
                            case "text/fb2+xml":
                                fileType = InputFileTypes.FB2;
                                break;
                            case "image/x-portable-anymap":
                                fileType = InputFileTypes.PNM;
                                break;
                            case "image/x-portable-arbitrarymap":
                                fileType = InputFileTypes.PAM;
                                break;
                            case "application/pdf":
                                fileType = InputFileTypes.PDF;
                                isVector = true;
                                break;
                            default:
                                fileType = InputFileTypes.PDF;
                                break;
                        }

                        string substring = uri.Substring(offset);


                        switch (type)
                        {
                            case "base64":
                                parsed = Convert.FromBase64String(uri.Substring(offset));
                                break;
                            case "":
                                parsed = (from el in System.Web.HttpUtility.UrlDecode(uri.Substring(offset)) select (byte)el).ToArray();
                                break;
                            default:
                                throw new InvalidDataException("Unknown data stream type!");
                        }

                        if (!isVector)
                        {
                            GCHandle handle = GCHandle.Alloc(parsed, GCHandleType.Pinned);

                            RasterImageStream img = new RasterImageStream(handle.AddrOfPinnedObject(), parsed.Length, fileType, interpolate: interpolate);

                            handle.Free();

                            Page pag = new Page(img.Width, img.Height);

                            pag.Graphics.DrawRasterImage(0, 0, img);

                            return pag;
                        }
                        else
```

```
                        {
                            string tempFile = Path.GetTempFileName();

                            using (MuPDFContext context = new MuPDFContext())
                            {
                                using (MuPDFDocument document = new MuPDFDocument(context, parsed, fileType))
                                {
                                    MuPDFDocument.CreateDocument(context, tempFile, DocumentOutputFileTypes.SVG, true, document.Pages[0]);
                                }
                            }

                            string tbr = "data:image/svg+xml;," + System.Web.HttpUtility.UrlEncode(File.ReadAllText(tempFile));

                            File.Delete(tempFile);

                            return parseSVG(tbr, interpolate);
                        }
                    }
                    else
                    {
                        return parseSVG(uri, interpolate);
                    }
                }

                return null;
            };
        }
    }
}
```

8.17 RasterImages.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Advanced;
using SixLabors.ImageSharp.Processing;
using System;
using System.IO;
using System.Runtime.InteropServices;

namespace VectSharp.ImageSharpUtils
{
    /// <summary>
    /// A <see cref="RasterImage"/> created from a file.
    /// </summary>
    public class RasterImageFile : RasterImage
    {
        /// <summary>
        /// Creates a new <see cref="RasterImage"/> from the specified file.
        /// </summary>
        /// <param name="fileName">The path to the file containing the image.</param>
        /// <param name="alpha">A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.</param>
        /// <param name="interpolate">A boolean value indicating whether the image should be interpolated when it is resized or not.</param>
        public RasterImageFile(string fileName, bool alpha = true, bool interpolate = true)
        {
            Image image;

            if (alpha)
            {
                image = Image.Load<SixLabors.ImageSharp.PixelFormats.Rgba32>(fileName, out SixLabors.ImageSharp.Formats.IImageFormat format);
            }
```

```
            else
            {
                image = Image.Load<SixLabors.ImageSharp.PixelFormats.Rgb24>(fileName, out SixLabors.ImageSharp.Formats.IImageFormat format);
            }

            image.Mutate(x => x.AutoOrient());

            int stride = image.Width * (alpha ? 4 : 3);
            int size = stride * image.Height;

            IntPtr tbr = Marshal.AllocHGlobal(size);
            GC.AddMemoryPressure(size);

            IntPtr pointer = tbr;

            unsafe
            {
                if (alpha)
                {
                    Image<SixLabors.ImageSharp.PixelFormats.Rgba32> img = (Image<SixLabors.ImageSharp.PixelFormats.Rgba32>)image;

                    for (int y = 0; y < image.Height; y++)
                    {
                        Memory<SixLabors.ImageSharp.PixelFormats.Rgba32> row = img.DangerousGetPixelRowMemory(y);

                        Span<SixLabors.ImageSharp.PixelFormats.Rgba32> newRow = new Span<SixLabors.ImageSharp.PixelFormats.Rgba32>(pointer.ToPointer(), row.Length);
                        row.Span.CopyTo(newRow);

                        pointer = IntPtr.Add(pointer, stride);
                    }
                }
                else
                {
                    Image<SixLabors.ImageSharp.PixelFormats.Rgb24> img = (Image<SixLabors.ImageSharp.PixelFormats.Rgb24>)image;

                    for (int y = 0; y < image.Height; y++)
                    {
                        Memory<SixLabors.ImageSharp.PixelFormats.Rgb24> row = img.DangerousGetPixelRowMemory(y);

                        Span<SixLabors.ImageSharp.PixelFormats.Rgb24> newRow = new Span<SixLabors.ImageSharp.PixelFormats.Rgb24>(pointer.ToPointer(), row.Length);
                        row.Span.CopyTo(newRow);

                        pointer = IntPtr.Add(pointer, stride);
                    }
                }
            }

            this.Width = image.Width;
            this.Height = image.Height;
            this.HasAlpha = alpha;
            this.Interpolate = interpolate;
            this.Id = Guid.NewGuid().ToString();
            this.ImageDataAddress = tbr;
            this.DataHolder = new DisposableIntPtr(tbr);
        }
    }

    /// <summary>
    /// A <see cref="RasterImage"/> created from a stream.
    /// </summary>
    public class RasterImageStream : RasterImage
    {
        /// <summary>
        /// Creates a new <see cref="RasterImage"/> from the specified stream.
        /// </summary>
        /// <param name="imageStream">The stream containing the image data.</param>
        /// <param name="alpha">A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.</param>
        /// <param name="interpolate">A boolean value indicating whether the image should be interpolated when it is resized or not.</param>
        public RasterImageStream(Stream imageStream, bool alpha = true, bool interpolate = true)
        {
            Image image;

            if (alpha)
            {
                image = Image.Load<SixLabors.ImageSharp.PixelFormats.Rgba32>(imageStream, out SixLabors.ImageSharp.Formats.IImageFormat format);
            }
            else
```

```
            {
                image = Image.Load<SixLabors.ImageSharp.PixelFormats.Rgb24>(imageStream, out SixLabors.ImageSharp.Formats.IImageFormat format);
            }

            image.Mutate(x => x.AutoOrient());

            int stride = image.Width * (alpha ? 4 : 3);
            int size = stride * image.Height;

            IntPtr tbr = Marshal.AllocHGlobal(size);
            GC.AddMemoryPressure(size);

            IntPtr pointer = tbr;

            unsafe
            {
                if (alpha)
                {
                    Image<SixLabors.ImageSharp.PixelFormats.Rgba32> img = (Image<SixLabors.ImageSharp.PixelFormats.Rgba32>)image;

                    for (int y = 0; y < image.Height; y++)
                    {
                        Memory<SixLabors.ImageSharp.PixelFormats.Rgba32> row = img.DangerousGetPixelRowMemory(y);

                        Span<SixLabors.ImageSharp.PixelFormats.Rgba32> newRow = new Span<SixLabors.ImageSharp.PixelFormats.Rgba32>(pointer.ToPointer(), row.Length);
                        row.Span.CopyTo(newRow);

                        pointer = IntPtr.Add(pointer, stride);
                    }
                }
                else
                {
                    Image<SixLabors.ImageSharp.PixelFormats.Rgb24> img = (Image<SixLabors.ImageSharp.PixelFormats.Rgb24>)image;

                    for (int y = 0; y < image.Height; y++)
                    {
                        Memory<SixLabors.ImageSharp.PixelFormats.Rgb24> row = img.DangerousGetPixelRowMemory(y);

                        Span<SixLabors.ImageSharp.PixelFormats.Rgb24> newRow = new Span<SixLabors.ImageSharp.PixelFormats.Rgb24>(pointer.ToPointer(), row.Length);
                        row.Span.CopyTo(newRow);

                        pointer = IntPtr.Add(pointer, stride);
                    }
                }
            }

            this.Width = image.Width;
            this.Height = image.Height;
            this.HasAlpha = alpha;
            this.Interpolate = interpolate;
            this.Id = Guid.NewGuid().ToString();
            this.ImageDataAddress = tbr;
            this.DataHolder = new DisposableIntPtr(tbr);
        }

        /// <summary>
        /// Creates a new <see cref="RasterImage"/> from the specified stream.
        /// </summary>
        /// <param name="imageAddress">A pointer to the address where the image data is contained.</param>
        /// <param name="imageLength">The length in bytes of the image data.</param>
        /// <param name="alpha">A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.</param>
        /// <param name="interpolate">A boolean value indicating whether the image should be interpolated when it is resized or not.</param>
        public RasterImageStream(IntPtr imageAddress, int imageLength, bool alpha = true, bool interpolate = true)
        {
            unsafe
            {
                ReadOnlySpan<byte> imageSpan = new ReadOnlySpan<byte>((void*)imageAddress, imageLength);

                Image image;

                if (alpha)
                {
                    image = Image.Load<SixLabors.ImageSharp.PixelFormats.Rgba32>(imageSpan, out SixLabors.ImageSharp.Formats.IImageFormat format);
                }
```

```
                else
                {
                    image = Image.Load<SixLabors.ImageSharp.PixelFormats.Rgb24>(imageSpan, out SixLabors.ImageSharp.Formats.IImageFormat format);
                }

                image.Mutate(x => x.AutoOrient());

                int stride = image.Width * (alpha ? 4 : 3);
                int size = stride * image.Height;

                IntPtr tbr = Marshal.AllocHGlobal(size);
                GC.AddMemoryPressure(size);

                IntPtr pointer = tbr;

                if (alpha)
                {
                    Image<SixLabors.ImageSharp.PixelFormats.Rgba32> img = (Image<SixLabors.ImageSharp.PixelFormats.Rgba32>)image;

                    for (int y = 0; y < image.Height; y++)
                    {
                        Memory<SixLabors.ImageSharp.PixelFormats.Rgba32> row = img.DangerousGetPixelRowMemory(y);

                        Span<SixLabors.ImageSharp.PixelFormats.Rgba32> newRow = new Span<SixLabors.ImageSharp.PixelFormats.Rgba32>(pointer.ToPointer(), row.Length);
                        row.Span.CopyTo(newRow);

                        pointer = IntPtr.Add(pointer, stride);
                    }
                }
                else
                {
                    Image<SixLabors.ImageSharp.PixelFormats.Rgb24> img = (Image<SixLabors.ImageSharp.PixelFormats.Rgb24>)image;

                    for (int y = 0; y < image.Height; y++)
                    {
                        Memory<SixLabors.ImageSharp.PixelFormats.Rgb24> row = img.DangerousGetPixelRowMemory(y);

                        Span<SixLabors.ImageSharp.PixelFormats.Rgb24> newRow = new Span<SixLabors.ImageSharp.PixelFormats.Rgb24>(pointer.ToPointer(), row.Length);
                        row.Span.CopyTo(newRow);

                        pointer = IntPtr.Add(pointer, stride);
                    }
                }

                this.Width = image.Width;
                this.Height = image.Height;
                this.HasAlpha = alpha;
                this.Interpolate = interpolate;
                this.Id = Guid.NewGuid().ToString();
                this.ImageDataAddress = tbr;
                this.DataHolder = new DisposableIntPtr(tbr);
            }
        }
    }
}
```

8.18 RasterImages.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020  Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Affero General Public License as
    published by the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU Affero General Public License for more details.

    You should have received a copy of the GNU Affero General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>
*/

using MuPDFCore;
```

```
using System;
using System.IO;
using System.Runtime.InteropServices;

namespace VectSharp.MuPDFUtils
{
    /// <summary>
    /// A <see cref="RasterImage"/> created from a file.
    /// </summary>
    public class RasterImageFile : RasterImage
    {
        /// <summary>
        /// Creates a new <see cref="RasterImage"/> from the specified file.
        /// </summary>
        /// <param name="fileName">The path to the file containing the image.</param>
        /// <param name="pageNumber">The number of the page in the file from which the image should be created, starting at 0. Only useful for multi-page formats, such as PDF.</param>
        /// <param name="scale">The scale factor at which to render the image.</param>
        /// <param name="alpha">A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.</param>
        /// <param name="interpolate">A boolean value indicating whether the image should be interpolated when it is resized or not.</param>
        public RasterImageFile(string fileName, int pageNumber = 0, double scale = 1, bool alpha = true, bool interpolate = true)
        {
            using (MuPDFContext context = new MuPDFContext())
            {
                using (MuPDFDocument document = new MuPDFDocument(context, fileName))
                {
                    RoundedRectangle roundedBounds = document.Pages[pageNumber].Bounds.Round(scale);

                    this.Width = roundedBounds.Width;
                    this.Height = roundedBounds.Height;

                    this.HasAlpha = alpha;
                    this.Interpolate = interpolate;

                    this.Id = Guid.NewGuid().ToString();

                    int imageSize = document.GetRenderedSize(pageNumber, scale, alpha ? MuPDFCore.PixelFormats.RGBA : MuPDFCore.PixelFormats.RGB);

                    this.ImageDataAddress = Marshal.AllocHGlobal(imageSize);
                    this.DataHolder = new DisposableIntPtr(this.ImageDataAddress);
                    GC.AddMemoryPressure(imageSize);

                    document.Render(pageNumber, scale, alpha ? MuPDFCore.PixelFormats.RGBA : MuPDFCore.PixelFormats.RGB, this.ImageDataAddress);
                }
            }
        }
    }

    /// <summary>
    /// A <see cref="RasterImage"/> created from a stream.
    /// </summary>
    public class RasterImageStream : RasterImage
    {
        /// <summary>
        /// Creates a new <see cref="RasterImage"/> from the specified stream.
        /// </summary>
        /// <param name="imageStream">The stream containing the image data.</param>
        /// <param name="fileType">The type of the image contained in the stream.</param>
        /// <param name="pageNumber">The number of the page in the file from which the image should be created, starting at 0. Only useful for multi-page formats, such as PDF.</param>
        /// <param name="scale">The scale factor at which to render the image.</param>
        /// <param name="alpha">A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.</param>
        /// <param name="interpolate">A boolean value indicating whether the image should be interpolated when it is resized or not.</param>
        public RasterImageStream(Stream imageStream, InputFileTypes fileType, int pageNumber = 0, double scale = 1, bool alpha = true, bool interpolate = true)
        {
            using (MuPDFContext context = new MuPDFContext())
            {
                IntPtr originalImageAddress;
                long originalImageLength;

                IDisposable toBeDisposed = null;
                GCHandle handleToFree;

                if (imageStream is MemoryStream ms)
                {
                    int origin = (int)ms.Seek(0, SeekOrigin.Begin);
                    originalImageLength = ms.Length;

                    handleToFree = GCHandle.Alloc(ms.GetBuffer(), GCHandleType.Pinned);
```

```
                    originalImageAddress = handleToFree.AddrOfPinnedObject();
                }
                else
                {
                    MemoryStream mem = new MemoryStream((int)imageStream.Length);
                    imageStream.CopyTo(mem);

                    toBeDisposed = mem;

                    int origin = (int)mem.Seek(0, SeekOrigin.Begin);
                    originalImageLength = mem.Length;

                    handleToFree = GCHandle.Alloc(mem.GetBuffer(), GCHandleType.Pinned);
                    originalImageAddress = handleToFree.AddrOfPinnedObject();
                }

                using (MuPDFDocument document = new MuPDFDocument(context, originalImageAddress, originalImageLength, fileType))
                {
                    RoundedRectangle roundedBounds = document.Pages[pageNumber].Bounds.Round(scale);

                    this.Width = roundedBounds.Width;
                    this.Height = roundedBounds.Height;

                    this.HasAlpha = alpha;
                    this.Interpolate = interpolate;

                    this.Id = Guid.NewGuid().ToString();

                    int imageSize = document.GetRenderedSize(pageNumber, scale, alpha ? MuPDFCore.PixelFormats.RGBA : MuPDFCore.PixelFormats.RGB);

                    this.ImageDataAddress = Marshal.AllocHGlobal(imageSize);
                    this.DataHolder = new DisposableIntPtr(this.ImageDataAddress);
                    GC.AddMemoryPressure(imageSize);

                    document.Render(pageNumber, scale, alpha ? MuPDFCore.PixelFormats.RGBA : MuPDFCore.PixelFormats.RGB, this.ImageDataAddress);
                }

                handleToFree.Free();
                toBeDisposed?.Dispose();
            }
        }

        /// <summary>
        /// Creates a new <see cref="RasterImage"/> from the specified stream.
        /// </summary>
        /// <param name="imageAddress">A pointer to the address where the image data is contained.</param>
        /// <param name="imageLength">The length in bytes of the image data.</param>
        /// <param name="fileType">The type of the image contained in the stream.</param>
        /// <param name="pageNumber">The number of the page in the file from which the image should be created, starting at 0. Only useful for multi-page formats, such as PDF.</param>
        /// <param name="scale">The scale factor at which to render the image.</param>
        /// <param name="alpha">A boolean value indicating whether transparency (alpha) data from the image should be preserved or not.</param>
        /// <param name="interpolate">A boolean value indicating whether the image should be interpolated when it is resized or not.</param>
        public RasterImageStream(IntPtr imageAddress, long imageLength, InputFileTypes fileType, int pageNumber = 0, double scale = 1, bool alpha = true, bool interpolate = true)
        {
            using (MuPDFContext context = new MuPDFContext())
            {
                using (MuPDFDocument document = new MuPDFDocument(context, imageAddress, imageLength, fileType))
                {
                    RoundedRectangle roundedBounds = document.Pages[pageNumber].Bounds.Round(scale);

                    this.Width = roundedBounds.Width;
                    this.Height = roundedBounds.Height;

                    this.HasAlpha = alpha;
                    this.Interpolate = interpolate;

                    this.Id = Guid.NewGuid().ToString();

                    int imageSize = document.GetRenderedSize(pageNumber, scale, alpha ? MuPDFCore.PixelFormats.RGBA : MuPDFCore.PixelFormats.RGB);

                    this.ImageDataAddress = Marshal.AllocHGlobal(imageSize);
                    this.DataHolder = new DisposableIntPtr(this.ImageDataAddress);
                    GC.AddMemoryPressure(imageSize);

                    document.Render(pageNumber, scale, alpha ? MuPDFCore.PixelFormats.RGBA : MuPDFCore.PixelFormats.RGB, this.ImageDataAddress);
                }
```

```
            }
        }
    }
}
```

8.19 PDFContext.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel;
using System.IO;
using System.IO.Compression;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
using VectSharp.Filters;

namespace VectSharp.PDF
{
    internal enum SegmentType
    {
        Move, Line, CubicBezier, Close
    }

    internal abstract class Segment
    {
        public abstract SegmentType Type { get; }
        public Point[] Points { get; protected set; }

        public virtual Point Point
        {
            get
            {
                return Points[Points.Length - 1];
            }
        }

        public abstract Segment Clone();
    }

    internal class MoveSegment : Segment
    {
        public override SegmentType Type => SegmentType.Move;

        public MoveSegment(Point p)
        {
            this.Points = new Point[] { p };
        }

        public MoveSegment(double x, double y)
        {
            this.Points = new Point[] { new Point(x, y) };
        }

        public override Segment Clone()
        {
            return new MoveSegment(this.Point);
        }
    }

    internal class LineSegment : Segment
    {
        public override SegmentType Type => SegmentType.Line;

        public LineSegment(Point p)
```

```
        {
            this.Points = new Point[] { p };
        }

        public LineSegment(double x, double y)
        {
            this.Points = new Point[] { new Point(x, y) };
        }

        public override Segment Clone()
        {
            return new LineSegment(this.Point);
        }
    }

    internal class CloseSegment : Segment
    {
        public override SegmentType Type => SegmentType.Close;

        public CloseSegment() { }

        public override Segment Clone()
        {
            return new CloseSegment();
        }
    }

    internal class CubicBezierSegment : Segment
    {
        public override SegmentType Type => SegmentType.CubicBezier;
        public CubicBezierSegment(double x1, double y1, double x2, double y2, double x3, double y3)
        {
            Points = new Point[] { new Point(x1, y1), new Point(x2, y2), new Point(x3, y3) };
        }

        public CubicBezierSegment(Point p1, Point p2, Point p3)
        {
            Points = new Point[] { p1, p2, p3 };
        }

        public override Segment Clone()
        {
            return new CubicBezierSegment(Points[0], Points[1], Points[2]);
        }
    }

    internal interface IFigure
    {
        Brush Fill { get; }
        Brush Stroke { get; }
        double LineWidth { get; }
        LineCaps LineCap { get; }
        LineJoins LineJoin { get; }
        LineDash LineDash { get; }
        bool IsClipping { get; }
        string Tag { get; }
        Rectangle GetBounds();
    }

    internal class TransformFigure : IFigure
    {
        public enum TransformTypes
        {
            Transform, Save, Restore
        }

        public TransformTypes TransformType { get; }

        public Brush Fill { get; }
        public Brush Stroke { get; }

        public bool IsClipping { get; }
        public double LineWidth { get; }

        public double[,] TransformationMatrix { get; }

        public LineCaps LineCap { get; }

        public LineJoins LineJoin { get; }

        public LineDash LineDash { get; }
        public string Tag { get; }

        public TransformFigure(TransformTypes type, double[,] transformationMatrix, string tag)
        {
            this.TransformType = type;
            this.TransformationMatrix = transformationMatrix;
```

```
            this.Tag = tag;
        }

        public Rectangle GetBounds()
        {
            return Rectangle.NaN;
        }
    }

    internal class RasterImageFigure : IFigure
    {

        public Brush Fill { get; }
        public Brush Stroke { get; }

        public bool IsClipping { get; }
        public double LineWidth { get; }

        public double[,] TransformationMatrix { get; }

        public LineCaps LineCap { get; }

        public LineJoins LineJoin { get; }

        public LineDash LineDash { get; }

        public RasterImage Image { get; }

        public string Tag { get; }

        public RasterImageFigure(RasterImage image, string tag)
        {
            this.Image = image;
            this.Tag = tag;
        }

        public Rectangle GetBounds()
        {
            return new Rectangle(0, 0, 1, 1);
        }
    }

    internal class PathFigure : IFigure
    {
        public Brush Fill { get; }
        public Brush Stroke { get; }
        public bool IsClipping { get; }
        public double LineWidth { get; }

        public LineCaps LineCap { get; }

        public LineJoins LineJoin { get; }

        public LineDash LineDash { get; }
        public Segment[] Segments { get; }

        public string Tag { get; }

        private Rectangle Bounds { get; }

        public PathFigure(IEnumerable<Segment> segments, Rectangle bounds, Brush fill, Brush stroke,
double lineWidth, LineCaps lineCap, LineJoins lineJoin, LineDash lineDash, bool isClipping, string
tag)
        {
            List<Segment> segs = new List<Segment>();

            foreach (Segment s in segments)
            {
                segs.Add(s.Clone());
            }

            this.Segments = segs.ToArray();

            Fill = fill;
            Stroke = stroke;
            LineWidth = lineWidth;
            LineCap = lineCap;
            LineJoin = lineJoin;
            LineDash = lineDash;
            IsClipping = isClipping;
            this.Tag = tag;

            if (stroke == null)
            {
                this.Bounds = bounds;
            }
            else
```

```
            {
                this.Bounds = new Rectangle(bounds.Location.X - lineWidth * 0.5, bounds.Location.Y - lineWidth * 0.5, bounds.Size.Width + lineWidth, bounds.Size.Height + lineWidth);
            }
        }

        public Rectangle GetBounds()
        {
            return Bounds;
        }
    }

    internal class TextFigure : IFigure
    {
        public Brush Fill { get; }
        public Brush Stroke { get; }
        public double LineWidth { get; }
        public bool IsClipping { get; }
        public LineCaps LineCap { get; }

        public LineJoins LineJoin { get; }

        public LineDash LineDash { get; }

        public string Text { get; }

        public Font Font { get; }

        public Point Position { get; }

        public TextBaselines TextBaseline { get; }

        public string Tag { get; }

        public TextFigure(string text, Font font, Point position, TextBaselines textBaseline, Brush fill, Brush stroke, double lineWidth, LineCaps lineCap, LineJoins lineJoin, LineDash lineDash, string tag)
        {
            Text = text;
            Font = font;
            Position = position;
            TextBaseline = textBaseline;

            Fill = fill;
            Stroke = stroke;
            LineWidth = lineWidth;
            LineCap = lineCap;
            LineJoin = lineJoin;
            LineDash = lineDash;
            this.Tag = tag;
        }

        public Rectangle GetBounds()
        {
            Font.DetailedFontMetrics metrics = this.Font.MeasureTextAdvanced(this.Text);

            switch (this.TextBaseline)
            {
                case TextBaselines.Top:
                    return new Rectangle(this.Position, new Size(metrics.Width, metrics.Height));
                case TextBaselines.Bottom:
                    return new Rectangle(this.Position.X, this.Position.Y - metrics.Height, metrics.Width, metrics.Height);
                case TextBaselines.Middle:
                    return new Rectangle(this.Position.X, this.Position.Y - metrics.Height * 0.5, metrics.Width, metrics.Height);
                case TextBaselines.Baseline:
                    return new Rectangle(this.Position.X, this.Position.Y - metrics.Top, metrics.Width, metrics.Height);
                default:
                    throw new System.ArgumentOutOfRangeException(nameof(TextBaseline), this.TextBaseline, "Invalid text baseline!");
            }
        }
    }

    internal class PDFFontDescriptor
    {
        private static readonly Random TagGenerator = new Random();
        private static readonly List<string> TagCache = new List<string>();

        public string FontName { get; }
        public string FontFamily { get; }
        public uint Flags { get; }
        public double[] FontBBox { get; }
        public int ItalicAngle => 0;
        public double Ascent { get; }
```

```
        public double Descent { get; }
        public double CapHeight { get { return Ascent; } }
        public int StemV => 80;
        public int StemH => 80;

        public PDFFontDescriptor(TrueTypeFile ttf, bool isSubset, bool isSymbolic)
        {
            this.Ascent = ttf.Get1000EmAscent();
            this.Descent = ttf.Get1000EmDescent();

            this.FontBBox = new double[] { ttf.Get1000EmXMin(), ttf.Get1000EmYMin(), ttf.Get1000EmXMax(), ttf.Get1000EmYMax() };

            bool fixedPitch = ttf.IsFixedPitch();

            bool serif = ttf.IsSerif();

            bool script = ttf.IsScript();

            bool italic = ttf.IsBold();

            bool allCap = false;

            bool smallCap = false;

            bool forceBold = false;

            this.Flags = (fixedPitch ? 1U : 0) | (serif ? 1U « 1 : 0) | (isSymbolic ? 1U « 2 : 0) | (script ? 1U « 3 : 0) | (!isSymbolic ? 1U « 5 : 0) | (italic ? 1U « 6 : 0) | (allCap ? 1U « 16 : 0) | (smallCap ? 1U « 17 : 0) | (forceBold ? 1U « 18 : 0);

            this.FontName = ttf.GetFontName();

            this.FontFamily = ttf.GetFontFamilyName();

            if (string.IsNullOrEmpty(this.FontFamily))
            {
                this.FontFamily = FontName;
            }

            if (isSubset)
            {
                string randString = "";

                while (randString.Length == 0 || TagCache.Contains(randString))
                {
                    randString = "";
                    for (int i = 0; i < 6; i++)
                    {
                        randString += (char)TagGenerator.Next(65, 91);
                    }
                }

                this.FontName = randString + "+" + this.FontName;
            }
        }
    }

    internal class PDFContext : IGraphicsContext
    {
        public string Tag { get; set; }
        public double Width { get; }
        public double Height { get; }


        private List<Segment> _currentFigure;

        internal List<IFigure> _figures;

        private Brush _strokeStyle;
        private Brush _fillStyle;
        private LineDash _lineDash;

        private readonly bool _textToPaths;

        private PDFContextInterpreter.FilterOption _filterOption;

        public PDFContext(double width, double height, Colour background, bool textToPaths, PDFContextInterpreter.FilterOption filterOption)
        {
            this.Width = width;
            this.Height = height;

            _currentFigure = new List<Segment>();
            _figures = new List<IFigure>();

            _strokeStyle = Colour.FromRgb(0, 0, 0);
```

```
            _fillStyle = Colour.FromRgb(0, 0, 0);
            LineWidth = 1;

            LineCap = LineCaps.Butt;
            LineJoin = LineJoins.Miter;
            _lineDash = new LineDash(0, 0, 0);

            _textToPaths = textToPaths;

            Font = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica), 12);

            TextBaseline = TextBaselines.Top;

            this.Translate(0, height);
            this.Scale(1, -1);

            this.Rectangle(0, 0, width, height);
            this.SetFillStyle(background);
            this.Fill();

            this.SetFillStyle(Colour.FromRgb(0, 0, 0));

            this._filterOption = filterOption;
        }


        public void MoveTo(double x, double y)
        {
            _currentFigure.Add(new MoveSegment(x, y));
        }

        public void LineTo(double x, double y)
        {
            _currentFigure.Add(new LineSegment(x, y));
        }

        public void Close()
        {
            _currentFigure.Add(new CloseSegment());
        }

        public void Rectangle(double x0, double y0, double width, double height)
        {
            MoveTo(x0, y0);
            LineTo(x0 + width, y0);
            LineTo(x0 + width, y0 + height);
            LineTo(x0, y0 + height);
            Close();
        }
        public void SetStrokeStyle((int r, int g, int b, double a) style)
        {
            _strokeStyle = Colour.FromRgba(style.r, style.g, style.b, style.a);
        }

        public void SetStrokeStyle(Brush style)
        {
            _strokeStyle = style;
        }

        public void SetFillStyle((int r, int g, int b, double a) style)
        {
            _fillStyle = Colour.FromRgba(style.r, style.g, style.b, style.a);
        }

        public void SetFillStyle(Brush style)
        {
            _fillStyle = style;
        }


        public Brush FillStyle { get { return _fillStyle; } }
        public Brush StrokeStyle { get { return _strokeStyle; } }

        public double LineWidth { get; set; }

        public LineCaps LineCap { get; set; }
        public LineJoins LineJoin { get; set; }

        internal static bool IsCompatible(Brush brush)
        {
            if (brush is SolidColourBrush)
            {
                return true;
            }
            else if (brush is GradientBrush gradient)
            {
```

```
                foreach (GradientStop stop in gradient.GradientStops)
                {
                    if (stop.Colour.A != 1)
                    {
                        return false;
                    }
                }
                return true;
            }
            else
            {
                return false;
            }
        }

        private static Brush RemoveAlpha(Brush brush)
        {
            if (brush is SolidColourBrush)
            {
                return brush;
            }
            else if (brush is LinearGradientBrush linear)
            {
                return new LinearGradientBrush(linear.StartPoint, linear.EndPoint, from el in linear.GradientStops select new GradientStop(el.Colour.WithAlpha(1.0), el.Offset));
            }
            else if (brush is RadialGradientBrush radial)
            {
                return new RadialGradientBrush(radial.FocalPoint, radial.Centre, radial.Radius, from el in radial.GradientStops select new GradientStop(el.Colour.WithAlpha(1.0), el.Offset));
            }
            else
            {
                return null;
            }
        }

        internal static Brush GetAlphaBrush(Brush brush)
        {
            if (brush is SolidColourBrush)
            {
                return brush;
            }
            else if (brush is LinearGradientBrush linear)
            {
                return new LinearGradientBrush(linear.StartPoint, linear.EndPoint, from el in linear.GradientStops select new GradientStop(Colour.FromRgb(el.Colour.A, el.Colour.A, el.Colour.A), el.Offset));
            }
            else if (brush is RadialGradientBrush radial)
            {
                return new RadialGradientBrush(radial.FocalPoint, radial.Centre, radial.Radius, from el in radial.GradientStops select new GradientStop(Colour.FromRgb(el.Colour.A, el.Colour.A, el.Colour.A), el.Offset));
            }
            else
            {
                return null;
            }
        }

        private static GraphicsPath GetGraphicsPath(IEnumerable<Segment> segments)
        {
            GraphicsPath tbr = new GraphicsPath();

            foreach (Segment seg in segments)
            {
                switch (seg.Type)
                {
                    case SegmentType.Close:
                        tbr.Close();
                        break;
                    case SegmentType.Move:
                        tbr.MoveTo(seg.Point);
                        break;
                    case SegmentType.Line:
                        tbr.LineTo(seg.Point);
                        break;
                    case SegmentType.CubicBezier:
                        tbr.CubicBezierTo(seg.Points[0], seg.Points[1], seg.Points[2]);
                        break;
                }
            }

            return tbr;
        }

```

```
        private Graphics GetCurrentFigureMask(Brush brush, bool stroke, bool blackBackground = false)
        {
            Graphics gpr = new Graphics();

            GraphicsPath path = GetGraphicsPath(_currentFigure);

            if (!stroke)
            {
                gpr.FillPath(path, brush);
            }
            else
            {
                gpr.StrokePath(path, brush, LineWidth, LineCap, LineJoin, _lineDash);
            }

            if (blackBackground)
            {
                Rectangle bounds = gpr.GetBounds();

                Graphics gpr2 = new Graphics();

                gpr2.FillRectangle(bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, Colours.White);
                gpr2.DrawGraphics(0, 0, gpr);

                gpr = gpr2;
            }

            return gpr;
        }

        public void Fill()
        {
            if (IsCompatible(_fillStyle))
            {
                _figures.Add(new PathFigure(_currentFigure, VectSharp.Rectangle.NaN, _fillStyle, null, 0, LineCaps.Butt, LineJoins.Bevel, new LineDash(0, 0, 0), false, this.Tag));
            }
            else
            {
                _figures.Add(new PathFigure(_currentFigure, GetGraphicsPath(_currentFigure).GetBounds(), _fillStyle, null, 0, LineCaps.Butt, LineJoins.Bevel, new LineDash(0, 0, 0), false, this.Tag));
            }

            _currentFigure = new List<Segment>();
        }

        public void Stroke()
        {
            if (IsCompatible(_strokeStyle))
            {
                _figures.Add(new PathFigure(_currentFigure, VectSharp.Rectangle.NaN, null, _strokeStyle, LineWidth, LineCap, LineJoin, _lineDash, false, this.Tag));
            }
            else
            {
                _figures.Add(new PathFigure(_currentFigure, GetGraphicsPath(_currentFigure).GetBounds(), null, _strokeStyle, LineWidth, LineCap, LineJoin, _lineDash, false, this.Tag));
            }

            _currentFigure = new List<Segment>();
        }

        public void SetClippingPath()
        {
            _figures.Add(new PathFigure(_currentFigure, VectSharp.Rectangle.NaN, null, null, 0, LineCaps.Butt, LineJoins.Bevel, new LineDash(0, 0, 0), true, this.Tag));
            _currentFigure = new List<Segment>();
        }

        public void CubicBezierTo(double x1, double y1, double x2, double y2, double x3, double y3)
        {
            _currentFigure.Add(new CubicBezierSegment(x1, y1, x2, y2, x3, y3));
        }

        public void SetLineDash(LineDash dash)
        {
            _lineDash = dash;
        }

        public Font Font { get; set; }

        private void PathText(string text, double x, double y)
        {
            GraphicsPath textPath = new GraphicsPath().AddText(x, y, text, Font, TextBaseline);
```

```

            for (int j = 0; j < textPath.Segments.Count; j++)
            {
                switch (textPath.Segments[j].Type)
                {
                    case VectSharp.SegmentType.Move:
                        this.MoveTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.Line:
                        this.LineTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.CubicBezier:
                        this.CubicBezierTo(textPath.Segments[j].Points[0].X, textPath.Segments[j].Points[0].Y, textPath.Segments[j].Points[1].X, textPath.Segments[j].Points[1].Y, textPath.Segments[j].Points[2].X, textPath.Segments[j].Points[2].Y);
                        break;
                    case VectSharp.SegmentType.Close:
                        this.Close();
                        break;
                }
            }
        }

        public void FillText(string text, double x, double y)
        {
            if (!_textToPaths)
            {
                _figures.Add(new TextFigure(text, Font, new Point(x, y), TextBaseline, _fillStyle, null, 0, LineCaps.Butt, LineJoins.Miter, new LineDash(0, 0, 0), this.Tag));
            }
            else
            {
                PathText(text, x, y);
                Fill();
            }
        }

        public TextBaselines TextBaseline { get; set; }

        public void Restore()
        {
            _figures.Add(new TransformFigure(TransformFigure.TransformTypes.Restore, null, this.Tag));
        }

        public void Rotate(double angle)
        {
            _figures.Add(new TransformFigure(TransformFigure.TransformTypes.Transform, new double[,] { { Math.Cos(angle), Math.Sin(angle), 0 }, { -Math.Sin(angle), Math.Cos(angle), 0 }, { 0, 0, 1 } }, this.Tag));
        }

        public void Save()
        {
            _figures.Add(new TransformFigure(TransformFigure.TransformTypes.Save, null, this.Tag));
        }


        public void StrokeText(string text, double x, double y)
        {
            if (!_textToPaths)
            {
                _figures.Add(new TextFigure(text, Font, new Point(x, y), TextBaseline, null, _strokeStyle, LineWidth, LineCap, LineJoin, _lineDash, this.Tag));
            }
            else
            {
                PathText(text, x, y);
                Stroke();
            }
        }

        public void Translate(double x, double y)
        {
            _figures.Add(new TransformFigure(TransformFigure.TransformTypes.Transform, new double[,] { { 1, 0, x }, { 0, 1, y }, { 0, 0, 1 } }, this.Tag));
        }

        public void Scale(double scaleX, double scaleY)
        {
            _figures.Add(new TransformFigure(TransformFigure.TransformTypes.Transform, new double[,] { { scaleX, 0, 0 }, { 0, scaleY, 0 }, { 0, 0, 1 } }, this.Tag));
        }

        public void Transform(double a, double b, double c, double d, double e, double f)
        {
            _figures.Add(new TransformFigure(TransformFigure.TransformTypes.Transform, new double[,] { { a, b, e }, { c, d, f }, { 0, 0, 1 } }, this.Tag));
```

```
        }

        public void DrawRasterImage(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image)
        {
            Save();

            MoveTo(destinationX, destinationY);
            LineTo(destinationX + destinationWidth, destinationY);
            LineTo(destinationX + destinationWidth, destinationY + destinationHeight);
            LineTo(destinationX, destinationY + destinationHeight);
            Close();
            SetClippingPath();

            double sourceRectX = (double)sourceX / image.Width;
            double sourceRectY = 1 - (double)sourceY / image.Height;
            double sourceRectWidth = (double)sourceWidth / image.Width;
            double sourceRectHeight = -(double)sourceHeight / image.Height;

            double scaleX = destinationWidth / sourceRectWidth;
            double scaleY = destinationHeight / sourceRectHeight;

            double translationX = destinationX / scaleX - sourceRectX;
            double translationY = destinationY / scaleY - sourceRectY;


            Scale(scaleX, scaleY);
            Translate(translationX, translationY);

            _figures.Add(new RasterImageFigure(image, this.Tag));

            Restore();
        }

        public void DrawFilteredGraphics(Graphics graphics, IFilter filter)
        {
            if (this._filterOption.Operation == PDFContextInterpreter.FilterOption.FilterOperations.RasteriseAll)
            {
                double scale = this._filterOption.RasterisationResolution;

                Rectangle bounds = graphics.GetBounds();

                bounds = new Rectangle(bounds.Location.X - filter.TopLeftMargin.X, bounds.Location.Y - filter.TopLeftMargin.Y, bounds.Size.Width + filter.TopLeftMargin.X + filter.BottomRightMargin.X, bounds.Size.Height + filter.TopLeftMargin.Y + filter.BottomRightMargin.Y);

                if (bounds.Size.Width > 0 && bounds.Size.Height > 0)
                {
                    if (!this._filterOption.RasterisationResolutionRelative)
                    {
                        scale = scale / Math.Min(bounds.Size.Width, bounds.Size.Height);
                    }

                    if (graphics.TryRasterise(bounds, scale, true, out RasterImage rasterised))
                    {
                        RasterImage filtered = null;

                        if (filter is ILocationInvariantFilter locInvFilter)
                        {
                            filtered = locInvFilter.Filter(rasterised, scale);
                        }
                        else if (filter is IFilterWithLocation filterWithLoc)
                        {
                            filtered = filterWithLoc.Filter(rasterised, bounds, scale);
                        }

                        if (filtered != null)
                        {
                            rasterised.Dispose();

                            DrawRasterImage(0, 0, filtered.Width, filtered.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, filtered);
                        }
                    }
                    else
                    {
                        throw new NotImplementedException(@"The filter could not be rasterised! You can avoid this error by doing one of the following:
• Add a reference to VectSharp.Raster or VectSharp.Raster.ImageSharp (you may also need to add a using directive somewhere to force the assembly to be loaded).
• Provide your own implementation of Graphics.RasterisationMethod.
• Set the FilterOption.Operation to ""IgnoreAll"" or ""SkipAll"".");
                    }
                }
            }
```

```
            else if (this._filterOption.Operation == PDFContextInterpreter.FilterOption.FilterOperations.IgnoreAll)
            {
                graphics.CopyToIGraphicsContext(this);
            }
            else
            {

            }
        }
    }

    /// <summary>
    /// Contains methods to render a <see cref="Document"/> as a PDF document.
    /// </summary>
    public static class PDFContextInterpreter
    {
        private static string GetKernedString(string text, Font font)
        {
            List<(string, Point)> tSpans = new List<(string, Point)>();

            StringBuilder currentRun = new StringBuilder();
            Point currentKerning = new Point();

            Point currentGlyphPlacementDelta = new Point();
            Point currentGlyphAdvanceDelta = new Point();
            Point nextGlyphPlacementDelta = new Point();
            Point nextGlyphAdvanceDelta = new Point();

            for (int i = 0; i < text.Length; i++)
            {
                if (i < text.Length - 1)
                {
                    currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                    currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                    nextGlyphAdvanceDelta = new Point();
                    nextGlyphPlacementDelta = new Point();

                    TrueTypeFile.PairKerning kerning = font.FontFamily.TrueTypeFile.Get1000EmKerning(text[i], text[i + 1]);

                    if (kerning != null)
                    {
                        currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                        currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                        nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                        nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                    }
                }

                if (currentGlyphPlacementDelta.X != 0 || currentGlyphPlacementDelta.Y != 0 || currentGlyphAdvanceDelta.X != 0 || currentGlyphAdvanceDelta.Y != 0)
                {
                    if (currentRun.Length > 0)
                    {
                        tSpans.Add((currentRun.ToString(), currentKerning));

                        tSpans.Add((text[i].ToString(), new Point(currentGlyphPlacementDelta.X, currentGlyphPlacementDelta.Y)));

                        currentRun.Clear();
                        currentKerning = new Point(currentGlyphAdvanceDelta.X - currentGlyphPlacementDelta.X, currentGlyphAdvanceDelta.Y - currentGlyphPlacementDelta.Y);
                    }
                    else
                    {
                        tSpans.Add((text[i].ToString(), new Point(currentGlyphPlacementDelta.X + currentKerning.X, currentGlyphPlacementDelta.Y + currentKerning.Y)));

                        currentRun.Clear();
                        currentKerning = new Point(currentGlyphAdvanceDelta.X - currentGlyphPlacementDelta.X, currentGlyphAdvanceDelta.Y - currentGlyphPlacementDelta.Y);
                    }
                }
                else
                {
                    currentRun.Append(text[i]);
                }
            }

            if (currentRun.Length > 0)
            {
```

```
                tSpans.Add((currentRun.ToString(), currentKerning));
            }

            StringBuilder sb = new StringBuilder();
            sb.Append("[");

            for (int i = 0; i < tSpans.Count; i++)
            {
                if (tSpans[i].Item2.X != 0)
                {
                    sb.Append((-tSpans[i].Item2.X).ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture));
                }

                sb.Append("(");
                sb.Append(EscapeStringForPDF(tSpans[i].Item1));
                sb.Append(")");
            }

            sb.Append("]");

            return sb.ToString();
        }


        private static string EscapeStringForPDF(string str)
        {
            StringBuilder sb = new StringBuilder();

            for (int i = 0; i < str.Length; i++)
            {
                char ch = str[i];

                if (CP1252Chars.Contains(ch))
                {
                    if ((int)ch < 128)
                    {
                        if (!"\n\r\t\b\f()\\".Contains(ch))
                        {
                            sb.Append(ch);
                        }
                        else
                        {
                            switch (ch)
                            {
                                case '\n':
                                    sb.Append("\\n");
                                    break;
                                case '\r':
                                    sb.Append("\\r");
                                    break;
                                case '\t':
                                    sb.Append("\\t");
                                    break;
                                case '\b':
                                    sb.Append("\\b");
                                    break;
                                case '\f':
                                    sb.Append("\\f");
                                    break;
                                case '\\':
                                    sb.Append("\\\\");
                                    break;
                                case '(':
                                    sb.Append("\\(");
                                    break;
                                case ')':
                                    sb.Append("\\)");
                                    break;
                            }
                        }
                    }
                    else
                    {
                        string octal = Convert.ToString((int)ch, 8);
                        while (octal.Length < 3)
                        {
                            octal = "0" + octal;
                        }
                        sb.Append("\\" + octal);
                    }
                }
                else
                {
                    sb.Append('?');
                }
            }
```

```
            return sb.ToString();
        }


        private static string EscapeSymbolStringForPDF(string str, Dictionary<char, int> glyphIndices)
        {
            StringBuilder sb = new StringBuilder();

            for (int i = 0; i < str.Length; i++)
            {
                sb.Append((glyphIndices[str[i]]).ToString("X4"));
            }
            return sb.ToString();
        }

        private static Dictionary<string, FontFamily> GetFontFamilies(PDFContext[] pdfContexts)
        {
            Dictionary<string, FontFamily> tbr = new Dictionary<string, FontFamily>();

            foreach (PDFContext ctx in pdfContexts)
            {
                foreach (IFigure act in ctx._figures)
                {
                    if (act is TextFigure figure &&
!tbr.ContainsKey(figure.Font.FontFamily.FamilyName))
                    {
                        tbr.Add(figure.Font.FontFamily.FamilyName,
FontFamily.ResolveFontFamily(figure.Font.FontFamily.FamilyName));
                    }
                }
            }

            return tbr;
        }

        private static Dictionary<string, HashSet<char» GetUsedChars(PDFContext[] pdfContexts)
        {
            Dictionary<string, HashSet<char» tbr = new Dictionary<string, HashSet<char»();

            foreach (PDFContext ctx in pdfContexts)
            {
                foreach (IFigure act in ctx._figures)
                {
                    if (act is TextFigure figure &&
!tbr.ContainsKey(figure.Font.FontFamily.FamilyName))
                    {
                        tbr.Add(figure.Font.FontFamily.FamilyName, new HashSet<char>(figure.Text));
                    }
                    else if (act is TextFigure figure1)
                    {
                        string txt = figure1.Text;
                        for (int i = 0; i < txt.Length; i++)
                        {
                            tbr[figure1.Font.FontFamily.FamilyName].Add(txt[i]);
                        }
                    }
                }
            }

            return tbr;
        }


        private static double[] GetAlphas(PDFContext[] pdfContexts)
        {
            HashSet<double> tbr = new HashSet<double>();

            tbr.Add(1);

            foreach (PDFContext ctx in pdfContexts)
            {
                foreach (IFigure act in ctx._figures)
                {
                    if (act.Stroke != null)
                    {
                        if (act.Stroke is SolidColourBrush solid)
                        {
                            tbr.Add(solid.A);
                        }
                        else if (act.Stroke is GradientBrush gradient)
                        {
                            foreach (GradientStop stop in gradient.GradientStops)
                            {
                                tbr.Add(stop.Colour.A);
                            }
                        }

```

```
                    }

                    if (act.Fill != null)
                    {
                        if (act.Fill is SolidColourBrush solid)
                        {
                            tbr.Add(solid.A);
                        }
                        else if (act.Fill is GradientBrush gradient)
                        {
                            foreach (GradientStop stop in gradient.GradientStops)
                            {
                                tbr.Add(stop.Colour.A);
                            }
                        }
                    }
                }
            }

            return tbr.ToArray();
        }

        private static Dictionary<string, RasterImage> GetAllImages(PDFContext[] pdfContexts)
        {
            Dictionary<string, RasterImage> tbr = new Dictionary<string, RasterImage>();

            foreach (PDFContext ctx in pdfContexts)
            {
                foreach (IFigure act in ctx._figures)
                {
                    if (act is RasterImageFigure figure && !tbr.ContainsKey(figure.Image.Id))
                    {
                        tbr.Add(figure.Image.Id, figure.Image);
                    }
                }
            }

            return tbr;
        }


        private static readonly char[] CP1252Chars = new char[] { '\u0000', '\u0001', '\u0002',
'\u0003', '\u0004', '\u0005', '\u0006', '\u0007', '\u0008', '\u0009', '\u000A', '\u000B', '\u000C',
'\u000D', '\u000E', '\u000F', '\u0010', '\u0011', '\u0012', '\u0013', '\u0014', '\u0015', '\u0016',
'\u0017', '\u0018', '\u0019', '\u001A', '\u001B', '\u001C', '\u001D', '\u001E', '\u001F', '\u0020',
'\u0021', '\u0022', '\u0023', '\u0024', '\u0025', '\u0026', '\u0027', '\u0028', '\u0029', '\u002A',
'\u002B', '\u002C', '\u002D', '\u002E', '\u002F', '\u0030', '\u0031', '\u0032', '\u0033', '\u0034',
'\u0035', '\u0036', '\u0037', '\u0038', '\u0039', '\u003A', '\u003B', '\u003C', '\u003D', '\u003E',
'\u003F', '\u0040', '\u0041', '\u0042', '\u0043', '\u0044', '\u0045', '\u0046', '\u0047', '\u0048',
'\u0049', '\u004A', '\u004B', '\u004C', '\u004D', '\u004E', '\u004F', '\u0050', '\u0051', '\u0052',
'\u0053', '\u0054', '\u0055', '\u0056', '\u0057', '\u0058', '\u0059', '\u005A', '\u005B', '\u005C',
'\u005D', '\u005E', '\u005F', '\u0060', '\u0061', '\u0062', '\u0063', '\u0064', '\u0065', '\u0066',
'\u0067', '\u0068', '\u0069', '\u006A', '\u006B', '\u006C', '\u006D', '\u006E', '\u006F', '\u0070',
'\u0071', '\u0072', '\u0073', '\u0074', '\u0075', '\u0076', '\u0077', '\u0078', '\u0079', '\u007A',
'\u007B', '\u007C', '\u007D', '\u007E', '\u007F', '\u20AC', '\u25A1', '\u201A', '\u0192', '\u201E',
'\u0000', '\u2020', '\u2021', '\u02C6', '\u2030', '\u0160', '\u2039', '\u0152', '\u25A1', '\u017D',
'\u25A1', '\u25A1', '\u0000', '\u0000', '\u0000', '\u0000', '\u2022', '\u0000', '\u0000', '\u02DC',
'\u2122', '\u0161', '\u203A', '\u0153', '\u25A1', '\u017E', '\u0178', '\u00A0', '\u00A1', '\u00A2',
'\u00A3', '\u00A4', '\u00A5', '\u00A6', '\u00A7', '\u00A8', '\u00A9', '\u00AA', '\u00AB', '\u00AC',
'\u00AD', '\u00AE', '\u00AF', '\u0000', '\u00B1', '\u00B2', '\u00B3', '\u00B4', '\u00B5', '\u00B6',
'\u00B7', '\u00B8', '\u00B9', '\u00BA', '\u00BB', '\u00BC', '\u00BD', '\u00BE', '\u00BF', '\u00C0',
'\u00C1', '\u00C2', '\u00C3', '\u00C4', '\u00C5', '\u00C6', '\u00C7', '\u00C8', '\u00C9', '\u00CA',
'\u00CB', '\u00CC', '\u00CD', '\u00CE', '\u00CF', '\u00D0', '\u00D1', '\u00D2', '\u00D3', '\u00D4',
'\u00D5', '\u00D6', '\u00D7', '\u00D8', '\u00D9', '\u00DA', '\u00DB', '\u00DC', '\u00DD', '\u00DE',
'\u00DF', '\u00E0', '\u00E1', '\u00E2', '\u00E3', '\u00E4', '\u00E5', '\u00E6', '\u00E7', '\u00E8',
'\u00E9', '\u00EA', '\u00EB', '\u00EC', '\u00ED', '\u00EE', '\u00EF', '\u00F0', '\u00F1', '\u00F2',
'\u00F3', '\u00F4', '\u00F5', '\u00F6', '\u00F7', '\u00F8', '\u00F9', '\u00FA', '\u00FB', '\u00FC',
'\u00FD', '\u00FE', '\u00FF' };

        /// <summary>
        /// Save the document to a PDF file.
        /// </summary>
        /// <param name="document">The <see cref="Document"/> to save.</param>
        /// <param name="fileName">The full path to the file to save. If it exists, it will be
overwritten.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the
file.</param>
        /// <param name="compressStreams">Indicates whether the streams in the PDF file should be
compressed.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If
this is provided, objects that have been drawn with a tag contained in the dictionary will become
hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#),
it is interpreted as the tag of another object in the current document; otherwise, it is interpreted
as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when
rendering the image.
        public static void SaveAsPDF(this Document document, string fileName, TextOptions textOption =
```

```
TextOptions.SubsetFonts, bool compressStreams = true, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default)
        {
            using (FileStream stream = new FileStream(fileName, FileMode.Create))
            {
                document.SaveAsPDF(stream, textOption, compressStreams, linkDestinations, filterOption);
            }
        }

        /// <summary>
        /// Defines whether the used fonts should be included in the file.
        /// </summary>
        public enum TextOptions
        {
            /// <summary>
            /// Embeds subsetted font files containing only the glyphs for the characters that have been used.
            /// </summary>
            SubsetFonts,

            /// <summary>
            /// Does not embed any font file and converts all text items into paths.
            /// </summary>
            ConvertIntoPaths
        }

        /// <summary>
        /// Determines how and whether image filters are rasterised.
        /// </summary>
        public class FilterOption
        {
            /// <summary>
            /// Defines whether image filters should be rasterised or not.
            /// </summary>
            public enum FilterOperations
            {
                /// <summary>
                /// Image filters will always be rasterised.
                /// </summary>
                RasteriseAll,

                /// <summary>
                /// All image filters will be ignored.
                /// </summary>
                IgnoreAll,

                /// <summary>
                /// All the images that should be drawn with a filter will be ignored.
                /// </summary>
                SkipAll
            }

            /// <summary>
            /// Defines whether image filters should be rasterised or not.
            /// </summary>
            public FilterOperations Operation { get; } = FilterOperations.RasteriseAll;

            /// <summary>
            /// The resolution that will be used to rasterise image filters. Depending on the value of <see cref="RasterisationResolutionRelative"/>, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.
            /// </summary>
            public double RasterisationResolution { get; } = 1;

            /// <summary>
            /// Determines whether the value of <see cref="RasterisationResolution"/> is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).
            /// </summary>
            public bool RasterisationResolutionRelative { get; } = true;

            /// <summary>
            /// The default options for image filter rasterisation.
            /// </summary>
            public static FilterOption Default = new FilterOption(FilterOperations.RasteriseAll, 1, true);

            /// <summary>
            /// Create a new <see cref="FilterOption"/> object.
            /// </summary>
            /// <param name="operation">Defines whether image filters should be rasterised or not.</param>
            /// <param name="rasterisationResolution">The resolution that will be used to rasterise image filters. Depending on the value of <see cref="RasterisationResolutionRelative"/>, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.</param>
```

```
            /// <param name="rasterisationResolutionRelative">Determines whether the value of <see cref="RasterisationResolution"/> is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).</param>
            public FilterOption(FilterOperations operation, double rasterisationResolution, bool rasterisationResolutionRelative)
            {
                this.Operation = operation;
                this.RasterisationResolution = rasterisationResolution;
                this.RasterisationResolutionRelative = rasterisationResolutionRelative;
            }
        }



        /// <summary>
        /// Save the document to a PDF stream.
        /// </summary>
        /// <param name="document">The <see cref="Document"/> to save.</param>
        /// <param name="stream">The stream to which the PDF data will be written.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the file.</param>
        /// <param name="compressStreams">Indicates whether the streams in the PDF file should be compressed.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        public static void SaveAsPDF(this Document document, Stream stream, TextOptions textOption = TextOptions.SubsetFonts, bool compressStreams = true, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default)
        {
            if (linkDestinations == null)
            {
                linkDestinations = new Dictionary<string, string>();
            }

            if (filterOption == null)
            {
                filterOption = FilterOption.Default;
            }

            long position = 0;

            List<long> objectPositions = new List<long>();

            int objectNum = 1;
            string currObject = "";

            int resourceObject = -1;

            StreamWriter sw = new StreamWriter(stream, Encoding.GetEncoding("ISO-8859-1"), 1024, true);

            //Header
            sw.Write("%PDF-1.4\n");
            position += 9;

            PDFContext[] pageContexts = new PDFContext[document.Pages.Count];

            for (int i = 0; i < document.Pages.Count; i++)
            {
                pageContexts[i] = new PDFContext(document.Pages[i].Width, document.Pages[i].Height, document.Pages[i].Background, textOption == TextOptions.ConvertIntoPaths, filterOption);
                document.Pages[i].Graphics.CopyToIGraphicsContext(pageContexts[i]);
            }

            Dictionary<string, FontFamily> allFontFamilies = GetFontFamilies(pageContexts);
            Dictionary<string, HashSet<char» usedChars = GetUsedChars(pageContexts);
            Dictionary<string, int> fontObjectNums = new Dictionary<string, int>();
            Dictionary<string, string> symbolFontIDs = new Dictionary<string, string>();
            Dictionary<string, string> nonSymbolFontIDs = new Dictionary<string, string>();
            Dictionary<string, Dictionary<char, int» symbolGlyphIndices = new Dictionary<string, Dictionary<char, int»();
            double[] alphas = GetAlphas(pageContexts);
            Dictionary<string, RasterImage> allImages = GetAllImages(pageContexts);
            Dictionary<string, int> imageObjectNums = new Dictionary<string, int>();

            int fontId = 1;

            foreach (KeyValuePair<string, FontFamily> kvp in allFontFamilies)
            {
                List<char> nonSymbol = new List<char>();
                List<char> symbol = new List<char>();

```

```
                foreach (char c in usedChars[kvp.Key])
                {
                    if (CP1252Chars.Contains(c))
                    {
                        nonSymbol.Add(c);
                    }
                    else
                    {
                        symbol.Add(c);
                    }
                }

                //Font
                if (((kvp.Value.IsStandardFamily && kvp.Value.FileName != "Symbol" && kvp.Value.FileName != "ZapfDingbats") && symbol.Count == 0) || kvp.Value.TrueTypeFile == null)
                {
                    fontObjectNums.Add("nonsymbol: " + kvp.Key, objectNum);
                    fontObjectNums.Add("symbol: " + kvp.Key, objectNum);
                    nonSymbolFontIDs.Add(kvp.Key, "F" + fontId.ToString());
                    symbolFontIDs.Add(kvp.Key, "F" + fontId.ToString());
                    objectPositions.Add(position);
                    currObject = objectNum.ToString() + " 0 obj\n« /Type /Font /Subtype /Type1 /BaseFont /" + (kvp.Value.IsStandardFamily ? kvp.Value.FileName.Replace(" ", "-") : kvp.Key) + " »\nendobj\n";
                    sw.Write(currObject);
                    position += currObject.Length;
                    objectNum++;
                    fontId++;
                }
                else
                {
                    int fontFileInd = objectNum;

                    TrueTypeFile subsettedFont = kvp.Value.TrueTypeFile.SubsetFont(new string(usedChars[kvp.Key].ToArray()));

                    Stream compressedStream;

                    if (!compressStreams)
                    {
                        compressedStream = subsettedFont.FontStream;
                    }
                    else
                    {
                        compressedStream = ZLibCompress(subsettedFont.FontStream);
                    }

                    long length = compressedStream.Length;

                    objectPositions.Add(position);
                    currObject = objectNum.ToString() + " 0 obj\n« /Length " + length.ToString() + " /Length1 " + subsettedFont.FontStream.Length.ToString();

                    if (compressStreams)
                    {
                        currObject += " /Filter [ /FlateDecode ]";
                    }

                    currObject += " »\nstream\n";
                    sw.Write(currObject);
                    position += currObject.Length;
                    sw.Flush();

                    compressedStream.Seek(0, SeekOrigin.Begin);
                    compressedStream.CopyTo(stream);

                    position += length;
                    currObject = "endstream\nendobj\n";
                    sw.Write(currObject);
                    position += currObject.Length;
                    objectNum++;


                    if (nonSymbol.Count > 0)
                    {
                        PDFFontDescriptor desc = new PDFFontDescriptor(subsettedFont, true, false);

                        int fontDescriptorInd = objectNum;
                        objectPositions.Add(position);

                        currObject = objectNum.ToString() + " 0 obj\n« /Type /FontDescriptor /FontName /" + desc.FontName + " /FontFamily (" + EscapeStringForPDF(desc.FontFamily) + ") /Flags " + desc.Flags.ToString();
                        currObject += " /FontBBox [ " + desc.FontBBox[0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + desc.FontBBox[1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
```

```
desc.FontBBox[2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + desc.FontBBox[3].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ] /ItalicAngle " + desc.ItalicAngle.ToString();
                        currObject += " /Ascent " + desc.Ascent.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " /Descent " + desc.Descent.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " /CapHeight " + desc.CapHeight.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " /StemV " + desc.StemV.ToString() + " /StemH " + desc.StemH.ToString() + " /FontFile2 " + fontFileInd.ToString() + " 0 R »\nendobj\n";
                        sw.Write(currObject);
                        position += currObject.Length;
                        objectNum++;


                        fontObjectNums.Add("nonsymbol: " + kvp.Key, objectNum);
                        nonSymbolFontIDs.Add(kvp.Key, "F" + fontId.ToString());
                        objectPositions.Add(position);

                        int firstChar = (from el in nonSymbol select Array.IndexOf(CP1252Chars, el)).Min();
                        int lastChar = (from el in nonSymbol select Array.IndexOf(CP1252Chars, el)).Max();

                        currObject = objectNum.ToString() + " 0 obj\n« /Type /Font /Subtype /TrueType /BaseFont /" + desc.FontName + " /FirstChar " + firstChar.ToString() + " /LastChar " + lastChar.ToString() + " /FontDescriptor " + fontDescriptorInd.ToString() + " 0 R /Encoding /WinAnsiEncoding /Widths [ ";

                        for (int i = firstChar; i <= lastChar; i++)
                        {
                            if (nonSymbol.Contains(CP1252Chars[i]))
                            {
                                currObject += subsettedFont.Get1000EmGlyphWidth(CP1252Chars[i]).ToString() + " ";
                            }
                            else
                            {
                                currObject += "0 ";
                            }
                        }

                        currObject += "] »\nendobj\n";
                        sw.Write(currObject);
                        position += currObject.Length;
                        objectNum++;
                        fontId++;
                    }


                    if (symbol.Count > 0)
                    {
                        PDFFontDescriptor desc = new PDFFontDescriptor(subsettedFont, true, true);

                        int fontDescriptorInd = objectNum;
                        objectPositions.Add(position);

                        currObject = objectNum.ToString() + " 0 obj\n« /Type /FontDescriptor /FontName /" + desc.FontName + " /FontFamily (" + EscapeStringForPDF(desc.FontFamily) + ") /Flags " + desc.Flags.ToString();
                        currObject += " /FontBBox [ " + desc.FontBBox[0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + desc.FontBBox[1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + desc.FontBBox[2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + desc.FontBBox[3].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ] /ItalicAngle " + desc.ItalicAngle.ToString();
                        currObject += " /Ascent " + desc.Ascent.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " /Descent " + desc.Descent.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " /CapHeight " + desc.CapHeight.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " /StemV " + desc.StemV.ToString() + " /StemH " + desc.StemH.ToString() + " /FontFile2 " + fontFileInd.ToString() + " 0 R »\nendobj\n";
                        sw.Write(currObject);
                        position += currObject.Length;
                        objectNum++;


                        Dictionary<char, int> glyphIndices = new Dictionary<char, int>();

                        for (int i = 0; i < symbol.Count; i++)
                        {
                            glyphIndices.Add(symbol[i], subsettedFont.GetGlyphIndex(symbol[i]));
                        }

                        symbolGlyphIndices.Add(kvp.Key, glyphIndices);
```

```

                        int descendantFontInd = objectNum;
                        objectPositions.Add(position);
                        currObject = objectNum.ToString() + " 0 obj\n« /Type /Font /Subtype /CIDFontType2 /BaseFont /" + desc.FontName + " /CIDSystemInfo « /Registry (Adobe) /Ordering (Identity) /Supplement 0 » /FontDescriptor " + fontDescriptorInd.ToString() + " 0 R ";
                        currObject += "/W [ ";

                        for (int i = 0; i < symbol.Count; i++)
                        {
                            currObject += glyphIndices[symbol[i]].ToString() + " [ ";
                            currObject += subsettedFont.Get1000EmGlyphWidth(symbol[i]).ToString() + " ] ";
                        }

                        currObject += "] »\nendobj\n";
                        sw.Write(currObject);
                        position += currObject.Length;
                        objectNum++;


                        string toUnicodeStream = "/CIDInit /ProcSet findresource begin\n12 dict begin\nbegincmap\n/CIDSystemInfo « /Registry (Adobe) /Ordering (UCS) /Supplement 0 » def\n";
                        toUnicodeStream += "/CMapName /Adobe-Identity-UCS def\n/CMapType 2 def\n1 begincodespacerange\n<0000> <ffff>\nendcodespacerange\n1 beginbfchar\n";
                        for (int i = 0; i < symbol.Count; i++)
                        {
                            toUnicodeStream += "<" + glyphIndices[symbol[i]].ToString("X4") + "> <" + ((int)symbol[i]).ToString("X4") + ">\n";
                        }
                        toUnicodeStream += "endbfchar\nendcmap\nCmapName currentdict /CMap defineresource pop\nend\nend\n";

                        MemoryStream uncompressedUnicode = new MemoryStream();

                        using (StreamWriter usw = new StreamWriter(uncompressedUnicode, Encoding.ASCII, 1024, true))
                        {
                            usw.Write(toUnicodeStream);
                        }

                        uncompressedUnicode.Seek(0, SeekOrigin.Begin);

                        MemoryStream compressedToUnicode;

                        if (!compressStreams)
                        {
                            compressedToUnicode = uncompressedUnicode;
                        }
                        else
                        {
                            compressedToUnicode = ZLibCompress(uncompressedUnicode);
                        }

                        long unicodeLength = compressedToUnicode.Length;

                        int toUnicodeInd = objectNum;
                        objectPositions.Add(position);
                        currObject = objectNum.ToString() + " 0 obj\n« /Length " + unicodeLength;

                        if (compressStreams)
                        {
                            currObject += " /Filter [ /FlateDecode ]";
                        }

                        currObject += " »\nstream\n";

                        sw.Write(currObject);
                        position += currObject.Length;
                        sw.Flush();

                        compressedToUnicode.WriteTo(stream);
                        position += unicodeLength;

                        currObject = "endstream\nendobj\n";
                        sw.Write(currObject);
                        position += currObject.Length;
                        objectNum++;


                        fontObjectNums.Add("symbol: " + kvp.Key, objectNum);
                        symbolFontIDs.Add(kvp.Key, "F" + fontId.ToString());
                        objectPositions.Add(position);
                        currObject = objectNum.ToString() + " 0 obj\n« /Type /Font /Subtype /Type0 /BaseFont /" + desc.FontName + " /Encoding /Identity-H /DescendantFonts [ " + descendantFontInd.ToString() + " 0 R ] /ToUnicode " + toUnicodeInd.ToString() + " 0 R »\nendobj\n";
                        sw.Write(currObject);
```

```
                        position += currObject.Length;
                        objectNum++;
                        fontId++;
                    }
                }
            }

            foreach (KeyValuePair<string, RasterImage> img in allImages)
            {
                RasterImage image = img.Value;
                int stride = image.Width * (image.HasAlpha ? 4 : 3);

                string filter = "";

                if (image.HasAlpha)
                {
                    objectPositions.Add(position);

                    filter = "";
                    MemoryStream alphaStream = new MemoryStream();

                    unsafe
                    {
                        byte* dataPointer = (byte*)image.ImageDataAddress;

                        if (compressStreams)
                        {
                            filter = "/FlateDecode";

                            for (int y = 0; y < image.Height; y++)
                            {
                                for (int x = 0; x < image.Width; x++)
                                {
                                    dataPointer += 3;
                                    alphaStream.WriteByte(*dataPointer);
                                    dataPointer++;
                                }
                            }

                            alphaStream.Seek(0, SeekOrigin.Begin);
                            MemoryStream compressed = ZLibCompress(alphaStream);
                            alphaStream.Dispose();
                            alphaStream = compressed;
                        }
                        else
                        {
                            filter = "/ASCIIHexDecode";

                            using (StreamWriter imageWriter = new StreamWriter(alphaStream, Encoding.ASCII, 1024, true))
                            {
                                for (int y = 0; y < image.Height; y++)
                                {
                                    for (int x = 0; x < image.Width; x++)
                                    {
                                        dataPointer += 3;
                                        imageWriter.Write((*dataPointer).ToString("X2"));
                                        dataPointer++;
                                    }
                                }
                            }
                        }
                    }

                    currObject = objectNum.ToString() + " 0 obj\n« /Type /XObject /Subtype /Image /Width " + image.Width.ToString() + " /Height " + image.Height.ToString() + " /ColorSpace /DeviceGray /BitsPerComponent 8 /Interpolate " + (image.Interpolate ? "true" : "false") + " /Filter " + filter + " /Length " + alphaStream.Length + " »\nstream\n";

                    sw.Write(currObject);
                    position += currObject.Length;
                    sw.Flush();

                    alphaStream.Seek(0, SeekOrigin.Begin);
                    alphaStream.CopyTo(stream);
                    position += alphaStream.Length;

                    currObject = "\nendstream\nendobj\n";
                    sw.Write(currObject);
                    position += currObject.Length;
                    objectNum++;

                    alphaStream.Dispose();

                }

                objectPositions.Add(position);
```

```
                int imageObjectNum = objectNum;

                filter = "";
                MemoryStream imageStream = new MemoryStream();

                unsafe
                {
                    byte* dataPointer = (byte*)image.ImageDataAddress;

                    if (compressStreams)
                    {
                        filter = "/FlateDecode";

                        if (image.HasAlpha)
                        {
                            for (int y = 0; y < image.Height; y++)
                            {
                                for (int x = 0; x < image.Width; x++)
                                {
                                    imageStream.WriteByte(*dataPointer);
                                    dataPointer++;
                                    imageStream.WriteByte(*dataPointer);
                                    dataPointer++;
                                    imageStream.WriteByte(*dataPointer);
                                    dataPointer++;
                                    dataPointer++;
                                }
                            }
                        }
                        else
                        {
                            for (int y = 0; y < image.Height; y++)
                            {
                                for (int x = 0; x < image.Width; x++)
                                {
                                    imageStream.WriteByte(*dataPointer);
                                    dataPointer++;
                                    imageStream.WriteByte(*dataPointer);
                                    dataPointer++;
                                    imageStream.WriteByte(*dataPointer);
                                    dataPointer++;
                                }
                            }
                        }

                        imageStream.Seek(0, SeekOrigin.Begin);
                        MemoryStream compressed = ZLibCompress(imageStream);
                        imageStream.Dispose();
                        imageStream = compressed;
                    }
                    else
                    {
                        filter = "/ASCIIHexDecode";

                        using (StreamWriter imageWriter = new StreamWriter(imageStream,
    Encoding.ASCII, 1024, true))
                        {
                            if (image.HasAlpha)
                            {
                                for (int y = 0; y < image.Height; y++)
                                {
                                    for (int x = 0; x < image.Width; x++)
                                    {
                                        imageWriter.Write((*dataPointer).ToString("X2"));
                                        dataPointer++;
                                        imageWriter.Write((*dataPointer).ToString("X2"));
                                        dataPointer++;
                                        imageWriter.Write((*dataPointer).ToString("X2"));
                                        dataPointer++;
                                        dataPointer++;
                                    }
                                }
                            }
                            else
                            {
                                for (int y = 0; y < image.Height; y++)
                                {
                                    for (int x = 0; x < image.Width; x++)
                                    {
                                        imageWriter.Write((*dataPointer).ToString("X2"));
                                        dataPointer++;
                                        imageWriter.Write((*dataPointer).ToString("X2"));
                                        dataPointer++;
                                        imageWriter.Write((*dataPointer).ToString("X2"));
                                        dataPointer++;
                                    }
                                }
```

```
                            }
                        }
                    }
                }

                currObject = objectNum.ToString() + " 0 obj\n« /Type /XObject /Subtype /Image /Width " + image.Width.ToString() + " /Height " + image.Height.ToString() + " /ColorSpace /DeviceRGB /BitsPerComponent 8 /Interpolate " + (image.Interpolate ? "true" : "false") + " /Filter " + filter + " /Length " + imageStream.Length + (image.HasAlpha ? " /SMask " + (objectNum - 1) + " 0 R" : "") + " »\nstream\n";

                sw.Write(currObject);
                position += currObject.Length;
                sw.Flush();

                imageStream.Seek(0, SeekOrigin.Begin);
                imageStream.CopyTo(stream);
                position += imageStream.Length;

                currObject = "\nendstream\nendobj\n";
                sw.Write(currObject);
                position += currObject.Length;
                objectNum++;

                imageStream.Dispose();

                imageObjectNums.Add(img.Key, imageObjectNum);
            }

            int fontListObject = -1;

            if (allFontFamilies.Count > 0)
            {
                //Fonts
                objectPositions.Add(position);
                fontListObject = objectNum;
                currObject = objectNum.ToString() + " 0 obj\n« ";
                foreach (KeyValuePair<string, string> kvp in nonSymbolFontIDs)
                {
                    currObject += "/" + kvp.Value + " " + fontObjectNums["nonsymbol: " + kvp.Key].ToString() + " 0 R ";
                }
                foreach (KeyValuePair<string, string> kvp in symbolFontIDs)
                {
                    currObject += "/" + kvp.Value + " " + fontObjectNums["symbol: " + kvp.Key].ToString() + " 0 R ";
                }
                currObject += "»\nendobj\n";
                sw.Write(currObject);
                position += currObject.Length;
                objectNum++;
            }


            int[] pageContentInd = new int[document.Pages.Count];


            List<(string, List<(double, double, double, double)>)>[] taggedObjectRectsByPage = new List<(string, List<(double, double, double, double)>)>[document.Pages.Count];
            Dictionary<string, int>[] taggedObjectRectsIndicesByPage = new Dictionary<string, int>[document.Pages.Count];
            List<(GradientBrush, double[,], IFigure)> gradients = new System.Collections.Generic.List<(GradientBrush, double[,], IFigure)>();


            for (int pageInd = 0; pageInd < document.Pages.Count; pageInd++)
            {
                taggedObjectRectsByPage[pageInd] = new List<(string, List<(double, double, double, double)>)>();
                taggedObjectRectsIndicesByPage[pageInd] = new Dictionary<string, int>();

                double[,] transformationMatrix = new double[3, 3] { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };
                Stack<double[,]> savedStates = new Stack<double[,]>();

                MemoryStream contentStream = new MemoryStream();

                using (StreamWriter ctW = new StreamWriter(contentStream, Encoding.ASCII, 1024, true))
                {
                    for (int i = 0; i < pageContexts[pageInd]._figures.Count; i++)
                    {
                        bool isTransform = pageContexts[pageInd]._figures[i] is TransformFigure;

                        bool isClip = pageContexts[pageInd]._figures[i] is PathFigure pathFig && pathFig.IsClipping;

                        if (!string.IsNullOrEmpty(pageContexts[pageInd]._figures[i].Tag) &&
```

```
!isTransform && !isClip)
                        {
                            (double, double, double, double) boundingRect = MeasureFigure(pageContexts[pageInd]._figures[i], ref transformationMatrix, savedStates);

                            if (!taggedObjectRectsIndicesByPage[pageInd].TryGetValue(pageContexts[pageInd]._figures[i].Tag, out int index))
                            {
                                taggedObjectRectsByPage[pageInd].Add((pageContexts[pageInd]._figures[i].Tag, new List<(double, double, double, double)> { boundingRect }));
                                taggedObjectRectsIndicesByPage[pageInd][pageContexts[pageInd]._figures[i].Tag] = taggedObjectRectsByPage[pageInd].Count - 1;
                            }
                            else
                            {
                                (string, List<(double, double, double, double)>) previousRect = taggedObjectRectsByPage[pageInd][index];
                                taggedObjectRectsByPage[pageInd][index].Item2.Add(boundingRect);
                            }
                        }
                        else if (isTransform)
                        {
                            MeasureFigure(pageContexts[pageInd]._figures[i], ref transformationMatrix, savedStates);
                        }

                        ctW.Write(FigureAsPDFString(pageContexts[pageInd]._figures[i], nonSymbolFontIDs, symbolFontIDs, symbolGlyphIndices, alphas, imageObjectNums, transformationMatrix, gradients));
                    }
                }

                //Contents
                objectPositions.Add(position);
                contentStream.Seek(0, SeekOrigin.Begin);

                MemoryStream compressedStream;

                if (!compressStreams)
                {
                    compressedStream = contentStream;
                }
                else
                {
                    compressedStream = ZLibCompress(contentStream);
                }

                long streamLength = compressedStream.Length;

                pageContentInd[pageInd] = objectNum;
                currObject = objectNum.ToString() + " 0 obj\n« /Length " + streamLength.ToString(System.Globalization.CultureInfo.InvariantCulture);

                if (compressStreams)
                {
                    currObject += " /Filter [ /FlateDecode ]";
                }

                currObject += " »\nstream\n";

                sw.Write(currObject);
                sw.Flush();

                position += currObject.Length;
                compressedStream.WriteTo(stream);
                position += streamLength;

                compressedStream.Dispose();

                currObject = "endstream\nendobj\n";
                sw.Write(currObject);
                position += currObject.Length;

                objectNum++;
            }

            List<int> gradientIndices = new List<int>(gradients.Count);
            List<int> gradientAlphaIndices = new List<int>(gradients.Count);
            List<int> gradientMaskIndices = new List<int>(gradients.Count);

            if (gradients.Count > 0)
            {
                for (int i = 0; i < gradients.Count; i++)
                {
```

```
                    (GradientBrush gradient, double[,] matrix, IFigure figure) = gradients[i];

                    //int functionObject = -1;

                    bool hasAlpha = false;

                    /*if (gradient.GradientStops.Count == 2)
                    {
                        objectPositions.Add(position);

                        currObject = objectNum.ToString() + " 0 obj\n« /FunctionType 2 /Domain [ 0 1 ] /C0 [ " + gradient.GradientStops[0].Colour.R.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[0].Colour.G.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[0].Colour.B.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] ";
                        currObject += "/C1 [ " + gradient.GradientStops[1].Colour.R.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[1].Colour.G.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[1].Colour.B.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] /N 1 »\nendobj\n";
                        sw.Write(currObject);
                        functionObject = objectNum;

                        position += currObject.Length;
                        objectNum++;

                        hasAlpha = gradient.GradientStops[0].Colour.A != 1 || gradient.GradientStops[1].Colour.A != 1;
                    }
                    else
                    {
                        List<double> bounds = new List<double>();
                        List<int> functionIndices = new List<int>();

                        for (int j = 0; j < gradient.GradientStops.Count - 1; j++)
                        {
                            objectPositions.Add(position);

                            currObject = objectNum.ToString() + " 0 obj\n« /FunctionType 2 /Domain [ 0 1 ] /C0 [ " + gradient.GradientStops[j].Colour.R.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[j].Colour.G.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[j].Colour.B.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] ";
                            currObject += "/C1 [ " + gradient.GradientStops[j + 1].Colour.R.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[j + 1].Colour.G.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[j + 1].Colour.B.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] /N 1 »\nendobj\n";
                            sw.Write(currObject);
                            functionIndices.Add(objectNum);

                            if (j < gradient.GradientStops.Count - 2)
                            {
                                bounds.Add(gradient.GradientStops[j + 1].Offset);
                            }

                            position += currObject.Length;
                            objectNum++;

                            if (gradient.GradientStops[j].Colour.A != 1)
                            {
                                hasAlpha = true;
                            }
                        }

                        objectPositions.Add(position);

                        currObject = objectNum.ToString() + " 0 obj\n« /FunctionType 3 /Domain [ 0 1 ] /Functions [ ";

                        for (int j = 0; j < functionIndices.Count; j++)
                        {
                            currObject += functionIndices[j].ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 R ";
                        }

                        currObject += "] /Bounds [ ";

                        for (int j = 0; j < bounds.Count; j++)
                        {
                            currObject += bounds[j].ToString(System.Globalization.CultureInfo.InvariantCulture) + " ";
                        }

                        currObject += "] /Encode [ ";
```

```

                        for (int j = 0; j < functionIndices.Count; j++)
                        {
                            currObject += "0 1 ";
                        }

                        currObject += "] »\nendobj\n";


                        sw.Write(currObject);
                        functionObject = objectNum;

                        position += currObject.Length;
                        objectNum++;

                    }

                    if (gradient is LinearGradientBrush linear)
                    {
                        objectPositions.Add(position);

                        currObject = objectNum.ToString() + " 0 obj\n« /Type /Pattern /PatternType 2 /Matrix [ " +

                        matrix[0, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[1, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[0, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[1, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[0, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[1, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ] ";

                        currObject += "/Shading « /ShadingType 2 /ColorSpace /DeviceRGB /Coords [ " + linear.StartPoint.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + linear.StartPoint.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + linear.EndPoint.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + linear.EndPoint.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] ";

                        currObject += "/Domain [ 0 1 ] /Extend [ true true ] /Function " + functionObject.ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 R » »\nendobj\n";
                        sw.Write(currObject);
                        gradientIndices.Add(objectNum);

                        position += currObject.Length;
                        objectNum++;
                    }
                    else if (gradient is RadialGradientBrush radial)
                    {
                        objectPositions.Add(position);

                        currObject = objectNum.ToString() + " 0 obj\n« /Type /Pattern /PatternType 2 /Matrix [ " +

                        matrix[0, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[1, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[0, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[1, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[0, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[1, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ] ";

                        currObject += "/Shading « /ShadingType 3 /ColorSpace /DeviceRGB /Coords [ " + radial.FocalPoint.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + radial.FocalPoint.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 " + radial.Centre.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + radial.Centre.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + radial.Radius.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] ";

                        currObject += "/Domain [ 0 1 ] /Extend [ true true ] /Function " + functionObject.ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 R » »\nendobj\n";
                        sw.Write(currObject);
                        gradientIndices.Add(objectNum);

                        position += currObject.Length;
                        objectNum++;
                    }*/

```

```
                    WriteGradient(true, ref gradient, ref objectPositions, ref position, ref currObject, ref objectNum, ref sw, ref hasAlpha, ref matrix, ref gradientIndices);

                    if (!hasAlpha)
                    {
                        gradientAlphaIndices.Add(-1);
                        gradientMaskIndices.Add(-1);
                    }
                    else
                    {
                        /*objectPositions.Add(position);
                        int alphaGradientIndex = objectNum;


                        **/

                        GradientBrush alphaGradient = (GradientBrush)PDFContext.GetAlphaBrush(gradient);

                        bool hasAlpha2 = false;

                        WriteGradient(false, ref alphaGradient, ref objectPositions, ref position, ref currObject, ref objectNum, ref sw, ref hasAlpha2, ref matrix, ref gradientAlphaIndices);

                        int alphaGradientIndex = gradientAlphaIndices[gradientAlphaIndices.Count - 1];

                        Rectangle bbox = figure.GetBounds();

                        MemoryStream contentStream = new MemoryStream();

                        using (StreamWriter ctW = new StreamWriter(contentStream, Encoding.ASCII, 1024, true))
                        {
                            ctW.Write("q\n");
                            ctW.Write(bbox.Location.X.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + bbox.Location.Y.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + bbox.Size.Width.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + bbox.Size.Height.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " re\n");
                            ctW.Write("/Pattern cs\n");
                            ctW.Write("/pa" + gradientAlphaIndices.Count + " scn\n");
                            ctW.Write("f\n");
                            ctW.Write("Q\n");
                        }

                        contentStream.Seek(0, SeekOrigin.Begin);

                        MemoryStream compressedStream;

                        if (!compressStreams)
                        {
                            compressedStream = contentStream;
                        }
                        else
                        {
                            compressedStream = ZLibCompress(contentStream);
                        }

                        long streamLength = compressedStream.Length;

                        objectPositions.Add(position);
                        int maskIndex = objectNum;

                        currObject = objectNum.ToString() + " 0 obj\n« /Type /XObject /Subtype /Form " +
                            "/Group « /Type /Group /S /Transparency /I true /CS /DeviceRGB » " +
                            "/BBox [ " +
                            bbox.Location.X.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + bbox.Location.Y.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + (bbox.Location.X + bbox.Size.Width).ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + (bbox.Location.Y + bbox.Size.Height).ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) +
                            " ] " + /*"/Matrix [ " +
                        matrix[0, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[1, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[0, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[1, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[0, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                        matrix[1, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ] " +*/
```

```
                        "/Resources « /Pattern « /pa" + gradientAlphaIndices.Count + " " + alphaGradientIndex.ToString() + " 0 R » » " +

                        "/Length " + streamLength.ToString();

                        if (compressStreams)
                        {
                            if (compressStreams)
                            {
                                currObject += " /Filter [ /FlateDecode ]";
                            }
                        }

                        currObject += " »\nstream\n";

                        sw.Write(currObject);
                        sw.Flush();

                        position += currObject.Length;
                        compressedStream.WriteTo(stream);
                        position += streamLength;

                        compressedStream.Dispose();

                        currObject = "endstream\nendobj\n";
                        sw.Write(currObject);
                        position += currObject.Length;

                        objectNum++;


                        objectPositions.Add(position);
                        int actualMaskIndex = objectNum;

                        gradientMaskIndices.Add(actualMaskIndex);

                        currObject = objectNum.ToString() + " 0 obj\n« /Type /ExtGState /SMask « /Type /Mask /S /Luminosity /G " + maskIndex.ToString() + " 0 R » »\nendobj\n";
                        sw.Write(currObject);
                        position += currObject.Length;
                        objectNum++;
                    }

                }
            }

            if (allFontFamilies.Count > 0)
            {

                //Resources
                objectPositions.Add(position);
                resourceObject = objectNum;
                currObject = objectNum.ToString() + " 0 obj\n« /Font " + fontListObject.ToString() + " 0 R";

                if (alphas.Length > 0 || gradientMaskIndices.Where(x => x >= 0).Any())
                {
                    currObject += " /ExtGState «\n";

                    for (int i = 0; i < alphas.Length; i++)
                    {
                        currObject += "/a" + i.ToString() + " « /CA " + alphas[i].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " /ca " + alphas[i].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " »\n";
                    }

                    for (int i = 0; i < gradientMaskIndices.Count; i++)
                    {
                        if (gradientMaskIndices[i] >= 0)
                        {
                            currObject += "/ma" + i.ToString() + " " + gradientMaskIndices[i].ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 R\n";
                        }
                    }

                    currObject += "»";
                }

                if (imageObjectNums.Count > 0)
                {
                    currObject += " /XObject «";

                    foreach (KeyValuePair<string, int> kvp in imageObjectNums)
                    {
                        currObject += " /Img" + kvp.Value.ToString() + " " + kvp.Value.ToString() + " 0 R";
```

```
                    }

                    currObject += " »";
                }

                if (gradientIndices.Count > 0)
                {
                    currObject += " /Pattern « ";

                    for (int i = 0; i < gradientIndices.Count; i++)
                    {
                        currObject += "/p" + i.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradientIndices[i].ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 R ";
                    }

                    currObject += "»";
                }

                currObject += " »\nendobj\n";
                sw.Write(currObject);
                position += currObject.Length;
                objectNum++;
            }
            else
            {
                //Resources
                objectPositions.Add(position);
                resourceObject = objectNum;
                currObject = objectNum.ToString() + " 0 obj\n«";

                if (alphas.Length > 0 || gradientMaskIndices.Where(x => x >= 0).Any())
                {
                    currObject += " /ExtGState «\n";

                    for (int i = 0; i < alphas.Length; i++)
                    {
                        currObject += "/a" + i.ToString() + " « /CA " + alphas[i].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " /ca " + alphas[i].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " »\n";
                    }

                    for (int i = 0; i < gradientMaskIndices.Count; i++)
                    {
                        if (gradientMaskIndices[i] >= 0)
                        {
                            currObject += "/ma" + i.ToString() + " " + gradientMaskIndices[i].ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 R\n";
                        }
                    }

                    currObject += "»";
                }

                if (imageObjectNums.Count > 0)
                {
                    currObject += " /XObject «";

                    foreach (KeyValuePair<string, int> kvp in imageObjectNums)
                    {
                        currObject += " /Img" + kvp.Value.ToString() + " " + kvp.Value.ToString() + " 0 R";
                    }

                    currObject += " »";
                }

                if (gradientIndices.Count > 0)
                {
                    currObject += " /Pattern « ";

                    for (int i = 0; i < gradientIndices.Count; i++)
                    {
                        currObject += "/p" + i.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradientIndices[i].ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 R ";
                    }

                    currObject += "»";
                }

                currObject += " »\nendobj\n";
                sw.Write(currObject);
                position += currObject.Length;
                objectNum++;
            }
```

```

            //Catalog
            objectPositions.Add(position);
            int rootObject = objectNum;
            currObject = objectNum.ToString() + " 0 obj\n« /Type /Catalog /Pages " + (objectNum + 1).ToString() + " 0 R »\nendobj\n";
            sw.Write(currObject);
            position += currObject.Length;
            objectNum++;

            objectPositions.Add(position);
            int pageParent = objectNum;
            objectNum++;

            List<int> pageObjectNums = new List<int>();

            //We do not have enough information to resolve all relative links yet (we need the object number for all the pages).
            List<(int annotationObjectNum, (double, double, double, double) annotationOrigin, int annotationDestinationPage, (double, double, double, double) annotationDestination)> postponedAnnotations = new List<(int, (double, double, double, double), int, (double, double, double, double))>();

            //Page
            for (int i = 0; i < document.Pages.Count; i++)
            {
                List<int> annotationsToInclude = new List<int>();

                //Annotations
                for (int j = 0; j < taggedObjectRectsByPage[i].Count; j++)
                {
                    if (linkDestinations.TryGetValue(taggedObjectRectsByPage[i][j].Item1, out string destination))
                    {
                        if (destination.StartsWith("#"))
                        {
                            //Leave these for later, once we have computed the object number for all the pages. But we need to include the annotation number in the page, so we start processing the annotation now.
                            for (int k = 0; k < taggedObjectRectsIndicesByPage.Length; k++)
                            {
                                if (taggedObjectRectsIndicesByPage[k].TryGetValue(destination.Substring(1), out int index))
                                {
                                    for (int l = 0; l < taggedObjectRectsByPage[i][j].Item2.Count; l++)
                                    {
                                        objectPositions.Add(position);
                                        annotationsToInclude.Add(objectNum);
                                        postponedAnnotations.Add((objectNum, taggedObjectRectsByPage[i][j].Item2[l], k, taggedObjectRectsByPage[k][index].Item2[0]));
                                        objectNum++;
                                    }

                                    //Only consider the first match for the local destination.
                                    break;
                                }
                            }
                        }
                        else
                        {
                            for (int l = 0; l < taggedObjectRectsByPage[i][j].Item2.Count; l++)
                            {
                                objectPositions.Add(position);
                                currObject = objectNum.ToString() + " 0 obj\n« /Type /Annot /Subtype /Link /Rect [" + taggedObjectRectsByPage[i][j].Item2[l].Item1.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + taggedObjectRectsByPage[i][j].Item2[l].Item2.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + taggedObjectRectsByPage[i][j].Item2[l].Item3.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + taggedObjectRectsByPage[i][j].Item2[l].Item4.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + "] " +
                                    "/A « /Type /Action /S /URI /URI (" + destination + ") » »\nendobj\n";
                                sw.Write(currObject);
                                annotationsToInclude.Add(objectNum);
                                objectNum++;
                                position += currObject.Length;
                            }
                        }
                    }
                }

                //Page
                objectPositions.Add(position);
                currObject = objectNum.ToString() + " 0 obj\n« /Type /Page /Parent " +
```

```
pageParent.ToString() + " 0 R /MediaBox [0 0 " +
document.Pages[i].Width.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " " +
document.Pages[i].Height.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + "] /Resources " + resourceObject.ToString() + "
0 R /Contents " + pageContentInd[i].ToString() + " 0 R ";

                if (annotationsToInclude.Count > 0)
                {
                    StringBuilder annotations = new StringBuilder();
                    annotations.Append("/Annots [ ");
                    for (int j = 0; j < annotationsToInclude.Count; j++)
                    {
                        annotations.Append(annotationsToInclude[j].ToString() + " 0 R ");
                    }
                    annotations.Append("] ");
                    currObject += annotations.ToString();
                }

                currObject += "»\nendobj\n";

                sw.Write(currObject);
                pageObjectNums.Add(objectNum);
                objectNum++;
                position += currObject.Length;
            }

            //Now we have enough information for the postponed annotations.
            foreach ((int annotationObjectNum, (double, double, double, double) annotationOrigin, int
annotationDestinationPage, (double, double, double, double) annotationDestination) in
postponedAnnotations)
            {
                objectPositions[annotationObjectNum - 1] = position;
                currObject = annotationObjectNum.ToString() + " 0 obj\n« /Type /Annot /Subtype /Link
/Rect [" + annotationOrigin.Item1.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " " +
annotationOrigin.Item2.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " " +
annotationOrigin.Item3.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " " +
annotationOrigin.Item4.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + "] " +
                    "/Dest [ " + pageObjectNums[annotationDestinationPage].ToString() + " 0 R /XYZ " +
annotationDestination.Item1.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " " +
annotationDestination.Item4.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " 0 ] »\nendobj\n";
                sw.Write(currObject);
                position += currObject.Length;
            }

            //Pages
            objectPositions[pageParent - 1] = position;
            currObject = pageParent.ToString() + " 0 obj\n« /Type /Pages /Kids [ ";
            for (int i = 0; i < document.Pages.Count; i++)
            {
                //currObject += (pageParent + i + 1).ToString() + " 0 R ";
                currObject += pageObjectNums[i].ToString() + " 0 R ";
            }
            currObject += "] /Count " + document.Pages.Count + " »\nendobj\n\n";
            sw.Write(currObject);
            position += currObject.Length;

            //XRef
            sw.Write("xref\n0 " + (objectPositions.Count + 1).ToString() + "\n0000000000 65535 f \n");
            for (int i = 0; i < objectPositions.Count; i++)
            {
                sw.Write(objectPositions[i].ToString("0000000000",
System.Globalization.CultureInfo.InvariantCulture) + " 00000 n \n");
            }

            //Trailer
            sw.Write("trailer\n« /Size " + (objectPositions.Count + 1).ToString() + " /Root " +
rootObject.ToString() + " 0 R »\nstartxref\n" + position.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + "\n%%EOF\n");

            sw.Flush();
            sw.Dispose();
        }

        private static void WriteGradient(bool includeMatrix, ref GradientBrush gradient, ref
List<long> objectPositions, ref long position, ref string currObject, ref int objectNum, ref
StreamWriter sw, ref bool hasAlpha, ref double[,] matrix, ref List<int> gradientIndices)
        {
            int functionObject = -1;

            if (gradient.GradientStops.Count == 2)
```

```
            {
                objectPositions.Add(position);

                currObject = objectNum.ToString() + " 0 obj\n« /FunctionType 2 /Domain [ 0 1 ] /C0 [ " + gradient.GradientStops[0].Colour.R.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[0].Colour.G.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[0].Colour.B.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] ";
                currObject += "/C1 [ " + gradient.GradientStops[1].Colour.R.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[1].Colour.G.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[1].Colour.B.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] /N 1 »\nendobj\n";
                sw.Write(currObject);
                functionObject = objectNum;

                position += currObject.Length;
                objectNum++;

                hasAlpha = gradient.GradientStops[0].Colour.A != 1 || gradient.GradientStops[1].Colour.A != 1;
            }
            else
            {
                List<double> bounds = new List<double>();
                List<int> functionIndices = new List<int>();

                for (int j = 0; j < gradient.GradientStops.Count - 1; j++)
                {
                    objectPositions.Add(position);

                    currObject = objectNum.ToString() + " 0 obj\n« /FunctionType 2 /Domain [ 0 1 ] /C0 [ " + gradient.GradientStops[j].Colour.R.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[j].Colour.G.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[j].Colour.B.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] ";
                    currObject += "/C1 [ " + gradient.GradientStops[j + 1].Colour.R.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[j + 1].Colour.G.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + gradient.GradientStops[j + 1].Colour.B.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] /N 1 »\nendobj\n";
                    sw.Write(currObject);
                    functionIndices.Add(objectNum);

                    if (j < gradient.GradientStops.Count - 2)
                    {
                        bounds.Add(gradient.GradientStops[j + 1].Offset);
                    }

                    position += currObject.Length;
                    objectNum++;

                    if (gradient.GradientStops[j].Colour.A != 1)
                    {
                        hasAlpha = true;
                    }
                }

                if (gradient.GradientStops[gradient.GradientStops.Count - 1].Colour.A != 1)
                {
                    hasAlpha = true;
                }

                objectPositions.Add(position);

                currObject = objectNum.ToString() + " 0 obj\n« /FunctionType 3 /Domain [ 0 1 ] /Functions [ ";

                for (int j = 0; j < functionIndices.Count; j++)
                {
                    currObject += functionIndices[j].ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 R ";
                }

                currObject += "] /Bounds [ ";

                for (int j = 0; j < bounds.Count; j++)
                {
                    currObject += bounds[j].ToString(System.Globalization.CultureInfo.InvariantCulture) + " ";
                }

                currObject += "] /Encode [ ";

                for (int j = 0; j < functionIndices.Count; j++)
                {
```

```
                    currObject += "0 1 ";
                }

                currObject += "] »\nendobj\n";


                sw.Write(currObject);
                functionObject = objectNum;

                position += currObject.Length;
                objectNum++;

            }

            if (gradient is LinearGradientBrush linear)
            {
                objectPositions.Add(position);

                currObject = objectNum.ToString() + " 0 obj\n« /Type /Pattern /PatternType 2 ";

                if (includeMatrix)
                {
                    currObject += "/Matrix [ " +

                    matrix[0, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                    matrix[1, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                    matrix[0, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                    matrix[1, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                    matrix[0, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                    matrix[1, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ] ";
                }

                currObject += "/Shading « /ShadingType 2 /ColorSpace /DeviceRGB /Coords [ " + linear.StartPoint.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + linear.StartPoint.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + linear.EndPoint.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + linear.EndPoint.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] ";

                currObject += "/Domain [ 0 1 ] /Extend [ true true ] /Function " + functionObject.ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 R » »\nendobj\n";
                sw.Write(currObject);
                gradientIndices.Add(objectNum);

                position += currObject.Length;
                objectNum++;
            }
            else if (gradient is RadialGradientBrush radial)
            {
                objectPositions.Add(position);

                currObject = objectNum.ToString() + " 0 obj\n« /Type /Pattern /PatternType 2 ";

                if (includeMatrix)
                {
                    currObject += "/Matrix [ " +

                    matrix[0, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                    matrix[1, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                    matrix[0, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                    matrix[1, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                    matrix[0, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " +
                    matrix[1, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ] ";
                }

                currObject += "/Shading « /ShadingType 3 /ColorSpace /DeviceRGB /Coords [ " + radial.FocalPoint.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + radial.FocalPoint.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 " + radial.Centre.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + radial.Centre.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + radial.Radius.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ] ";

                currObject += "/Domain [ 0 1 ] /Extend [ true true ] /Function " + functionObject.ToString(System.Globalization.CultureInfo.InvariantCulture) + " 0 R » »\nendobj\n";
                sw.Write(currObject);
                gradientIndices.Add(objectNum);
```

```
                position += currObject.Length;
                objectNum++;
            }
        }

        private static (double, double, double, double) MeasureFigure(IFigure figure, ref double[,] transformationMatrix, Stack<double[,]> savedStates)
        {
            if (figure is PathFigure)
            {
                PathFigure fig = figure as PathFigure;

                double minX = double.MaxValue;
                double maxX = double.MinValue;
                double minY = double.MaxValue;
                double maxY = double.MinValue;

                for (int i = 0; i < fig.Segments.Length; i++)
                {
                    switch (fig.Segments[i].Type)
                    {
                        case SegmentType.Move:
                            {
                                Point pt = transformationMatrix.Multiply(fig.Segments[i].Point);
                                minX = Math.Min(minX, pt.X);
                                minY = Math.Min(minY, pt.Y);
                                maxX = Math.Max(maxX, pt.X);
                                maxY = Math.Max(maxY, pt.Y);
                            }
                            break;
                        case SegmentType.Line:
                            {
                                Point pt = transformationMatrix.Multiply(fig.Segments[i].Point);
                                minX = Math.Min(minX, pt.X);
                                minY = Math.Min(minY, pt.Y);
                                maxX = Math.Max(maxX, pt.X);
                                maxY = Math.Max(maxY, pt.Y);
                            }
                            break;
                        case SegmentType.CubicBezier:
                            for (int j = 0; j < fig.Segments[i].Points.Length; j++)
                            {
                                Point pt = transformationMatrix.Multiply(fig.Segments[i].Points[j]);
                                minX = Math.Min(minX, pt.X);
                                minY = Math.Min(minY, pt.Y);
                                maxX = Math.Max(maxX, pt.X);
                                maxY = Math.Max(maxY, pt.Y);
                            }
                            break;
                        case SegmentType.Close:
                            break;
                    }
                }

                return (minX, minY, maxX, maxY);
            }
            else if (figure is TextFigure)
            {
                TextFigure fig = figure as TextFigure;

                double realX = fig.Position.X;

                if (fig.Font.FontFamily.TrueTypeFile != null)
                {
                    realX = fig.Position.X - fig.Font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(fig.Text[0]).LeftSideBearing * fig.Font.FontSize / 1000;
                }

                double yMax = 0;
                double yMin = 0;

                if (fig.Font.FontFamily.TrueTypeFile != null)
                {
                    for (int i = 0; i < fig.Text.Length; i++)
                    {
                        TrueTypeFile.VerticalMetrics vMet = fig.Font.FontFamily.TrueTypeFile.Get1000EmGlyphVerticalMetrics(fig.Text[i]);
                        yMin = Math.Min(yMin, vMet.YMin * fig.Font.FontSize / 1000);
                        yMax = Math.Max(yMax, vMet.YMax * fig.Font.FontSize / 1000);
                    }
                }

                double realY = fig.Position.Y;

                if (fig.TextBaseline == TextBaselines.Bottom)
```

```
                {
                    realY -= yMax;
                }
                else if (fig.TextBaseline == TextBaselines.Top)
                {
                    realY -= yMin;
                }
                else if (fig.TextBaseline == TextBaselines.Middle)
                {
                    realY -= (yMax + yMin) * 0.5;
                }
                else if (fig.TextBaseline == TextBaselines.Baseline)
                {
                    realY -= yMax + yMin;
                }

                Font.DetailedFontMetrics metrics = fig.Font.MeasureTextAdvanced(fig.Text);

                Point corner1 = new Point(fig.Position.X - metrics.LeftSideBearing, realY + metrics.Top);
                Point corner2 = new Point(fig.Position.X + metrics.Width, realY + metrics.Top);
                Point corner3 = new Point(fig.Position.X + metrics.Width, realY + metrics.Bottom);
                Point corner4 = new Point(fig.Position.X - metrics.LeftSideBearing, realY + metrics.Bottom);

                corner1 = transformationMatrix.Multiply(corner1);
                corner2 = transformationMatrix.Multiply(corner2);
                corner3 = transformationMatrix.Multiply(corner3);
                corner4 = transformationMatrix.Multiply(corner4);

                return (Math.Min(corner1.X, Math.Min(corner2.X, Math.Min(corner3.X, corner4.X))), Math.Min(corner1.Y, Math.Min(corner2.Y, Math.Min(corner3.Y, corner4.Y))), Math.Max(corner1.X, Math.Max(corner2.X, Math.Max(corner3.X, corner4.X))), Math.Max(corner1.Y, Math.Max(corner2.Y, Math.Max(corner3.Y, corner4.Y))));
            }
            else if (figure is TransformFigure transf)
            {
                if (transf.TransformType == TransformFigure.TransformTypes.Transform)
                {
                    transformationMatrix = transformationMatrix.Multiply(transf.TransformationMatrix);
                }
                else if (transf.TransformType == TransformFigure.TransformTypes.Save)
                {
                    savedStates.Push(transformationMatrix);
                }
                else if (transf.TransformType == TransformFigure.TransformTypes.Restore)
                {
                    transformationMatrix = savedStates.Pop();
                }

                return (0, 0, 0, 0);
            }
            else if (figure is RasterImageFigure)
            {
                Point corner1 = new Point(0, 0);
                Point corner2 = new Point(0, 1);
                Point corner3 = new Point(1, 1);
                Point corner4 = new Point(1, 0);

                corner1 = transformationMatrix.Multiply(corner1);
                corner2 = transformationMatrix.Multiply(corner2);
                corner3 = transformationMatrix.Multiply(corner3);
                corner4 = transformationMatrix.Multiply(corner4);

                return (Math.Min(corner1.X, Math.Min(corner2.X, Math.Min(corner3.X, corner4.X))), Math.Min(corner1.Y, Math.Min(corner2.Y, Math.Min(corner3.Y, corner4.Y))), Math.Max(corner1.X, Math.Max(corner2.X, Math.Max(corner3.X, corner4.X))), Math.Max(corner1.Y, Math.Max(corner2.Y, Math.Max(corner3.Y, corner4.Y))));
            }
            else
            {
                return (0, 0, 0, 0);
            }
        }

        private static double[,] Multiply(this double[,] matrix, double[,] matrix2)
        {
            double[,] tbr = new double[3, 3];

            tbr[0, 0] = matrix[0, 0] * matrix2[0, 0] - matrix[0, 1] * matrix2[1, 0] + matrix[0, 2] * matrix2[2, 0];
            tbr[0, 1] = -matrix[0, 0] * matrix2[0, 1] + matrix[0, 1] * matrix2[1, 1] + matrix[0, 2] * matrix2[2, 1];
            tbr[0, 2] = matrix[0, 0] * matrix2[0, 2] + matrix[0, 1] * matrix2[1, 2] + matrix[0, 2] * matrix2[2, 2];

            tbr[1, 0] = matrix[1, 0] * matrix2[0, 0] - matrix[1, 1] * matrix2[1, 0] + matrix[1, 2] *
```

```
matrix2[2, 0];
            tbr[1, 1] = -matrix[1, 0] * matrix2[0, 1] + matrix[1, 1] * matrix2[1, 1] + matrix[1, 2] * matrix2[2, 1];
            tbr[1, 2] = matrix[1, 0] * matrix2[0, 2] + matrix[1, 1] * matrix2[1, 2] + matrix[1, 2] * matrix2[2, 2];

            tbr[2, 0] = matrix[2, 0] * matrix2[0, 0] - matrix[2, 1] * matrix2[1, 0] + matrix[2, 2] * matrix2[2, 0];
            tbr[2, 1] = -matrix[2, 0] * matrix2[0, 1] + matrix[2, 1] * matrix2[1, 1] + matrix[2, 2] * matrix2[2, 1];
            tbr[2, 2] = matrix[2, 0] * matrix2[0, 2] + matrix[2, 1] * matrix2[1, 2] + matrix[2, 2] * matrix2[2, 2];

            return tbr;
        }

        private static Point Multiply(this double[,] matrix, Point point)
        {
            double[] transPt = new double[3];

            transPt[0] = matrix[0, 0] * point.X + matrix[0, 1] * point.Y + matrix[0, 2];
            transPt[1] = matrix[1, 0] * point.X + matrix[1, 1] * point.Y + matrix[1, 2];
            transPt[2] = matrix[2, 0] * point.X + matrix[2, 1] * point.Y + matrix[2, 2];

            return new Point(transPt[0] / transPt[2], transPt[1] / transPt[2]);
        }

        private static string FigureAsPDFString(IFigure figure, Dictionary<string, string> nonSymbolFontIds, Dictionary<string, string> symbolFontIds, Dictionary<string, Dictionary<char, int»> symbolGlyphIndices, double[] alphas, Dictionary<string, int> imageObjectNums, double[,] transformationMatrix, List<(GradientBrush, double[,], IFigure)> gradients)
        {

            StringBuilder sb = new StringBuilder();

            bool restoreAtEnd = false;

            if (figure.Fill != null)
            {
                if (figure.Fill is SolidColourBrush solid)
                {
                    sb.Append(solid.R.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + solid.G.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + solid.B.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " rg\n");
                    sb.Append("/a" + Array.IndexOf(alphas, solid.A).ToString() + " gs\n");
                }
                else if (figure.Fill is GradientBrush gradient)
                {
                    int brushIndex = gradients.Count;
                    double[,] clonedMatrix = new double[3, 3] { { transformationMatrix[0, 0], transformationMatrix[0, 1], transformationMatrix[0, 2] }, { transformationMatrix[1, 0], transformationMatrix[1, 1], transformationMatrix[1, 2] }, { transformationMatrix[2, 0], transformationMatrix[2, 1], transformationMatrix[2, 2] } };

                    gradients.Add((gradient, clonedMatrix, figure));

                    bool gradientCompatible = PDFContext.IsCompatible(gradient);

                    if (!gradientCompatible)
                    {
                        sb.Append("q\n");
                        sb.Append("/ma" + brushIndex.ToString(System.Globalization.CultureInfo.InvariantCulture) + " gs ");
                        restoreAtEnd = true;
                    }

                    sb.Append("/Pattern cs /p" + brushIndex.ToString(System.Globalization.CultureInfo.InvariantCulture) + " scn /a" + Array.IndexOf(alphas, 1.0).ToString() + " gs\n");
                }
            }

            if (figure.Stroke != null)
            {
                if (figure.Stroke is SolidColourBrush solid)
                {
                    sb.Append(solid.R.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + solid.G.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + solid.B.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " RG\n");
                    sb.Append("/a" + Array.IndexOf(alphas, solid.A).ToString() + " gs\n");
                }
                else if (figure.Stroke is GradientBrush gradient)
                {
                    int brushIndex = gradients.Count;
                    double[,] clonedMatrix = new double[3, 3] { { transformationMatrix[0, 0],
```

```
transformationMatrix[0, 1], transformationMatrix[0, 2] }, { transformationMatrix[1, 0],
transformationMatrix[1, 1], transformationMatrix[1, 2] }, { transformationMatrix[2, 0],
transformationMatrix[2, 1], transformationMatrix[2, 2] } };

                    gradients.Add((gradient, clonedMatrix, figure));

                    bool gradientCompatible = PDFContext.IsCompatible(gradient);

                    if (!gradientCompatible)
                    {
                        sb.Append("q\n");
                        sb.Append("/ma" +
brushIndex.ToString(System.Globalization.CultureInfo.InvariantCulture) + " gs ");
                        restoreAtEnd = true;
                    }

                    sb.Append("/Pattern CS /p" +
brushIndex.ToString(System.Globalization.CultureInfo.InvariantCulture) + " SCN /a" +
Array.IndexOf(alphas, 1.0).ToString() + " gs\n");
                }

                sb.Append(figure.LineWidth.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " w\n");
                sb.Append(((int)figure.LineCap).ToString() + " J\n");
                sb.Append(((int)figure.LineJoin).ToString() + " j\n");
                if (figure.LineDash.UnitsOff != 0 || figure.LineDash.UnitsOn != 0)
                {
                    sb.Append("[ " + figure.LineDash.UnitsOn.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " " +
figure.LineDash.UnitsOff.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " ] " +
figure.LineDash.Phase.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " d\n");
                }
                else
                {
                    sb.Append("[] 0 d\n");
                }
            }

            if (figure is PathFigure)
            {
                PathFigure fig = figure as PathFigure;

                for (int i = 0; i < fig.Segments.Length; i++)
                {
                    switch (fig.Segments[i].Type)
                    {
                        case SegmentType.Move:
                            {
                                sb.Append(fig.Segments[i].Point.X.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " " +
fig.Segments[i].Point.Y.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " m ");
                            }
                            break;
                        case SegmentType.Line:
                            {
                                sb.Append(fig.Segments[i].Point.X.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " " +
fig.Segments[i].Point.Y.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " l ");
                            }
                            break;
                        case SegmentType.CubicBezier:
                            for (int j = 0; j < fig.Segments[i].Points.Length; j++)
                            {
                                sb.Append(fig.Segments[i].Points[j].X.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " " +
fig.Segments[i].Points[j].Y.ToString("0.################",
System.Globalization.CultureInfo.InvariantCulture) + " ");
                            }
                            sb.Append("c ");
                            break;
                        case SegmentType.Close:
                            sb.Append("h ");
                            break;
                    }
                }

                if (fig.IsClipping)
                {
                    sb.Append("W n\n");
                }
                else
                {
                    if (fig.Fill != null)
```

```
                    {
                        sb.Append("f*\n");
                    }

                    if (fig.Stroke != null)
                    {
                        sb.Append("S\n");
                    }
                }
            }
            else if (figure is TextFigure)
            {
                TextFigure fig = figure as TextFigure;

                List<(string txt, bool isSymbolic)> segments = new List<(string txt, bool isSymbolic)>();

                StringBuilder currSeg = new StringBuilder();
                bool currSymbolic = false;

                for (int i = 0; i < fig.Text.Length; i++)
                {
                    if (CP1252Chars.Contains(fig.Text[i]))
                    {
                        if (!currSymbolic)
                        {
                            currSeg.Append(fig.Text[i]);
                        }
                        else
                        {
                            if (currSeg.Length > 0)
                            {
                                segments.Add((currSeg.ToString(), currSymbolic));
                            }

                            currSeg = new StringBuilder();
                            currSymbolic = false;
                            currSeg.Append(fig.Text[i]);
                        }
                    }
                    else
                    {
                        if (currSymbolic)
                        {
                            currSeg.Append(fig.Text[i]);
                        }
                        else
                        {
                            if (currSeg.Length > 0)
                            {
                                segments.Add((currSeg.ToString(), currSymbolic));
                            }

                            currSeg = new StringBuilder();
                            currSymbolic = true;
                            currSeg.Append(fig.Text[i]);
                        }
                    }
                }

                if (currSeg.Length > 0)
                {
                    segments.Add((currSeg.ToString(), currSymbolic));
                }



                double realX = fig.Position.X;

                if (fig.Font.FontFamily.TrueTypeFile != null)
                {
                    realX = fig.Position.X - fig.Font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(fig.Text[0]).LeftSideBearing * fig.Font.FontSize / 1000;
                }

                double yMax = 0;
                double yMin = 0;

                if (fig.Font.FontFamily.TrueTypeFile != null)
                {
                    for (int i = 0; i < fig.Text.Length; i++)
                    {
                        TrueTypeFile.VerticalMetrics vMet = fig.Font.FontFamily.TrueTypeFile.Get1000EmGlyphVerticalMetrics(fig.Text[i]);
                        yMin = Math.Min(yMin, vMet.YMin * fig.Font.FontSize / 1000);
                        yMax = Math.Max(yMax, vMet.YMax * fig.Font.FontSize / 1000);
```

```
                    }
                }

                double realY = fig.Position.Y;

                if (fig.TextBaseline == TextBaselines.Bottom)
                {
                    realY -= yMax;
                }
                else if (fig.TextBaseline == TextBaselines.Top)
                {
                    realY -= yMin;
                }
                else if (fig.TextBaseline == TextBaselines.Middle)
                {
                    realY -= (yMax + yMin) * 0.5;
                }
                else if (fig.TextBaseline == TextBaselines.Baseline)
                {
                    realY -= yMax + yMin;
                }

                double middleY = realY + (yMax + yMin) * 0.5;



                sb.Append("q\n1 0 0 1 0 " + (middleY).ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " cm\n");
                sb.Append("1 0 0 -1 0 0 cm\n");
                sb.Append("1 0 0 1 0 " + (-middleY).ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " cm\n");

                sb.Append("BT\n");

                if (figure.Stroke != null && figure.Fill != null)
                {
                    sb.Append("2 Tr\n");
                }
                else if (figure.Stroke != null)
                {
                    sb.Append("1 Tr\n");
                }
                else if (figure.Fill != null)
                {
                    sb.Append("0 Tr\n");
                }

                sb.Append(realX.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " " + realY.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " Td\n");

                for (int i = 0; i < segments.Count; i++)
                {
                    if (!segments[i].isSymbolic)
                    {
                        sb.Append("/" + nonSymbolFontIds[fig.Font.FontFamily.FamilyName] + " " + fig.Font.FontSize.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " Tf\n");
                        sb.Append(GetKernedString(segments[i].txt, fig.Font) + " TJ\n");
                    }
                    else
                    {
                        sb.Append("/" + symbolFontIds[fig.Font.FontFamily.FamilyName] + " " + fig.Font.FontSize.ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " Tf\n");
                        sb.Append("<" + EscapeSymbolStringForPDF(segments[i].txt, symbolGlyphIndices[fig.Font.FontFamily.FamilyName]) + "> Tj\n");
                    }
                }

                sb.Append("ET\nQ\n");
            }
            else if (figure is TransformFigure transf)
            {
                if (transf.TransformType == TransformFigure.TransformTypes.Transform)
                {
                    sb.Append(transf.TransformationMatrix[0, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ");
                    sb.Append(transf.TransformationMatrix[0, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ");
                    sb.Append(transf.TransformationMatrix[1, 0].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ");
                    sb.Append(transf.TransformationMatrix[1, 1].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ");
                    sb.Append(transf.TransformationMatrix[0, 2].ToString("0.################", System.Globalization.CultureInfo.InvariantCulture) + " ");
                    sb.Append(transf.TransformationMatrix[1, 2].ToString("0.################",
```

```
System.Globalization.CultureInfo.InvariantCulture) + " cm\n");
                }
                else if (transf.TransformType == TransformFigure.TransformTypes.Save)
                {
                    sb.Append("q\n");
                }
                else if (transf.TransformType == TransformFigure.TransformTypes.Restore)
                {
                    sb.Append("Q\n");
                }
            }
            else if (figure is RasterImageFigure fig)
            {
                sb.Append("/a" + Array.IndexOf(alphas, 1).ToString() + " gs\n");

                int imageNum = imageObjectNums[fig.Image.Id];

                sb.Append("/Img" + imageNum.ToString() + " Do\n");
            }

            if (restoreAtEnd)
            {
                sb.Append("Q\n");
            }

            return sb.ToString();
        }

        internal static MemoryStream ZLibCompress(Stream contentStream)
        {
            MemoryStream compressedStream = new MemoryStream();
            compressedStream.Write(new byte[] { 0x78, 0x01 }, 0, 2);

            using (DeflateStream deflate = new DeflateStream(compressedStream,
CompressionLevel.Optimal, true))
            {
                contentStream.CopyTo(deflate);
            }
            contentStream.Seek(0, SeekOrigin.Begin);

            uint checksum = Adler32(contentStream);

            compressedStream.Write(new byte[] { (byte)((checksum » 24) & 255), (byte)((checksum » 16)
& 255), (byte)((checksum » 8) & 255), (byte)(checksum & 255) }, 0, 4);

            compressedStream.Seek(0, SeekOrigin.Begin);

            return compressedStream;
        }

        internal static uint Adler32(Stream contentStream)
        {
            uint s1 = 1;
            uint s2 = 0;

            int readByte;

            while ((readByte = contentStream.ReadByte()) >= 0)
            {
                s1 = (s1 + (byte)readByte) % 65521U;
                s2 = (s2 + s1) % 65521U;
            }

            return (s2 « 16) + s1;
        }
    }
}
```

8.20 GradientBrushApplicator.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

```

```
    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Buffers;
using System.Numerics;
using SixLabors.ImageSharp.Memory;
using SixLabors.ImageSharp.PixelFormats;
using SixLabors.ImageSharp.Drawing.Processing;
using SixLabors.ImageSharp;
using System.Reflection;

namespace VectSharp.Raster.ImageSharp
{
    // Adapted from https://github.com/SixLabors/ImageSharp.Drawing/blob/master/src/ImageSharp.Drawing/Processing/GradientBrush.cs - Why isn't this public? D:
    internal abstract class GradientBrushApplicator<TPixel> : BrushApplicator<TPixel>
             where TPixel : unmanaged, IPixel<TPixel>
    {
        private static readonly TPixel Transparent = Color.Transparent.ToPixel<TPixel>();

        private readonly ColorStop[] colorStops;

        private readonly GradientRepetitionMode repetitionMode;

        private readonly MemoryAllocator allocator;

        private readonly int scalineWidth;

        private readonly object blenderBuffers;

        private bool isDisposed;

        private PixelBlender<TPixel> _blender;

        IMemoryOwner<float> amountBuffer;
        IMemoryOwner<TPixel> overlayBuffer;
        static Type constructedLocalBlenderBuffersType;

        static GradientBrushApplicator()
        {
            Assembly utilities = Assembly.Load("SixLabors.ImageSharp.Drawing");
            Type genericThreadLocalBlenderBuffersType = utilities.GetType("SixLabors.ImageSharp.Drawing.Utilities.ThreadLocalBlenderBuffers`1", true);

            Type[] typeArgs = new Type[] { typeof(TPixel) };
            constructedLocalBlenderBuffersType = genericThreadLocalBlenderBuffersType.MakeGenericType(typeArgs);
        }


        /// <summary>
        /// Initializes a new instance of the <see cref="GradientBrushApplicator{TPixel}"/> class.
        /// </summary>
        /// <param name="configuration">The configuration instance to use when performing operations.</param>
        /// <param name="options">The graphics options.</param>
        /// <param name="target">The target image.</param>
        /// <param name="colorStops">An array of color stops sorted by their position.</param>
        /// <param name="repetitionMode">Defines if and how the gradient should be repeated.</param>
        protected GradientBrushApplicator(
            Configuration configuration,
            GraphicsOptions options,
            ImageFrame<TPixel> target,
            ColorStop[] colorStops,
            GradientRepetitionMode repetitionMode)
            : base(configuration, options, target)
        {
            // TODO: requires colorStops to be sorted by position.
            // Use Array.Sort with a custom comparer.
            this.colorStops = colorStops;
            this.repetitionMode = repetitionMode;
            this.scalineWidth = target.Width;
            this.allocator = configuration.MemoryAllocator;

            this.blenderBuffers = Activator.CreateInstance(constructedLocalBlenderBuffersType, new object[] { this.allocator, this.scalineWidth, false });

            this._blender = (PixelBlender<TPixel>)this.GetType().GetProperty("Blender", BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.GetProperty).GetValue(this);

            object data = this.blenderBuffers.GetType().GetField("data", BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.GetField).GetValue(this.blenderBuffers);
            object bufferOwner = data.GetType().GetProperty("Value", BindingFlags.Instance | BindingFlags.Public | BindingFlags.GetProperty).GetValue(data);
```

```
            amountBuffer = (IMemoryOwner<float>)bufferOwner.GetType().GetField("amountBuffer", BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.GetField).GetValue(bufferOwner);
            overlayBuffer = (IMemoryOwner<TPixel>)bufferOwner.GetType().GetField("overlayBuffer", BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.GetField).GetValue(bufferOwner);
        }

        internal TPixel this[int x, int y]
        {
            get
            {
                float positionOnCompleteGradient = this.PositionOnGradient(x + 0.5f, y + 0.5f);

                switch (this.repetitionMode)
                {
                    case GradientRepetitionMode.None:
                        // do nothing. The following could be done, but is not necessary:
                        // onLocalGradient = Math.Min(0, Math.Max(1, onLocalGradient));
                        break;
                    case GradientRepetitionMode.Repeat:
                        positionOnCompleteGradient %= 1;
                        break;
                    case GradientRepetitionMode.Reflect:
                        positionOnCompleteGradient %= 2;
                        if (positionOnCompleteGradient > 1)
                        {
                            positionOnCompleteGradient = 2 - positionOnCompleteGradient;
                        }

                        break;
                    case GradientRepetitionMode.DontFill:
                        if (positionOnCompleteGradient > 1 || positionOnCompleteGradient < 0)
                        {
                            return Transparent;
                        }

                        break;
                    default:
                        throw new ArgumentOutOfRangeException();
                }

                (ColorStop from, ColorStop to) = this.GetGradientSegment(positionOnCompleteGradient);

                if (from.Color.Equals(to.Color))
                {
                    return from.Color.ToPixel<TPixel>();
                }

                float onLocalGradient = (positionOnCompleteGradient - from.Ratio) / (to.Ratio - from.Ratio);

                return new Color(Vector4.Lerp((Vector4)from.Color, (Vector4)to.Color, onLocalGradient)).ToPixel<TPixel>();
            }
        }

        /// <inheritdoc />
        public override void Apply(Span<float> scanline, int x, int y)
        {
            Span<float> amounts = amountBuffer.Memory.Span.Slice(0, scanline.Length);
            Span<TPixel> overlays = overlayBuffer.Memory.Span.Slice(0, scanline.Length);

            float blendPercentage = this.Options.BlendPercentage;

            // TODO: Remove bounds checks.
            if (blendPercentage < 1)
            {
                for (int i = 0; i < scanline.Length; i++)
                {
                    amounts[i] = scanline[i] * blendPercentage;
                    overlays[i] = this[x + i, y];
                }
            }
            else
            {
                for (int i = 0; i < scanline.Length; i++)
                {
                    amounts[i] = scanline[i];
                    overlays[i] = this[x + i, y];
                }
            }

            Span<TPixel> destinationRow = this.Target.PixelBuffer.DangerousGetRowSpan(y).Slice(x, scanline.Length);

            this._blender.Blend(this.Configuration, destinationRow, destinationRow, overlays, amounts);
        }
```

```

        /// <summary>
        /// Calculates the position on the gradient for a given point.
        /// This method is abstract as it's content depends on the shape of the gradient.
        /// </summary>
        /// <param name="x">The x-coordinate of the point.</param>
        /// <param name="y">The y-coordinate of the point.</param>
        /// <returns>
        /// The position the given point has on the gradient.
        /// The position is not bound to the [0..1] interval.
        /// Values outside of that interval may be treated differently,
        /// e.g. for the <see cref="GradientRepetitionMode" /> enum.
        /// </returns>
        protected abstract float PositionOnGradient(float x, float y);

        /// <inheritdoc/>
        protected override void Dispose(bool disposing)
        {
            if (this.isDisposed)
            {
                return;
            }

            base.Dispose(disposing);

            if (disposing)
            {
                ((IDisposable)this.blenderBuffers).Dispose();
            }

            this.isDisposed = true;
        }

        private (ColorStop From, ColorStop To) GetGradientSegment(float positionOnCompleteGradient)
        {
            ColorStop localGradientFrom = this.colorStops[0];
            ColorStop localGradientTo = default;

            // TODO: ensure colorStops has at least 2 items (technically 1 would be okay, but that's
    no gradient)
            foreach (ColorStop colorStop in this.colorStops)
            {
                localGradientTo = colorStop;

                if (colorStop.Ratio > positionOnCompleteGradient)
                {
                    // we're done here, so break it!
                    break;
                }

                localGradientFrom = localGradientTo;
            }

            return (localGradientFrom, localGradientTo);
        }
    }
}
```

8.21 ImageSharpContext.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using SixLabors.ImageSharp;
using SixLabors.ImageSharp.Drawing;
using SixLabors.ImageSharp.Processing;
using SixLabors.ImageSharp.Drawing.Processing;
using System.Collections.Generic;
```

```
using SixLabors.ImageSharp.Advanced;
using System.IO;
using VectSharp.Filters;
using SixLabors.ImageSharp.Formats.Gif;
using System.Threading.Tasks;

namespace VectSharp.Raster.ImageSharp
{
    internal static class MatrixUtils
    {
        public static System.Numerics.Matrix3x2 ToMatrix(this double[,] matrix)
        {
            return new System.Numerics.Matrix3x2((float)matrix[0, 0], (float)matrix[1, 0], (float)matrix[0, 1], (float)matrix[1, 1], (float)matrix[0, 2], (float)matrix[1, 2]);
        }

        public static double[] Multiply(double[,] matrix, double[] vector)
        {
            double[] tbr = new double[2];

            tbr[0] = matrix[0, 0] * vector[0] + matrix[0, 1] * vector[1] + matrix[0, 2];
            tbr[1] = matrix[1, 0] * vector[0] + matrix[1, 1] * vector[1] + matrix[1, 2];

            return tbr;
        }

        public static Point Multiply(this double[,] matrix, Point vector)
        {
            return new Point(matrix[0, 0] * vector.X + matrix[0, 1] * vector.Y + matrix[0, 2], matrix[1, 0] * vector.X + matrix[1, 1] * vector.Y + matrix[1, 2]);
        }

        public static double[,] Multiply(double[,] matrix1, double[,] matrix2)
        {
            double[,] tbr = new double[3, 3];

            for (int i = 0; i < 3; i++)
            {
                for (int j = 0; j < 3; j++)
                {
                    for (int k = 0; k < 3; k++)
                    {
                        tbr[i, j] += matrix1[i, k] * matrix2[k, j];
                    }
                }
            }

            return tbr;
        }

        public static double[,] Rotate(double[,] matrix, double angle)
        {
            double[,] rotationMatrix = new double[3, 3];
            rotationMatrix[0, 0] = Math.Cos(angle);
            rotationMatrix[0, 1] = -Math.Sin(angle);
            rotationMatrix[1, 0] = Math.Sin(angle);
            rotationMatrix[1, 1] = Math.Cos(angle);
            rotationMatrix[2, 2] = 1;

            return Multiply(matrix, rotationMatrix);
        }

        public static double[,] Translate(double[,] matrix, double x, double y)
        {
            double[,] translationMatrix = new double[3, 3];
            translationMatrix[0, 0] = 1;
            translationMatrix[0, 2] = x;
            translationMatrix[1, 1] = 1;
            translationMatrix[1, 2] = y;
            translationMatrix[2, 2] = 1;

            return Multiply(matrix, translationMatrix);
        }

        public static double[,] Scale(double[,] matrix, double scaleX, double scaleY)
        {
            double[,] scaleMatrix = new double[3, 3];
            scaleMatrix[0, 0] = scaleX;
            scaleMatrix[1, 1] = scaleY;
            scaleMatrix[2, 2] = 1;

            return Multiply(matrix, scaleMatrix);
        }

        public static double[,] Identity = new double[,] { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };
        public static double[,] Invert(double[,] m)
        {
```

```
            double[,] tbr = new double[3, 3];

            tbr[0, 0] = (m[1, 1] * m[2, 2] - m[1, 2] * m[2, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[0, 1] = -(m[0, 1] * m[2, 2] - m[0, 2] * m[2, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[0, 2] = (m[0, 1] * m[1, 2] - m[0, 2] * m[1, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 0] = -(m[1, 0] * m[2, 2] - m[1, 2] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 1] = (m[0, 0] * m[2, 2] - m[0, 2] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 2] = -(m[0, 0] * m[1, 2] - m[0, 2] * m[1, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 0] = (m[1, 0] * m[2, 1] - m[1, 1] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 1] = -(m[0, 0] * m[2, 1] - m[0, 1] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 2] = (m[0, 0] * m[1, 1] - m[0, 1] * m[1, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);

            return tbr;
        }

        public static double Determinant(double[,] matrix)
        {
            return (matrix[0, 0] * matrix[1, 1] - matrix[1, 0] * matrix[0, 1]) * matrix[2, 2] - (matrix[0, 0] * matrix[1, 2] - matrix[1, 0] * matrix[0, 2]) * matrix[2, 1] + (matrix[0, 1] * matrix[1, 2] - matrix[1, 1] * matrix[0, 2]) * matrix[2, 0];
        }
    }

    internal class ImageSharpContext : IGraphicsContext
    {
        public Image<SixLabors.ImageSharp.PixelFormats.Rgba32> Image { get; }
        private Image Buffer { get; set; }
        private Image ClipBuffer { get; }

        public double Width { get; private set; }

        public double Height { get; private set; }

        public int IntWidth { get; private set; }
        public int IntHeight { get; private set; }

        public Font Font { get; set; }
        public TextBaselines TextBaseline { get; set; }

        public Brush FillStyle { get; set; }

        public Brush StrokeStyle { get; set; }

        public double LineWidth { get; set; }
        public LineCaps LineCap { get; set; }
        public LineJoins LineJoin { get; set; }

        public LineDash Dash { get; set; }
        public string Tag { get; set; }

        private double[,] _transform;
        private readonly Stack<double[,]> states;

        private readonly Stack<Image> clips;
        private Image CurrentClip;

        private List<IDisposable> disposables;

        PathBuilder currentPath;
        bool figureInitialised;
        Point currentPoint;
        double scaleFactor;

        public ImageSharpContext(double width, double height, double scaleFactor, Colour backgroundColour)
        {
            currentPath = new PathBuilder();
            figureInitialised = false;
            currentPoint = new Point();
```

```

            IntWidth = (int)(width * scaleFactor);
            IntHeight = (int)(height * scaleFactor);

            this.scaleFactor = Math.Sqrt(IntWidth / width * IntHeight / height);

            _transform = new double[3, 3];

            _transform[0, 0] = scaleFactor;
            _transform[1, 1] = scaleFactor;
            _transform[2, 2] = 1;

            states = new Stack<double[,]>();

            Width = width;
            Height = height;

            this.Image = new Image<SixLabors.ImageSharp.PixelFormats.Rgba32>(IntWidth, IntHeight);
            this.Image.Mutate(x => x.Fill(backgroundColour.ToImageSharpColor()));

            this.Buffer = new Image<SixLabors.ImageSharp.PixelFormats.Rgba32>(IntWidth, IntHeight);
            this.Buffer.Mutate(x => x.Clear(Color.FromRgba(0, 0, 0, 0)));

            this.ClipBuffer = new Image<SixLabors.ImageSharp.PixelFormats.Rgba32>(IntWidth, IntHeight);
            this.ClipBuffer.Mutate(x => x.Clear(Color.FromRgba(0, 0, 0, 0)));

            this.clips = new Stack<Image>();
            this.CurrentClip = null;

            disposables = new List<IDisposable>();
        }

        public void DisposeAllExceptImage()
        {
            for (int i = 0; i < disposables.Count; i++)
            {
                disposables[i].Dispose();
                this.CurrentClip?.Dispose();
                this.Buffer.Dispose();
                this.ClipBuffer.Dispose();

            }
        }

        public void Close()
        {
            this.currentPath.CloseFigure();
            this.figureInitialised = false;
        }

        public void CubicBezierTo(double p1X, double p1Y, double p2X, double p2Y, double p3X, double p3Y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref p1X, ref p1Y, ref p2X, ref p2Y, ref p3X, ref p3Y);

            if (figureInitialised)
            {
                Point p1 = new Point(p1X, p1Y);
                Point p2 = new Point(p2X, p2Y);
                Point p3 = new Point(p3X, p3Y);
                currentPath.AddCubicBezier(currentPoint.ToPointF(1), p1.ToPointF(1), p2.ToPointF(1), p3.ToPointF(1));
                currentPoint = p3;
            }
            else
            {
                currentPath.StartFigure();
                currentPoint = new Point(p3X, p3Y);
                figureInitialised = true;
            }
        }

        public void DrawRasterImage(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image)
        {
            Image sourceImage;

            unsafe
            {
                if (image.HasAlpha)
                {
                    ReadOnlySpan<byte> data = new ReadOnlySpan<byte>((void*)image.ImageDataAddress, image.Width * image.Height * 4);
                    sourceImage = SixLabors.ImageSharp.Image.LoadPixelData<SixLabors.ImageSharp.PixelFormats.Rgba32>(data, image.Width,
```

```
image.Height);
                }
                else
                {
                    ReadOnlySpan<byte> data = new ReadOnlySpan<byte>((void*)image.ImageDataAddress, image.Width * image.Height * 3);
                    sourceImage = SixLabors.ImageSharp.Image.LoadPixelData<SixLabors.ImageSharp.PixelFormats.Rgb24>(data, image.Width, image.Height);
                }
            }

            DrawImage(sourceX, sourceY, sourceWidth, sourceHeight, destinationX, destinationY, destinationWidth, destinationHeight, image.Interpolate, sourceImage);
        }

        internal void DrawImage(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, bool interpolate, Image sourceImage)
        {
            sourceImage.Mutate(x => x.Crop(new SixLabors.ImageSharp.Rectangle(sourceX, sourceY, sourceWidth, sourceHeight)));

            Point targetPoint = _transform.Multiply(new Point(destinationX, destinationY));
            Point targetPointX = _transform.Multiply(new Point(destinationX + destinationWidth, destinationY));
            Point targetPointY = _transform.Multiply(new Point(destinationX, destinationY + destinationHeight));
            Point targetPointXY = _transform.Multiply(new Point(destinationX + destinationWidth, destinationY + destinationHeight));

            double minX = Math.Min(Math.Min(targetPoint.X, targetPointX.X), Math.Min(targetPointY.X, targetPointXY.X));
            double minY = Math.Min(Math.Min(targetPoint.Y, targetPointX.Y), Math.Min(targetPointY.Y, targetPointXY.Y));

            double maxX = Math.Max(Math.Max(targetPoint.X, targetPointX.X), Math.Max(targetPointY.X, targetPointXY.X));
            double maxY = Math.Max(Math.Max(targetPoint.Y, targetPointX.Y), Math.Max(targetPointY.Y, targetPointXY.Y));

            double[,] currTransform = MatrixUtils.Multiply(MatrixUtils.Translate(_transform, destinationX, destinationY), MatrixUtils.Scale(MatrixUtils.Identity, destinationWidth / sourceImage.Width, destinationHeight / sourceImage.Height));

            Point origin = currTransform.Multiply(new Point(0, 0));

            double[,] translation = MatrixUtils.Translate(MatrixUtils.Identity, -minX, -minY);

            double[,] centeredTransform = MatrixUtils.Multiply(translation, currTransform);

            SixLabors.ImageSharp.Processing.Processors.Transforms.IResampler resampler;

            if (interpolate)
            {
                resampler = KnownResamplers.Bicubic;
            }
            else
            {
                resampler = KnownResamplers.NearestNeighbor;
            }

            sourceImage.Mutate(x => x.Transform(new SixLabors.ImageSharp.Rectangle(sourceX, sourceY, sourceWidth, sourceHeight), centeredTransform.ToMatrix(), new SixLabors.ImageSharp.Size((int)Math.Round(maxX - minX), (int)Math.Round(maxY - minY)), resampler));

            if (this.CurrentClip == null)
            {
                this.Image.Mutate(x => x.CheckedDrawImage(sourceImage, new SixLabors.ImageSharp.Point((int)Math.Round(minX), (int)Math.Round(minY)), 1));
            }
            else
            {
                this.ClipBuffer.Mutate(x => x.Clear(Color.FromRgba(0, 0, 0, 0)));
                this.ClipBuffer.Mutate(x => x.CheckedDrawImage(sourceImage, new SixLabors.ImageSharp.Point((int)Math.Round(minX), (int)Math.Round(minY)), 1));

                GraphicsOptions opt = new GraphicsOptions() { AlphaCompositionMode = SixLabors.ImageSharp.PixelFormats.PixelAlphaCompositionMode.SrcIn };

                this.Buffer.Dispose();

                this.Buffer = this.CurrentClip.Clone(x => x.CheckedDrawImage(this.ClipBuffer, opt));

                this.Image.Mutate(x => x.CheckedDrawImage(this.Buffer, 1));
            }
        }
```

```

        public void Fill()
        {
            IPath path = this.currentPath.Build();

            if (this.CurrentClip == null)
            {
                this.Image.Mutate(x => x.Fill(new DrawingOptions() { Transform = _transform.ToMatrix() }, this.FillStyle.ToImageSharpBrush(_transform), path));
            }
            else
            {
                this.ClipBuffer.Mutate(x => x.Clear(Color.FromRgba(0, 0, 0, 0)));
                this.ClipBuffer.Mutate(x => x.Fill(new DrawingOptions() { Transform = _transform.ToMatrix() }, this.FillStyle.ToImageSharpBrush(_transform), path));

                GraphicsOptions opt = new GraphicsOptions() { AlphaCompositionMode = SixLabors.ImageSharp.PixelFormats.PixelAlphaCompositionMode.SrcIn };

                this.Buffer.Dispose();

                this.Buffer = this.CurrentClip.Clone(x => x.CheckedDrawImage(this.ClipBuffer, opt));

                this.Image.Mutate(x => x.CheckedDrawImage(this.Buffer, 1));
            }

            this.currentPath = new PathBuilder();
            this.figureInitialised = false;
        }

        public void FillText(string text, double x, double y)
        {
            PathText(text, x, y);
            Fill();
        }

        private void PathText(string text, double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            GraphicsPath textPath = new GraphicsPath().AddText(x, y, text, Font, TextBaseline);

            for (int j = 0; j < textPath.Segments.Count; j++)
            {
                switch (textPath.Segments[j].Type)
                {
                    case VectSharp.SegmentType.Move:
                        this.MoveTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.Line:
                        this.LineTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.CubicBezier:
                        this.CubicBezierTo(textPath.Segments[j].Points[0].X, textPath.Segments[j].Points[0].Y, textPath.Segments[j].Points[1].X, textPath.Segments[j].Points[1].Y, textPath.Segments[j].Points[2].X, textPath.Segments[j].Points[2].Y);
                        break;
                    case VectSharp.SegmentType.Close:
                        this.Close();
                        break;
                }
            }
        }

        public void LineTo(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            if (figureInitialised)
            {
                Point newPoint = new Point(x, y);
                currentPath.AddLine(currentPoint.ToPointF(1), newPoint.ToPointF(1));
                currentPoint = newPoint;
            }
            else
            {
                currentPath.StartFigure();
                currentPoint = new Point(x, y);
                figureInitialised = true;
            }
        }

        public void MoveTo(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            currentPath.StartFigure();
```

```
            currentPoint = new Point(x, y);
            figureInitialised = true;
        }

        public void Rectangle(double x0, double y0, double width, double height)
        {
            MoveTo(x0, y0);
            LineTo(x0 + width, y0);
            LineTo(x0 + width, y0 + height);
            LineTo(x0, y0 + height);
            Close();
        }

        public void Restore()
        {
            _transform = states.Pop();

            Image newClip = clips.Pop();
            if (CurrentClip != newClip)
            {
                CurrentClip.Dispose();
            }

            CurrentClip = newClip;
        }

        public void Rotate(double angle)
        {
            Utils.CoerceNaNAndInfinityToZero(ref angle);

            _transform = MatrixUtils.Rotate(_transform, angle);

            currentPath = new PathBuilder();
            figureInitialised = false;
        }

        public void Save()
        {
            states.Push((double[,])_transform.Clone());
            clips.Push(CurrentClip);
        }

        public void Scale(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            _transform = MatrixUtils.Scale(_transform, x, y);

            currentPath = new PathBuilder();
            figureInitialised = false;
        }

        public void SetClippingPath()
        {
            IPath path = this.currentPath.Build();

            Image newClip;

            DrawingOptions opt = new DrawingOptions() { Transform = _transform.ToMatrix() };

            if (this.CurrentClip == null)
            {
                newClip = new Image<SixLabors.ImageSharp.PixelFormats.Rgba32>(IntWidth, IntHeight);
                newClip.Mutate(x => x.Clear(Color.FromRgba(0, 0, 0, 0)));
                newClip.Mutate(x => x.Fill(opt, Color.FromRgb(0, 0, 0), path));
            }
            else
            {
                this.Buffer.Mutate(x => x.Clear(Color.FromRgba(0, 0, 0, 0)));
                this.Buffer.Mutate(x => x.Fill(opt, Color.FromRgb(0, 0, 0), path));

                newClip = this.CurrentClip.Clone(x => x.CheckedDrawImage(this.Buffer, new
    GraphicsOptions() { AlphaCompositionMode =
    SixLabors.ImageSharp.PixelFormats.PixelAlphaCompositionMode.SrcIn }));

                disposables.Add(this.CurrentClip);
            }

            this.CurrentClip = newClip;

            this.currentPath = new PathBuilder();
            this.figureInitialised = false;
        }

        public void SetFillStyle((int r, int g, int b, double a) style)
        {
            this.FillStyle = Colour.FromRgba(style);
```

```
        }

        public void SetFillStyle(Brush style)
        {
            this.FillStyle = style;
        }

        public void SetLineDash(LineDash dash)
        {
            this.Dash = dash;
        }

        public void SetStrokeStyle((int r, int g, int b, double a) style)
        {
            this.StrokeStyle = Colour.FromRgba(style);
        }

        public void SetStrokeStyle(Brush style)
        {
            this.StrokeStyle = style;
        }

        public void Stroke()
        {
            if (this.LineWidth > 0)
            {
                if (this.CurrentClip == null)
                {
                    this.Image.Mutate(x => x.Fill(new DrawingOptions() { Transform = _transform.ToMatrix() }, this.StrokeStyle.ToImageSharpBrush(_transform), this.currentPath.Build().GenerateOutline((float)(this.LineWidth), this.Dash.ToImageSharpDash(this.LineWidth), false, this.LineJoin.ToImageSharpJoint(), this.LineCap.ToImageSharpCap())));
                }
                else
                {
                    this.ClipBuffer.Mutate(x => x.Clear(Color.FromRgba(0, 0, 0, 0)));
                    this.ClipBuffer.Mutate(x => x.Fill(new DrawingOptions() { Transform = _transform.ToMatrix() }, this.StrokeStyle.ToImageSharpBrush(_transform), this.currentPath.Build().GenerateOutline((float)(this.LineWidth), this.Dash.ToImageSharpDash(this.LineWidth), false, this.LineJoin.ToImageSharpJoint(), this.LineCap.ToImageSharpCap())));

                    GraphicsOptions opt = new GraphicsOptions() { AlphaCompositionMode = SixLabors.ImageSharp.PixelFormats.PixelAlphaCompositionMode.SrcIn };

                    this.Buffer.Dispose();

                    this.Buffer = this.CurrentClip.Clone(x => x.CheckedDrawImage(this.ClipBuffer, opt));

                    this.Image.Mutate(x => x.CheckedDrawImage(this.Buffer, 1));
                }
            }

            this.currentPath = new PathBuilder();
            this.figureInitialised = false;
        }

        public void StrokeText(string text, double x, double y)
        {
            PathText(text, x, y);
            Stroke();
        }

        public void Transform(double a, double b, double c, double d, double e, double f)
        {
            Utils.CoerceNaNAndInfinityToZero(ref a, ref b, ref c, ref d, ref e, ref f);

            double[,] transfMatrix = new double[3, 3] { { a, c, e }, { b, d, f }, { 0, 0, 1 } };
            _transform = MatrixUtils.Multiply(_transform, transfMatrix);

            currentPath = new PathBuilder();
            figureInitialised = false;
        }

        public void Translate(double x, double y)
        {
            Utils.CoerceNaNAndInfinityToZero(ref x, ref y);

            _transform = MatrixUtils.Translate(_transform, x, y);

            currentPath = new PathBuilder();
            figureInitialised = false;
        }

        public void DrawFilteredGraphics(Graphics graphics, IFilter filter)
```

```
        {
            double scale = this.scaleFactor * Math.Sqrt(MatrixUtils.Determinant(_transform));

            Rectangle bounds = graphics.GetBounds();

            bounds = new Rectangle(bounds.Location.X - filter.TopLeftMargin.X, bounds.Location.Y - filter.TopLeftMargin.Y, bounds.Size.Width + filter.TopLeftMargin.X + filter.BottomRightMargin.X, bounds.Size.Height + filter.TopLeftMargin.Y + filter.BottomRightMargin.Y);

            if (bounds.Size.Width > 0 && bounds.Size.Height > 0)
            {
                bool rasterisationNeeded = true;

                if (filter is GaussianBlurFilter gauss)
                {
                    rasterisationNeeded = false;

                    Page pag = new Page(1, 1);
                    pag.Graphics.DrawGraphics(0, 0, graphics);
                    pag.Crop(bounds.Location, bounds.Size);

                    Image<SixLabors.ImageSharp.PixelFormats.Rgba32> img = ImageSharpContextInterpreter.SaveAsImage(pag, scale);
                    img.Mutate(x => x.GaussianBlur((float)(gauss.StandardDeviation * scale)));

                    DrawImage(0, 0, img.Width, img.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, true, img);

                    img.Dispose();
                }
                else if (filter is ColourMatrixFilter cmf)
                {
                    rasterisationNeeded = false;

                    Page pag = new Page(1, 1);
                    pag.Graphics.DrawGraphics(0, 0, graphics);
                    pag.Crop(bounds.Location, bounds.Size);

                    Image<SixLabors.ImageSharp.PixelFormats.Rgba32> img = ImageSharpContextInterpreter.SaveAsImage(pag, scale);

                    img.Mutate(x => x.Filter(new ColorMatrix((float)cmf.ColourMatrix.R1, (float)cmf.ColourMatrix.G1, (float)cmf.ColourMatrix.B1, (float)cmf.ColourMatrix.A1, (float)cmf.ColourMatrix.R2, (float)cmf.ColourMatrix.G2, (float)cmf.ColourMatrix.B2, (float)cmf.ColourMatrix.A2, (float)cmf.ColourMatrix.R3, (float)cmf.ColourMatrix.G3, (float)cmf.ColourMatrix.B3, (float)cmf.ColourMatrix.A3, (float)cmf.ColourMatrix.R4, (float)cmf.ColourMatrix.G4, (float)cmf.ColourMatrix.B4, (float)cmf.ColourMatrix.A4, (float)cmf.ColourMatrix.R5, (float)cmf.ColourMatrix.G5, (float)cmf.ColourMatrix.B5, (float)cmf.ColourMatrix.A5)));

                    DrawImage(0, 0, img.Width, img.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, true, img);

                    img.Dispose();
                }
                else if (filter is BoxBlurFilter bbf)
                {
                    rasterisationNeeded = false;

                    Page pag = new Page(1, 1);
                    pag.Graphics.DrawGraphics(0, 0, graphics);
                    pag.Crop(bounds.Location, bounds.Size);

                    Image<SixLabors.ImageSharp.PixelFormats.Rgba32> img = ImageSharpContextInterpreter.SaveAsImage(pag, scale);

                    img.Mutate(x => x.BoxBlur((int)Math.Round(bbf.BoxRadius * scale)));

                    DrawImage(0, 0, img.Width, img.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, true, img);

                    img.Dispose();
                }
                else if (filter is CompositeLocationInvariantFilter comp)
                {
                    bool allSupported = true;

                    foreach (IFilter filter2 in comp.Filters)
                    {
                        if (!(filter2 is GaussianBlurFilter) && !(filter2 is ColourMatrixFilter) && !(filter2 is BoxBlurFilter))
                        {
                            allSupported = false;
                            break;
                        }
                    }

```

```
                    if (allSupported)
                    {
                        rasterisationNeeded = false;

                        Page pag = new Page(1, 1);
                        pag.Graphics.DrawGraphics(0, 0, graphics);
                        pag.Crop(bounds.Location, bounds.Size);

                        Image<SixLabors.ImageSharp.PixelFormats.Rgba32> img = ImageSharpContextInterpreter.SaveAsImage(pag, scale);

                        foreach (IFilter filter2 in comp.Filters)
                        {
                            if (filter2 is GaussianBlurFilter gauss2)
                            {
                                img.Mutate(x => x.GaussianBlur((float)(gauss2.StandardDeviation * scale)));
                            }
                            else if (filter2 is ColourMatrixFilter cmf2)
                            {
                                img.Mutate(x => x.Filter(new ColorMatrix((float)cmf2.ColourMatrix.R1, (float)cmf2.ColourMatrix.G1, (float)cmf2.ColourMatrix.B1, (float)cmf2.ColourMatrix.A1, (float)cmf2.ColourMatrix.R2, (float)cmf2.ColourMatrix.G2, (float)cmf2.ColourMatrix.B2, (float)cmf2.ColourMatrix.A2, (float)cmf2.ColourMatrix.R3, (float)cmf2.ColourMatrix.G3, (float)cmf2.ColourMatrix.B3, (float)cmf2.ColourMatrix.A3, (float)cmf2.ColourMatrix.R4, (float)cmf2.ColourMatrix.G4, (float)cmf2.ColourMatrix.B4, (float)cmf2.ColourMatrix.A4, (float)cmf2.ColourMatrix.R5, (float)cmf2.ColourMatrix.G5, (float)cmf2.ColourMatrix.B5, (float)cmf2.ColourMatrix.A5)));
                            }
                            else if (filter2 is BoxBlurFilter bbf2)
                            {
                                img.Mutate(x => x.BoxBlur((int)Math.Round(bbf2.BoxRadius * scale)));
                            }
                        }


                        DrawImage(0, 0, img.Width, img.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, true, img);

                        img.Dispose();
                    }
                }

                if (rasterisationNeeded)
                {
                    RasterImage rasterised = ImageSharpContextInterpreter.Rasterise(graphics, bounds, scale, true);
                    RasterImage filtered = null;

                    if (filter is IFilterWithRasterisableParameter filterWithRastParam)
                    {
                        filterWithRastParam.RasteriseParameter(ImageSharpContextInterpreter.Rasterise, scale);
                    }

                    if (filter is ILocationInvariantFilter locInvFilter)
                    {
                        filtered = locInvFilter.Filter(rasterised, scale);
                    }
                    else if (filter is IFilterWithLocation filterWithLoc)
                    {
                        filtered = filterWithLoc.Filter(rasterised, bounds, scale);
                    }

                    if (filtered != null)
                    {
                        rasterised.Dispose();

                        DrawRasterImage(0, 0, filtered.Width, filtered.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, filtered);
                    }
                }
            }
        }
    }

    internal static class Utils
    {
        public static IImageProcessingContext CheckedDrawImage(this IImageProcessingContext x, Image image, SixLabors.ImageSharp.Point point, float opacity)
        {
            int x0A = 0;
            int y0A = 0;
            SixLabors.ImageSharp.Size size = x.GetCurrentSize();
            int x1A = size.Width;
            int y1A = size.Height;

```

```
            int x0B = point.X;
            int y0B = point.Y;
            int x1B = point.X + image.Width;
            int y1B = point.Y + image.Height;

            if (!(x0A >= x1B || x1A <= x0B || y0A >= y1B || y1A <= y0B))
            {
                return x.DrawImage(image, point, opacity);
            }
            else
            {
                return x;
            }
        }

        public static IImageProcessingContext CheckedDrawImage(this IImageProcessingContext x, Image image, float opacity)
        {
            return x.DrawImage(image, opacity);
        }

        public static IImageProcessingContext CheckedDrawImage(this IImageProcessingContext x, Image image, GraphicsOptions options)
        {
            return x.DrawImage(image, options);
        }

        public static PointF ToPointF(this Point pt, double scaleFactor)
        {
            return new PointF((float)(pt.X * scaleFactor), (float)(pt.Y * scaleFactor));
        }

        public static void CoerceNaNAndInfinityToZero(ref double val)
        {
            if (double.IsNaN(val) || double.IsInfinity(val) || val == double.MinValue || val == double.MaxValue)
            {
                val = 0;
            }
        }

        public static void CoerceNaNAndInfinityToZero(ref double val1, ref double val2)
        {
            if (double.IsNaN(val1) || double.IsInfinity(val1) || val1 == double.MinValue || val1 == double.MaxValue)
            {
                val1 = 0;
            }

            if (double.IsNaN(val2) || double.IsInfinity(val2) || val2 == double.MinValue || val2 == double.MaxValue)
            {
                val2 = 0;
            }
        }

        public static void CoerceNaNAndInfinityToZero(ref double val1, ref double val2, ref double val3, ref double val4)
        {
            if (double.IsNaN(val1) || double.IsInfinity(val1) || val1 == double.MinValue || val1 == double.MaxValue)
            {
                val1 = 0;
            }

            if (double.IsNaN(val2) || double.IsInfinity(val2) || val2 == double.MinValue || val2 == double.MaxValue)
            {
                val2 = 0;
            }

            if (double.IsNaN(val3) || double.IsInfinity(val3) || val3 == double.MinValue || val3 == double.MaxValue)
            {
                val3 = 0;
            }

            if (double.IsNaN(val4) || double.IsInfinity(val4) || val4 == double.MinValue || val4 == double.MaxValue)
            {
                val4 = 0;
            }
        }

        public static void CoerceNaNAndInfinityToZero(ref double val1, ref double val2, ref double val3, ref double val4, ref double val5, ref double val6)
        {
```

```
            if (double.IsNaN(val1) || double.IsInfinity(val1) || val1 == double.MinValue || val1 == double.MaxValue)
            {
                val1 = 0;
            }

            if (double.IsNaN(val2) || double.IsInfinity(val2) || val2 == double.MinValue || val2 == double.MaxValue)
            {
                val2 = 0;
            }

            if (double.IsNaN(val3) || double.IsInfinity(val3) || val3 == double.MinValue || val3 == double.MaxValue)
            {
                val3 = 0;
            }

            if (double.IsNaN(val4) || double.IsInfinity(val4) || val4 == double.MinValue || val4 == double.MaxValue)
            {
                val4 = 0;
            }

            if (double.IsNaN(val5) || double.IsInfinity(val5) || val5 == double.MinValue || val5 == double.MaxValue)
            {
                val5 = 0;
            }

            if (double.IsNaN(val6) || double.IsInfinity(val6) || val6 == double.MinValue || val6 == double.MaxValue)
            {
                val6 = 0;
            }
        }

        public static Color ToImageSharpColor(this Colour colour)
        {
            return Color.FromRgba((byte)(colour.R * 255), (byte)(colour.G * 255), (byte)(colour.B * 255), (byte)(colour.A * 255));
        }

        public static IBrush ToImageSharpBrush(this Brush brush, double[,] transform)
        {
            if (brush is SolidColourBrush solid)
            {
                return new SixLabors.ImageSharp.Drawing.Processing.SolidBrush(solid.Colour.ToImageSharpColor());
            }
            else if (brush is LinearGradientBrush linear)
            {
                ColorStop[] colorStops = new ColorStop[linear.GradientStops.Count];

                for (int i = 0; i < linear.GradientStops.Count; i++)
                {
                    colorStops[i] = new ColorStop((float)linear.GradientStops[i].Offset, linear.GradientStops[i].Colour.ToImageSharpColor());
                }

                return new SixLabors.ImageSharp.Drawing.Processing.LinearGradientBrush(transform.Multiply(linear.StartPoint).ToPointF(1), transform.Multiply(linear.EndPoint).ToPointF(1), GradientRepetitionMode.None, colorStops);
            }
            else if (brush is RadialGradientBrush radial)
            {
                ColorStop[] colorStops = new ColorStop[radial.GradientStops.Count];

                for (int i = 0; i < radial.GradientStops.Count; i++)
                {
                    colorStops[i] = new ColorStop((float)radial.GradientStops[i].Offset, radial.GradientStops[i].Colour.ToImageSharpColor());
                }

                return new RadialGradientBrushSVGStyle(radial.Centre, radial.FocalPoint, radial.Radius, transform, GradientRepetitionMode.None, colorStops);
            }
            else
            {
                throw new NotImplementedException();
            }
        }

        public static float[] ToImageSharpDash(this LineDash dash, double lineThickness)
        {
            if (dash.UnitsOn > 0 || dash.UnitsOff > 0)
            {
```

```
                return new float[]
                {
                    (float)(dash.UnitsOn / lineThickness),
                    (float)(dash.UnitsOff / lineThickness)
                };
            }
            else
            {
                return new float[] { };
            }
        }

        public static JointStyle ToImageSharpJoint(this LineJoins join)
        {
            switch (join)
            {
                case LineJoins.Round:
                    return JointStyle.Round;
                case LineJoins.Miter:
                    return JointStyle.Miter;
                case LineJoins.Bevel:
                    return JointStyle.Square;
                default:
                    return JointStyle.Miter;
            }
        }

        public static EndCapStyle ToImageSharpCap(this LineCaps cap)
        {
            switch (cap)
            {
                case LineCaps.Square:
                    return EndCapStyle.Square;
                case LineCaps.Round:
                    return EndCapStyle.Round;
                case LineCaps.Butt:
                    return EndCapStyle.Butt;
                default:
                    throw new NotImplementedException();
            }
        }

        internal class RadialGradientBrushSVGStyle : SixLabors.ImageSharp.Drawing.Processing.GradientBrush
        {
            private readonly Point Centre;
            private readonly Point FocalPoint;
            private readonly double Radius;
            private readonly double[,] InverseTransform;

            public RadialGradientBrushSVGStyle(Point centre, Point focalPoint, double radius, double[,] transform, GradientRepetitionMode repetitionMode, params ColorStop[] colorStops) : base(repetitionMode, colorStops)
            {
                Centre = centre;
                FocalPoint = focalPoint;
                Radius = radius;
                InverseTransform = MatrixUtils.Invert(transform);
            }

            public override BrushApplicator<TPixel> CreateApplicator<TPixel>(Configuration configuration, GraphicsOptions options, ImageFrame<TPixel> source, RectangleF region)
            {
                return new RadialGradientBrushPDFStyleApplicator<TPixel>(configuration, options, source, Centre, FocalPoint, Radius, InverseTransform, ColorStops, RepetitionMode);
            }

            private class RadialGradientBrushPDFStyleApplicator<TPixel> : GradientBrushApplicator<TPixel> where TPixel : unmanaged, SixLabors.ImageSharp.PixelFormats.IPixel<TPixel>
            {
                private readonly Point Centre;
                private readonly Point FocalPoint;
                private readonly double Radius;
                private readonly double[,] InverseTransform;

                private readonly double a;

                public RadialGradientBrushPDFStyleApplicator(
                    Configuration configuration,
                    GraphicsOptions options,
                    ImageFrame<TPixel> target,
                    Point centre, Point focalPoint, double radius, double[,] inverseTransform,
                    ColorStop[] colorStops,
                    GradientRepetitionMode repetitionMode)
                : base(configuration, options, target, colorStops, repetitionMode)
                {
```

```
                    this.Centre = centre;
                    this.FocalPoint = focalPoint;
                    this.Radius = radius;
                    this.InverseTransform = inverseTransform;

                    a = (centre.X - focalPoint.X) * (centre.X - focalPoint.X) + (centre.Y - focalPoint.Y) * (centre.Y - focalPoint.Y) - radius * radius;
                }

                Random rnd = new Random();

                protected override float PositionOnGradient(float x, float y)
                {
                    Point realPoint = InverseTransform.Multiply(new Point(x, y));

                    double c = (realPoint.X - FocalPoint.X) * (realPoint.X - FocalPoint.X) + (realPoint.Y - FocalPoint.Y) * (realPoint.Y - FocalPoint.Y);

                    double halfB = -((realPoint.X - FocalPoint.X) * (Centre.X - FocalPoint.X) + (realPoint.Y - FocalPoint.Y) * (Centre.Y - FocalPoint.Y));

                    double sqrt = Math.Sqrt(halfB * halfB - a * c);

                    double tbr1 = (-halfB + sqrt) / a;
                    double tbr2 = (-halfB - sqrt) / a;

                    if (tbr1 >= 0 && tbr2 < 0)
                    {
                        return (float)tbr1;
                    }
                    else if (tbr1 < 0 && tbr2 >= 0)
                    {
                        return (float)tbr2;
                    }
                    else if (tbr1 < 0 && tbr2 < 0)
                    {
                        return 0;
                    }
                    else
                    {
                        return (float)Math.Min(tbr1, tbr2);
                    }
                }

                /// <inheritdoc/>
                public override void Apply(Span<float> scanline, int x, int y)
                {
                    base.Apply(scanline, x, y);
                }
            }
        }
    }

    /// <summary>
    /// Enumeration containing the supported output formats.
    /// </summary>
    public enum OutputFormats
    {
        /// <summary>
        /// Windows bitmap format
        /// </summary>
        BMP,

        /// <summary>
        /// Graphics interchange format
        /// </summary>
        GIF,

        /// <summary>
        /// Joint photographic experts group format
        /// </summary>
        JPEG,


        /// <summary>
        /// Portable bitmap format
        /// </summary>
        PBM,

        /// <summary>
        /// Portable network graphics format
        /// </summary>
        PNG,

        /// <summary>
        /// Truevision graphics adapter format
        /// </summary>
```

```
        TGA,

        /// <summary>
        /// Tag image file format
        /// </summary>
        TIFF,

        /// <summary>
        /// WebP format
        /// </summary>
        WebP
    }

    /// <summary>
    /// Contains methods to render a <see cref="Page"/> to an <see cref="Image"/>.
    /// </summary>
    public static class ImageSharpContextInterpreter
    {
        /// <summary>
        /// Render the page to an <see cref="Image"/> object.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="scale">The scale to be used when rasterising the page. This will determine the width and height of the <see cref="Image"/>.</param>
        /// <returns>An <see cref="Image"/> containing the rasterised page.</returns>
        public static Image<SixLabors.ImageSharp.PixelFormats.Rgba32> SaveAsImage(this Page page, double scale = 1)
        {
            ImageSharpContext ctx = new ImageSharpContext(page.Width, page.Height, scale, page.Background);
            page.Graphics.CopyToIGraphicsContext(ctx);
            ctx.DisposeAllExceptImage();
            return ctx.Image;
        }


        /// <summary>
        /// Render the page to an image stream.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="imageStream">The <see cref="Stream"/> on which the image data will be written.</param>
        /// <param name="outputFormat">The format of the image that will be created.</param>
        /// <param name="scale">The scale to be used when rasterising the page. This will determine the width and height of the image.</param>
        public static void SaveAsImage(this Page page, Stream imageStream, OutputFormats outputFormat, double scale = 1)
        {
            Image image = SaveAsImage(page, scale);

            switch (outputFormat)
            {
                case OutputFormats.BMP:
                    image.SaveAsBmp(imageStream);
                    break;
                case OutputFormats.GIF:
                    image.SaveAsGif(imageStream);
                    break;
                case OutputFormats.JPEG:
                    image.SaveAsJpeg(imageStream);
                    break;
                case OutputFormats.PBM:
                    image.SaveAsPbm(imageStream);
                    break;
                case OutputFormats.PNG:
                    image.SaveAsPng(imageStream);
                    break;
                case OutputFormats.TGA:
                    image.SaveAsTga(imageStream);
                    break;
                case OutputFormats.TIFF:
                    image.SaveAsTiff(imageStream);
                    break;
                case OutputFormats.WebP:
                    image.SaveAsWebp(imageStream);
                    break;
            }

            image.Dispose();
        }

        /// <summary>
        /// The exception that is raised when the output file format is not specified and the file name does not have an extension corresponding to a known file format.
        /// </summary>
        public class UnknownFormatException : Exception
        {
```

```
            /// <summary>
            /// The extension of the file that does not correspond to any known file format.
            /// </summary>
            public string Format { get; }

            internal UnknownFormatException(string format) : base("The extension " + format + " does not correspond to any known file format!")
            {
                this.Format = format;
            }
        }

        /// <summary>
        /// Render the page to an image file.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="fileName">The path of the file where the image will be saved.</param>
        /// <param name="outputFormat">The format of the image that will be created. If this is <see langword="null" /> (the default), the format is desumed from the extension of the file.</param>
        /// <param name="scale">The scale to be used when rasterising the page. This will determine the width and height of the image.</param>
        public static void SaveAsImage(this Page page, string fileName, OutputFormats? outputFormat = null, double scale = 1)
        {
            OutputFormats actualOutputFormat;

            if (outputFormat.HasValue)
            {
                actualOutputFormat = outputFormat.Value;
            }
            else
            {
                string extension = System.IO.Path.GetExtension(fileName).ToLower();

                switch (extension)
                {
                    case ".bmp":
                    case ".dib":
                        actualOutputFormat = OutputFormats.BMP;
                        break;

                    case ".gif":
                        actualOutputFormat = OutputFormats.GIF;
                        break;

                    case ".jpeg":
                    case ".jpg":
                        actualOutputFormat = OutputFormats.JPEG;
                        break;

                    case ".pbm":
                        actualOutputFormat = OutputFormats.PBM;
                        break;

                    case ".png":
                        actualOutputFormat = OutputFormats.PNG;
                        break;

                    case ".tga":
                    case ".targa":
                        actualOutputFormat = OutputFormats.TGA;
                        break;

                    case ".tif":
                    case ".tiff":
                        actualOutputFormat = OutputFormats.TIFF;
                        break;

                    case ".webp":
                        actualOutputFormat = OutputFormats.WebP;
                        break;

                    default:
                        throw new UnknownFormatException(extension);
                }
            }

            using (FileStream imageStream = new FileStream(fileName, FileMode.Create))
            {
                SaveAsImage(page, imageStream, actualOutputFormat, scale);
            }
        }

        /// <summary>
        /// Render the page to raw pixel data, in 32bpp RGBA format.
        /// </summary>
        /// <param name="pag">The <see cref="Page"/> to render.</param>
```

```
        /// <param name="scale">The scale to be used when rasterising the page. This will determine the width and height of the image.</param>
        /// <param name="width">The width of the rendered image.</param>
        /// <param name="height">The height of the rendered image.</param>
        /// <param name="totalSize">The size in bytes of the raw pixel data.</param>
        /// <returns>A <see cref="DisposableIntPtr"/> containing a pointer to the raw pixel data, stored in unmanaged memory. Dispose this object to release the unmanaged memory.</returns>
        public static DisposableIntPtr SaveAsRawBytes(this Page pag, out int width, out int height, out int totalSize, double scale = 1)
        {
            Image<SixLabors.ImageSharp.PixelFormats.Rgba32> img = SaveAsImage(pag, scale);

            int stride = img.Width * 4;
            int size = stride * img.Height;

            IntPtr tbr = System.Runtime.InteropServices.Marshal.AllocHGlobal(size);
            GC.AddMemoryPressure(size);

            IntPtr pointer = tbr;

            unsafe
            {
                for (int y = 0; y < img.Height; y++)
                {
                    Memory<SixLabors.ImageSharp.PixelFormats.Rgba32> row = img.DangerousGetPixelRowMemory(y);

                    Span<SixLabors.ImageSharp.PixelFormats.Rgba32> newRow = new Span<SixLabors.ImageSharp.PixelFormats.Rgba32>(pointer.ToPointer(), row.Length);
                    row.Span.CopyTo(newRow);

                    pointer = IntPtr.Add(pointer, stride);
                }
            }

            width = img.Width;
            height = img.Height;
            totalSize = size;

            img.Dispose();

            return new DisposableIntPtr(tbr);
        }


        /// <summary>
        /// Return the page to raw pixel data, in 32bpp RGBA format.
        /// </summary>
        /// <param name="pag">The <see cref="Page"/> to render.</param>
        /// <param name="scale">The scale to be used when rasterising the page. This will determine the width and height of the image.</param>
        /// <param name="width">The width of the rendered image.</param>
        /// <param name="height">The height of the rendered image.</param>
        /// <returns>A byte array containing the raw pixel data.</returns>
        public static byte[] SaveAsRawBytes(this Page pag, out int width, out int height, double scale = 1)
        {
            Image<SixLabors.ImageSharp.PixelFormats.Rgba32> img = SaveAsImage(pag, scale);

            int stride = img.Width * 4;
            int size = stride * img.Height;

            byte[] tbr = new byte[size];

            unsafe
            {
                for (int y = 0; y < img.Height; y++)
                {
                    Memory<SixLabors.ImageSharp.PixelFormats.Rgba32> row = img.DangerousGetPixelRowMemory(y);

                    Span<byte> bytes = System.Runtime.InteropServices.MemoryMarshal.Cast<SixLabors.ImageSharp.PixelFormats.Rgba32, byte>(row.Span);
                    Span<byte> newRow = new Span<byte>(tbr, y * stride, stride);
                    bytes.CopyTo(newRow);
                }
            }

            width = img.Width;
            height = img.Height;

            img.Dispose();

            return tbr;
        }

```

```
        /// <summary>
        /// Rasterise a region of a <see cref="Graphics"/> object.
        /// </summary>
        /// <param name="graphics">The <see cref="Graphics"/> object that will be rasterised.</param>
        /// <param name="region">The region of the <paramref name="graphics"/> that will be rasterised.</param>
        /// <param name="scale">The scale at which the image will be rendered.</param>
        /// <param name="interpolate">Whether the resulting image should be interpolated or not when it is drawn on another <see cref="Graphics"/> surface.</param>
        /// <returns>A <see cref="RasterImage"/> containing the rasterised graphics.</returns>
        public static RasterImage Rasterise(this Graphics graphics, Rectangle region, double scale, bool interpolate)
        {
            Page pag = new Page(1, 1);
            pag.Graphics.DrawGraphics(0, 0, graphics);
            pag.Crop(region.Location, region.Size);

            Image<SixLabors.ImageSharp.PixelFormats.Rgba32> img = SaveAsImage(pag, scale);

            int stride = img.Width * 4;
            int size = stride * img.Height;

            IntPtr tbr = System.Runtime.InteropServices.Marshal.AllocHGlobal(size);
            GC.AddMemoryPressure(size);

            IntPtr pointer = tbr;

            unsafe
            {
                for (int y = 0; y < img.Height; y++)
                {
                    Memory<SixLabors.ImageSharp.PixelFormats.Rgba32> row = img.DangerousGetPixelRowMemory(y);

                    Span<SixLabors.ImageSharp.PixelFormats.Rgba32> newRow = new Span<SixLabors.ImageSharp.PixelFormats.Rgba32>(pointer.ToPointer(), row.Length);
                    row.Span.CopyTo(newRow);

                    pointer = IntPtr.Add(pointer, stride);
                }
            }

            int width = img.Width;
            int height = img.Height;

            img.Dispose();

            DisposableIntPtr disp = new DisposableIntPtr(tbr);

            return new RasterImage(ref disp, width, height, true, interpolate);
        }

        /// <summary>
        /// Saves the animation to an animated GIF.
        /// </summary>
        /// <param name="animation">The animation to export.</param>
        /// <param name="scale">The scale at which the animation will be rendered.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated GIF specification at 50 fps.</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="colorTableMode">Determines whether a single colour table should be used for the whole image, or if a different colour table should be used for each frame.</param>
        /// <returns>An <see cref="Image"/> containing the animated GIF.</returns>
        public static Image SaveAsAnimatedGIF(this Animation animation, double scale = 1, double frameRate = 50, double durationScaling = 1, GifColorTableMode colorTableMode = GifColorTableMode.Local)
        {
            frameRate = Math.Min(frameRate, 50);

            Image gifAnimation = new Image<SixLabors.ImageSharp.PixelFormats.Rgba32>((int)(animation.Width * scale), (int)(animation.Height * scale));

            int frames = (int)Math.Ceiling(animation.Duration * frameRate * durationScaling / 1000);

            double accumulatedDelay = 0;

            for (int i = 0; i < frames; i++)
            {
                double frameTime = i / frameRate / durationScaling * 1000;
                double frameDuration = Math.Min(animation.Duration - i / durationScaling / frameRate * 1000, 1000 / frameRate / durationScaling);

                int gifFrameDuration = (int)Math.Round(frameDuration * durationScaling * 0.1 + accumulatedDelay);
```

```

                if (gifFrameDuration < 2)
                {
                    accumulatedDelay += frameDuration * durationScaling * 0.1;
                }
                else
                {
                    accumulatedDelay += frameDuration * durationScaling * 0.1 - gifFrameDuration;

                    Page pag = animation.GetFrameAtAbsolute(frameTime);

                    Image frame = pag.SaveAsImage(scale);

                    GifFrameMetadata frameMetadata = frame.Frames[0].Metadata.GetFormatMetadata(GifFormat.Instance);
                    frameMetadata.FrameDelay = gifFrameDuration;
                    frameMetadata.DisposalMethod = GifDisposalMethod.RestoreToPrevious;
                    gifAnimation.Frames.AddFrame(frame.Frames[0]);
                }
            }

            if ((int)accumulatedDelay > 0)
            {
                gifAnimation.Frames[gifAnimation.Frames.Count - 1].Metadata.GetFormatMetadata(GifFormat.Instance).FrameDelay += (int)accumulatedDelay;
            }

            gifAnimation.Frames.RemoveFrame(0);

            GifMetadata metadata = gifAnimation.Metadata.GetFormatMetadata(GifFormat.Instance);
            metadata.ColorTableMode = colorTableMode;
            metadata.RepeatCount = (ushort)animation.RepeatCount;

            return gifAnimation;
        }

        /// <summary>
        /// Saves the animation to an animated GIF stream.
        /// </summary>
        /// <param name="animation">The animation to export.</param>
        /// <param name="imageStream">The stream on which the animated GIF will be written.</param>
        /// <param name="scale">The scale at which the animation will be rendered.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated GIF specification at 50 fps.</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="colorTableMode">Determines whether a single colour table should be used for the whole image, or if a different colour table should be used for each frame.</param>
        public static void SaveAsAnimatedGIF(this Animation animation, Stream imageStream, double scale = 1, double frameRate = 50, double durationScaling = 1, GifColorTableMode colorTableMode = GifColorTableMode.Local)
        {
            Image gifAnimation = SaveAsAnimatedGIF(animation, scale, frameRate, durationScaling, colorTableMode);

            gifAnimation.SaveAsGif(imageStream);
        }

        /// <summary>
        /// Saves the animation to an animated GIF file.
        /// </summary>
        /// <param name="animation">The animation to export.</param>
        /// <param name="fileName">The output file to create.</param>
        /// <param name="scale">The scale at which the animation will be rendered.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated GIF specification at 50 fps.</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="colorTableMode">Determines whether a single colour table should be used for the whole image, or if a different colour table should be used for each frame.</param>
        public static void SaveAsAnimatedGIF(this Animation animation, string fileName, double scale = 1, double frameRate = 50, double durationScaling = 1, GifColorTableMode colorTableMode = GifColorTableMode.Local)
        {
            using (FileStream fs = File.Create(fileName))
            {
                SaveAsAnimatedGIF(animation, fs, scale, frameRate, durationScaling, colorTableMode);
            }
        }

        /// <summary>
        /// Saves the animation to a stream in animated PNG format.
        /// </summary>
        /// <param name="animation">The animation to export.</param>
        /// <param name="imageStream">The stream on which the animated PNG will be written.</param>
```

```
        /// <param name="scale">The scale at which the animation will be rendered.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated PNG specification at 90 fps.</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="interframeCompression">The kind of compression that will be used to reduce file size. Note that if the animation has a transparent background, no compression can be performed, and the value of this parameter is ignored.</param>
        public static void SaveAsAnimatedPNG(this Animation animation, Stream imageStream, double scale = 1, double frameRate = 60, double durationScaling = 1, AnimatedPNG.InterframeCompression interframeCompression = AnimatedPNG.InterframeCompression.First)
        {
            frameRate = Math.Min(frameRate, 90);

            int frames = (int)Math.Ceiling(animation.Duration * frameRate * durationScaling / 1000);

            Stream fs = imageStream;

            AnimatedPNG.CompressedFrame[] compressedFrames = new AnimatedPNG.CompressedFrame[frames];

            int width = (int)(animation.Width * scale);
            int height = (int)(animation.Height * scale);

            if (animation.Background.A < 1 || interframeCompression == AnimatedPNG.InterframeCompression.None)
            {
                Parallel.For(0, frames, i =>
                {
                    double frameTime = i / frameRate / durationScaling * 1000;

                    double frameDuration = Math.Min((animation.Duration - frameTime) * durationScaling, 1000 / frameRate);

                    Page pag = animation.GetFrameAtAbsolute(frameTime);

                    using (DisposableIntPtr rawFrame = pag.SaveAsRawBytes(out _, out _, out _, scale))
                    {
                        compressedFrames[i] = new AnimatedPNG.CompressedFrame(rawFrame, width, height, true, frameDuration);
                    }
                });
            }
            else if (interframeCompression == AnimatedPNG.InterframeCompression.First)
            {
                DisposableIntPtr firstFrame = animation.GetFrameAtAbsolute(0).SaveAsRawBytes(out _, out _, out _, scale);
                compressedFrames[0] = new AnimatedPNG.CompressedFrame(firstFrame, width, height, true, Math.Min(animation.Duration * durationScaling, 1000 / frameRate));

                Parallel.For(1, frames, i =>
                {
                    double frameTime = i / frameRate / durationScaling * 1000;

                    double frameDuration = Math.Min((animation.Duration - frameTime) * durationScaling, 1000 / frameRate);

                    Page pag = animation.GetFrameAtAbsolute(frameTime);

                    using (DisposableIntPtr rawFrame = pag.SaveAsRawBytes(out _, out _, out _, scale))
                    {
                        compressedFrames[i] = new AnimatedPNG.CompressedFrame(rawFrame, firstFrame, true, width, height, true, frameDuration);
                    }
                });

                firstFrame.Dispose();
            }
            else if (interframeCompression == AnimatedPNG.InterframeCompression.Previous)
            {
                DisposableIntPtr[] rawFrames = new DisposableIntPtr[frames];

                Parallel.For(0, frames, i =>
                {
                    double frameTime = i / frameRate / durationScaling * 1000;
                    Page pag = animation.GetFrameAtAbsolute(frameTime);
                    rawFrames[i] = pag.SaveAsRawBytes(out _, out _, out _, scale);
                });

                compressedFrames[0] = new AnimatedPNG.CompressedFrame(rawFrames[0], width, height, true, Math.Min(animation.Duration * durationScaling, 1000 / frameRate));

                Parallel.For(1, frames, i =>
                {
                    double frameTime = i / frameRate / durationScaling * 1000;
                    double frameDuration = Math.Min((animation.Duration - frameTime) * durationScaling, 1000 / frameRate);
```

```
                    compressedFrames[i] = new AnimatedPNG.CompressedFrame(rawFrames[i], rawFrames[i - 1], false, width, height, true, frameDuration);
                });
            }

            AnimatedPNG.Create(fs, (int)(animation.Width * scale), (int)(animation.Height * scale), true, compressedFrames, animation.RepeatCount);

            for (int i = 0; i < compressedFrames.Length; i++)
            {
                compressedFrames[i].Dispose();
            }
        }

        /// <summary>
        /// Saves the animation to an animated PNG file.
        /// </summary>
        /// <param name="animation">The animation to export.</param>
        /// <param name="fileName">The output file to create.</param>
        /// <param name="scale">The scale at which the animation will be rendered.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated PNG specification at 90 fps.</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="interframeCompression">The kind of compression that will be used to reduce file size. Note that if the animation has a transparent background, no compression can be performed, and the value of this parameter is ignored.</param>
        public static void SaveAsAnimatedPNG(this Animation animation, string fileName, double scale = 1, double frameRate = 60, double durationScaling = 1, AnimatedPNG.InterframeCompression interframeCompression = AnimatedPNG.InterframeCompression.First)
        {
            using (FileStream fs = File.Create(fileName))
            {
                SaveAsAnimatedPNG(animation, fs, scale, frameRate, durationScaling, interframeCompression);
            }
        }
    }
}
```

8.22 Raster.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020  Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Affero General Public License as
    published by the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU Affero General Public License for more details.

    You should have received a copy of the GNU Affero General Public License
    along with this program.  If not, see <http://www.gnu.org/licenses/>
*/

using MuPDFCore;
using System;
using System.IO;
using System.Runtime.InteropServices;
using System.Threading.Tasks;
using VectSharp.PDF;

namespace VectSharp.Raster
{
    /// <summary>
    /// Contains methods to render a page to a PNG image.
    /// </summary>
    public static class Raster
    {

        /// <summary>
        /// Render the page to a PNG file.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="fileName">The full path to the file to save. If it exists, it will be overwritten.</param>
        /// <param name="scale">The scale to be used when rasterising the page. This will determine the width and height of the image file.</param>
```

```
        public static void SaveAsPNG(this Page page, string fileName, double scale = 1)
        {
            Document doc = new Document();
            doc.Pages.Add(page);

            MemoryStream ms = new MemoryStream();
            doc.SaveAsPDF(ms, textOption: PDFContextInterpreter.TextOptions.ConvertIntoPaths, filterOption: new PDFContextInterpreter.FilterOption(PDFContextInterpreter.FilterOption.FilterOperations.RasteriseAll, scale, true));

            using (MuPDFContext context = new MuPDFContext())
            {
                using (MuPDFDocument muDoc = new MuPDFDocument(context, ref ms, InputFileTypes.PDF))
                {
                    muDoc.SaveImage(0, scale, MuPDFCore.PixelFormats.RGBA, fileName, RasterOutputFileTypes.PNG);
                }
            }

            ms.Dispose();
        }

        /// <summary>
        /// Render the page to a PNG stream.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="stream">The stream to which the PNG data will be written.</param>
        /// <param name="scale">The scale to be used when rasterising the page. This will determine the width and height of the image file.</param>
        public static void SaveAsPNG(this Page page, Stream stream, double scale = 1)
        {
            Document doc = new Document();
            doc.Pages.Add(page);

            MemoryStream ms = new MemoryStream();
            doc.SaveAsPDF(ms, textOption: PDFContextInterpreter.TextOptions.ConvertIntoPaths, filterOption: new PDFContextInterpreter.FilterOption(PDFContextInterpreter.FilterOption.FilterOperations.RasteriseAll, scale, true));

            using (MuPDFContext context = new MuPDFContext())
            {
                using (MuPDFDocument muDoc = new MuPDFDocument(context, ref ms, InputFileTypes.PDF))
                {
                    muDoc.WriteImage(0, scale, MuPDFCore.PixelFormats.RGBA, stream, RasterOutputFileTypes.PNG);
                }
            }

            ms.Dispose();
        }

        /// <summary>
        /// Rasterise a region of a <see cref="Graphics"/> object.
        /// </summary>
        /// <param name="graphics">The <see cref="Graphics"/> object that will be rasterised.</param>
        /// <param name="region">The region of the <paramref name="graphics"/> that will be rasterised.</param>
        /// <param name="scale">The scale at which the image will be rendered.</param>
        /// <param name="interpolate">Whether the resulting image should be interpolated or not when it is drawn on another <see cref="Graphics"/> surface.</param>
        /// <returns>A <see cref="RasterImage"/> containing the rasterised graphics.</returns>
        public static RasterImage Rasterise(this Graphics graphics, Rectangle region, double scale, bool interpolate)
        {
            Page pag = new Page(1, 1);
            pag.Graphics.DrawGraphics(0, 0, graphics);
            pag.Crop(region.Location, region.Size);

            Document doc = new Document();
            doc.Pages.Add(pag);

            MemoryStream ms = new MemoryStream();
            doc.SaveAsPDF(ms, textOption: PDFContextInterpreter.TextOptions.ConvertIntoPaths, filterOption: new PDFContextInterpreter.FilterOption(PDFContextInterpreter.FilterOption.FilterOperations.RasteriseAll, scale, true));

            IntPtr imageDataAddress;

            int width;
            int height;

            using (MuPDFContext context = new MuPDFContext())
            {
                using (MuPDFDocument muDoc = new MuPDFDocument(context, ref ms, InputFileTypes.PDF))
```

```
                {
                    int imageSize = muDoc.GetRenderedSize(0, scale, MuPDFCore.PixelFormats.RGBA);

                    RoundedRectangle roundedBounds = muDoc.Pages[0].Bounds.Round(scale);

                    width = roundedBounds.Width;
                    height = roundedBounds.Height;


                    imageDataAddress = Marshal.AllocHGlobal(imageSize);
                    GC.AddMemoryPressure(imageSize);

                    muDoc.Render(0, scale, MuPDFCore.PixelFormats.RGBA, imageDataAddress);
                }
            }

            ms.Dispose();

            DisposableIntPtr disposableAddress = new DisposableIntPtr(imageDataAddress);

            return new RasterImage(ref disposableAddress, width, height, true, interpolate);
        }

        /// <summary>
        /// Render the page to raw pixel data, in 32bpp RGBA format.
        /// </summary>
        /// <param name="pag">The <see cref="Page"/> to render.</param>
        /// <param name="scale">The scale to be used when rasterising the page. This will determine the width and height of the image.</param>
        /// <param name="width">The width of the rendered image.</param>
        /// <param name="height">The height of the rendered image.</param>
        /// <param name="totalSize">The size in bytes of the raw pixel data.</param>
        /// <returns>A <see cref="DisposableIntPtr"/> containing a pointer to the raw pixel data, stored in unmanaged memory. Dispose this object to release the unmanaged memory.</returns>
        public static DisposableIntPtr SaveAsRawBytes(this Page pag, out int width, out int height, out int totalSize, double scale = 1)
        {
            Document doc = new Document();
            doc.Pages.Add(pag);

            MemoryStream ms = new MemoryStream();
            doc.SaveAsPDF(ms, textOption: PDFContextInterpreter.TextOptions.ConvertIntoPaths, filterOption: new PDFContextInterpreter.FilterOption(PDFContextInterpreter.FilterOption.FilterOperations.RasteriseAll, scale, true));

            IntPtr renderedPage;

            using (MuPDFContext context = new MuPDFContext())
            {
                using (MuPDFDocument muDoc = new MuPDFDocument(context, ref ms, InputFileTypes.PDF))
                {
                    totalSize = muDoc.GetRenderedSize(0, scale, MuPDFCore.PixelFormats.RGBA);

                    renderedPage = Marshal.AllocHGlobal(totalSize);

                    RoundedRectangle bounds = muDoc.Pages[0].Bounds.Round(scale);

                    width = bounds.Width;
                    height = bounds.Height;

                    muDoc.Render(0, scale, MuPDFCore.PixelFormats.RGBA, renderedPage);
                }
            }

            ms.Dispose();

            return new DisposableIntPtr(renderedPage);
        }

        /// <summary>
        /// Saves the animation to a stream in animated PNG format.
        /// </summary>
        /// <param name="animation">The animation to export.</param>
        /// <param name="imageStream">The stream on which the animated PNG will be written.</param>
        /// <param name="scale">The scale at which the animation will be rendered.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated PNG specification at 90 fps.</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="interframeCompression">The kind of compression that will be used to reduce file size. Note that if the animation has a transparent background, no compression can be performed, and the value of this parameter is ignored.</param>
        public static void SaveAsAnimatedPNG(this Animation animation, Stream imageStream, double scale = 1, double frameRate = 60, double durationScaling = 1, AnimatedPNG.InterframeCompression interframeCompression = AnimatedPNG.InterframeCompression.First)
```

```
        {
            frameRate = Math.Min(frameRate, 90);

            int frames = (int)Math.Ceiling(animation.Duration * frameRate * durationScaling / 1000);

            Stream fs = imageStream;

            AnimatedPNG.CompressedFrame[] compressedFrames = new AnimatedPNG.CompressedFrame[frames];

            int width = (int)(animation.Width * scale);
            int height = (int)(animation.Height * scale);

            if (animation.Background.A < 1 || interframeCompression == AnimatedPNG.InterframeCompression.None)
            {
                Parallel.For(0, frames, i =>
                {
                    double frameTime = i / frameRate / durationScaling * 1000;

                    double frameDuration = Math.Min((animation.Duration - frameTime) * durationScaling, 1000 / frameRate);

                    Page pag = animation.GetFrameAtAbsolute(frameTime);

                    using (DisposableIntPtr rawFrame = pag.SaveAsRawBytes(out _, out _, out _, scale))
                    {
                        compressedFrames[i] = new AnimatedPNG.CompressedFrame(rawFrame, width, height, true, frameDuration);
                    }
                });
            }
            else if (interframeCompression == AnimatedPNG.InterframeCompression.First)
            {
                DisposableIntPtr firstFrame = animation.GetFrameAtAbsolute(0).SaveAsRawBytes(out _, out _, out _, scale);
                compressedFrames[0] = new AnimatedPNG.CompressedFrame(firstFrame, width, height, true, Math.Min(animation.Duration * durationScaling, 1000 / frameRate));

                Parallel.For(1, frames, i =>
                {
                    double frameTime = i / frameRate / durationScaling * 1000;

                    double frameDuration = Math.Min((animation.Duration - frameTime) * durationScaling, 1000 / frameRate);

                    Page pag = animation.GetFrameAtAbsolute(frameTime);

                    using (DisposableIntPtr rawFrame = pag.SaveAsRawBytes(out _, out _, out _, scale))
                    {
                        compressedFrames[i] = new AnimatedPNG.CompressedFrame(rawFrame, firstFrame, true, width, height, true, frameDuration);
                    }
                });

                firstFrame.Dispose();
            }
            else if (interframeCompression == AnimatedPNG.InterframeCompression.Previous)
            {
                DisposableIntPtr[] rawFrames = new DisposableIntPtr[frames];

                Parallel.For(0, frames, i =>
                {
                    double frameTime = i / frameRate / durationScaling * 1000;
                    Page pag = animation.GetFrameAtAbsolute(frameTime);
                    rawFrames[i] = pag.SaveAsRawBytes(out _, out _, out _, scale);
                });

                compressedFrames[0] = new AnimatedPNG.CompressedFrame(rawFrames[0], width, height, true, Math.Min(animation.Duration * durationScaling, 1000 / frameRate));

                Parallel.For(1, frames, i =>
                {
                    double frameTime = i / frameRate / durationScaling * 1000;
                    double frameDuration = Math.Min((animation.Duration - frameTime) * durationScaling, 1000 / frameRate);
                    compressedFrames[i] = new AnimatedPNG.CompressedFrame(rawFrames[i], rawFrames[i - 1], false, width, height, true, frameDuration);
                });
            }

            AnimatedPNG.Create(fs, (int)(animation.Width * scale), (int)(animation.Height * scale), true, compressedFrames, animation.RepeatCount);
        }

        /// <summary>
        /// Saves the animation to an animated PNG file.
        /// </summary>
```

```
        /// <param name="animation">The animation to export.</param>
        /// <param name="fileName">The output file to create.</param>
        /// <param name="scale">The scale at which the animation will be rendered.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second (fps). This is capped by the animated PNG specification at 90 fps.</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="interframeCompression">The kind of compression that will be used to reduce file size. Note that if the animation has a transparent background, no compression can be performed, and the value of this parameter is ignored.</param>
        public static void SaveAsAnimatedPNG(this Animation animation, string fileName, double scale = 1, double frameRate = 60, double durationScaling = 1, AnimatedPNG.InterframeCompression interframeCompression = AnimatedPNG.InterframeCompression.First)
        {
            using (FileStream fs = File.Create(fileName))
            {
                SaveAsAnimatedPNG(animation, fs, scale, frameRate, durationScaling, interframeCompression);
            }
        }
    }
}
```

8.23 SVGContext.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using ExCSS;
using System;
using System.Collections.Generic;
using System.Data;
using System.IO;
using System.Linq;
using System.Reflection;
using System.Text;
using System.Threading.Tasks;
using System.Transactions;
using System.Xml;
using VectSharp.Filters;

namespace VectSharp.SVG
{
    internal static class MatrixUtils
    {
        public static double[] Multiply(double[,] matrix, double[] vector)
        {
            double[] tbr = new double[2];

            tbr[0] = matrix[0, 0] * vector[0] + matrix[0, 1] * vector[1] + matrix[0, 2];
            tbr[1] = matrix[1, 0] * vector[0] + matrix[1, 1] * vector[1] + matrix[1, 2];

            return tbr;
        }

        public static Point Multiply(double[,] matrix, Point vector)
        {
            double[] tbr = new double[2];

            tbr[0] = matrix[0, 0] * vector.X + matrix[0, 1] * vector.Y + matrix[0, 2];
            tbr[1] = matrix[1, 0] * vector.X + matrix[1, 1] * vector.Y + matrix[1, 2];

            return new Point(tbr[0], tbr[1]);
        }

        public static double[,] Multiply(double[,] matrix1, double[,] matrix2)
        {
            double[,] tbr = new double[3, 3];
```

```
            for (int i = 0; i < 3; i++)
            {
                for (int j = 0; j < 3; j++)
                {
                    for (int k = 0; k < 3; k++)
                    {
                        tbr[i, j] += matrix1[i, k] * matrix2[k, j];
                    }
                }
            }

            return tbr;
        }

        public static double[,] Rotate(double[,] matrix, double angle)
        {
            double[,] rotationMatrix = new double[3, 3];
            rotationMatrix[0, 0] = Math.Cos(angle);
            rotationMatrix[0, 1] = -Math.Sin(angle);
            rotationMatrix[1, 0] = Math.Sin(angle);
            rotationMatrix[1, 1] = Math.Cos(angle);
            rotationMatrix[2, 2] = 1;

            return Multiply(matrix, rotationMatrix);
        }

        public static double[,] Translate(double[,] matrix, double x, double y)
        {
            double[,] translationMatrix = new double[3, 3];
            translationMatrix[0, 0] = 1;
            translationMatrix[0, 2] = x;
            translationMatrix[1, 1] = 1;
            translationMatrix[1, 2] = y;
            translationMatrix[2, 2] = 1;

            return Multiply(matrix, translationMatrix);
        }

        public static double[,] Scale(double[,] matrix, double scaleX, double scaleY)
        {
            double[,] scaleMatrix = new double[3, 3];
            scaleMatrix[0, 0] = scaleX;
            scaleMatrix[1, 1] = scaleY;
            scaleMatrix[2, 2] = 1;

            return Multiply(matrix, scaleMatrix);
        }

        public static double[,] Identity = new double[,] { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };

        public static double[,] Invert(double[,] m)
        {
            double[,] tbr = new double[3, 3];

            tbr[0, 0] = (m[1, 1] * m[2, 2] - m[1, 2] * m[2, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[0, 1] = -(m[0, 1] * m[2, 2] - m[0, 2] * m[2, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[0, 2] = (m[0, 1] * m[1, 2] - m[0, 2] * m[1, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 0] = -(m[1, 0] * m[2, 2] - m[1, 2] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 1] = (m[0, 0] * m[2, 2] - m[0, 2] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 2] = -(m[0, 0] * m[1, 2] - m[0, 2] * m[1, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 0] = (m[1, 0] * m[2, 1] - m[1, 1] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 1] = -(m[0, 0] * m[2, 1] - m[0, 1] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 2] = (m[0, 0] * m[1, 1] - m[0, 1] * m[1, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);

            return tbr;
        }
    }
```

```
    internal class SVGFigure
    {
        public Point StartPoint { get; set; }
        public Point CurrentPoint { get; set; }

        public string Data { get; set; }
        public int PointCount { get; set; } = 0;
    }

    internal class SVGPathObject
    {
        public List<SVGFigure> Figures { get; set; } = new List<SVGFigure>();
    }


    internal class SVGContext : IGraphicsContext
    {
        public Dictionary<string, FontFamily> UsedFontFamilies;
        public Dictionary<string, HashSet<char» UsedChars;


        public const string SVGNamespace = "http://www.w3.org/2000/svg";

        public double Width { get; }

        public double Height { get; }

        public Font Font { get; set; }
        public TextBaselines TextBaseline { get; set; }

        public Brush FillStyle { get; private set; }

        public Brush StrokeStyle { get; private set; }

        public double LineWidth { get; set; }
        public LineCaps LineCap { private get; set; }
        public LineJoins LineJoin { private get; set; }

        private LineDash _lineDash;

        private SVGPathObject currentPath;
        private SVGFigure currentFigure;

        public XmlDocument Document;
        private XmlElement currentElement;

        public string Tag { get; set; }
        public string TagPrefix { get; }

        private bool ReuseGradients { get; }

        private double[,] _transform;
        private Stack<double[,]> states;

        private string _currClipPath;
        private Stack<string> clipPaths;

        private bool TextToPaths = false;
        private SVGContextInterpreter.TextOptions TextOption =
SVGContextInterpreter.TextOptions.SubsetFonts;

        private Dictionary<string, string> linkDestinations;

        XmlElement definitions;
        Dictionary<Brush, string> gradients;
        int gradientCount = 0;

        private SVGContextInterpreter.FilterOption FilterOption;

        private bool UseStyles;
        internal Dictionary<string, Dictionary<string, string» Styles;
        XmlElement StyleElement;

        private string GetClass(Dictionary<string, string> style)
        {
            foreach (KeyValuePair<string, Dictionary<string, string» styleClass in Styles)
            {
                if (styleClass.Value.Count == style.Count)
                {
                    bool mismatch = false;

                    foreach (KeyValuePair<string, string> styleItem in style)
                    {
                        if (!styleClass.Value.TryGetValue(styleItem.Key, out string value))
                        {
                            mismatch = true;
                            break;
```

```
                        }
                        else if (value != styleItem.Value)
                        {
                            mismatch = true;
                            break;
                        }
                    }

                    if (!mismatch)
                    {
                        return styleClass.Key;
                    }
                }
            }

            string tbr = "class" + (Styles.Count + 1).ToString();

            Styles.Add(tbr, style);

            return tbr;
        }

        public SVGContext(double width, double height, bool textToPaths, SVGContextInterpreter.TextOptions textOption, Dictionary<string, string> linkDestinations, SVGContextInterpreter.FilterOption filterOption, bool useStyles, string tagPrefix, bool reuseGradients)
        {
            this.linkDestinations = linkDestinations;

            this.TagPrefix = tagPrefix;

            this.Width = width;
            this.Height = height;

            currentPath = new SVGPathObject();
            currentFigure = new SVGFigure();

            StrokeStyle = Colour.FromRgba(0, 0, 0, 0);
            FillStyle = Colour.FromRgb(0, 0, 0);
            LineWidth = 1;

            LineCap = LineCaps.Butt;
            LineJoin = LineJoins.Miter;
            _lineDash = new LineDash(0, 0, 0);

            Font = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica), 12);

            TextBaseline = TextBaselines.Top;

            _transform = new double[3, 3];

            _transform[0, 0] = 1;
            _transform[1, 1] = 1;
            _transform[2, 2] = 1;

            states = new Stack<double[,]>();

            clipPaths = new Stack<string>();
            clipPaths.Push(null);
            _currClipPath = null;

            UsedFontFamilies = new Dictionary<string, FontFamily>();
            UsedChars = new Dictionary<string, HashSet<char»();

            Document = new XmlDocument();

            Document.InsertBefore(Document.CreateXmlDeclaration("1.0", "UTF-8", null), Document.DocumentElement);

            currentElement = Document.CreateElement(null, "svg", SVGNamespace);
            currentElement.SetAttribute("xmlns:xlink", "http://www.w3.org/1999/xlink");
            currentElement.SetAttribute("viewBox", "0 0 " + width.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + height.ToString(System.Globalization.CultureInfo.InvariantCulture));
            currentElement.SetAttribute("version", "1.1");
            Document.AppendChild(currentElement);

            definitions = Document.CreateElement("defs", SVGNamespace);
            gradients = new Dictionary<Brush, string>();
            currentElement.AppendChild(definitions);

            this.UseStyles = useStyles;
            StyleElement = Document.CreateElement("style", SVGNamespace);
            StyleElement.InnerText = "\t";
            currentElement.AppendChild(StyleElement);
            Styles = new Dictionary<string, Dictionary<string, string»();
```

```
            this.TextToPaths = textToPaths;
            this.TextOption = textOption;
            this.FilterOption = filterOption;
            this.ReuseGradients = reuseGradients;
        }


        private void PathText(string text, double x, double y)
        {
            GraphicsPath textPath = new GraphicsPath().AddText(x, y, text, Font, TextBaseline);

            for (int j = 0; j < textPath.Segments.Count; j++)
            {
                switch (textPath.Segments[j].Type)
                {
                    case VectSharp.SegmentType.Move:
                        this.MoveTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.Line:
                        this.LineTo(textPath.Segments[j].Point.X, textPath.Segments[j].Point.Y);
                        break;
                    case VectSharp.SegmentType.CubicBezier:
                        this.CubicBezierTo(textPath.Segments[j].Points[0].X, textPath.Segments[j].Points[0].Y, textPath.Segments[j].Points[1].X, textPath.Segments[j].Points[1].Y, textPath.Segments[j].Points[2].X, textPath.Segments[j].Points[2].Y);
                        break;
                    case VectSharp.SegmentType.Close:
                        this.Close();
                        break;
                }
            }
        }

        public void Close()
        {
            if (currentFigure.PointCount > 0)
            {
                currentFigure.Data += "Z ";
                currentPath.Figures.Add(currentFigure);

                currentFigure = new SVGFigure();
            }
        }

        public void CubicBezierTo(double p1X, double p1Y, double p2X, double p2Y, double p3X, double p3Y)
        {
            if (currentFigure.PointCount == 0)
            {
                currentFigure.StartPoint = new Point(p1X, p1Y);
            }

            currentFigure.CurrentPoint = new Point(p3X, p3Y);
            currentFigure.Data += "C " + p1X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + p1Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + ", " +
                p2X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + p2Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + ", " +
                p3X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + p3Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ";
            currentFigure.PointCount += 3;
        }

        public void Fill()
        {
            if (currentFigure.PointCount > 0)
            {
                currentPath.Figures.Add(currentFigure);
            }

            XmlElement currElement = currentElement;

            if (!string.IsNullOrEmpty(_currClipPath))
            {
                currentElement = Document.CreateElement("g", SVGNamespace);
                currentElement.SetAttribute("clip-path", _currClipPath);
                currElement.AppendChild(currentElement);
            }

            XmlElement path = Document.CreateElement("path", SVGNamespace);
            path.SetAttribute("d", currentPath.Figures.Aggregate("", (a, b) => a + b.Data));

            string gradientName = null;

            {
                if (FillStyle is LinearGradientBrush linearGradient)
```

```
                {
                    if (!ReuseGradients || !gradients.TryGetValue(linearGradient, out gradientName))
                    {
                        if (!string.IsNullOrEmpty(Tag))
                        {
                            gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Tag;
                        }
                        else
                        {
                            gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Guid.NewGuid().ToString("N");
                        }

                        XmlElement gradientElement = linearGradient.ToLinearGradient(Document, gradientName);
                        this.definitions.AppendChild(gradientElement);

                        if (ReuseGradients)
                        {
                            gradients.Add(linearGradient, gradientName);
                        }

                        gradientCount++;
                    }
                }
                else if (FillStyle is RadialGradientBrush radialGradient)
                {
                    if (!ReuseGradients || !gradients.TryGetValue(radialGradient, out gradientName))
                    {
                        if (!string.IsNullOrEmpty(Tag))
                        {
                            gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Tag;
                        }
                        else
                        {
                            gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Guid.NewGuid().ToString("N");
                        }

                        XmlElement gradientElement = radialGradient.ToRadialGradient(Document, gradientName);
                        this.definitions.AppendChild(gradientElement);

                        if (ReuseGradients)
                        {
                            gradients.Add(radialGradient, gradientName);
                        }

                        gradientCount++;
                    }
                }
            }

            if (!UseStyles)
            {
                path.SetAttribute("stroke", "none");

                if (FillStyle is SolidColourBrush solid)
                {
                    path.SetAttribute("fill", solid.Colour.ToCSSString(false));
                    path.SetAttribute("fill-opacity", solid.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
                }
                else if (FillStyle is LinearGradientBrush linearGradient)
                {
                    path.SetAttribute("fill", "url(#" + gradientName + ")");
                }
                else if (FillStyle is RadialGradientBrush radialGradient)
                {
                    path.SetAttribute("fill", "url(#" + gradientName + ")");
                }

                path.SetAttribute("transform", "matrix(" + _transform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");
            }
            else
            {
                Dictionary<string, string> style = new Dictionary<string, string>();

                style.Add("stroke", "none");
```

```
                if (FillStyle is SolidColourBrush solid)
                {
                    style.Add("fill", solid.Colour.ToCSSString(false));
                    style.Add("fill-opacity", solid.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
                }
                else if (FillStyle is LinearGradientBrush linearGradient)
                {
                    style.Add("fill", "url(#" + gradientName + ")");
                }
                else if (FillStyle is RadialGradientBrush radialGradient)
                {
                    style.Add("fill", "url(#" + gradientName + ")");
                }

                style.Add("transform", "matrix(" + _transform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");

                string className = GetClass(style);

                path.SetAttribute("class", className);
            }

            if (!string.IsNullOrEmpty(Tag))
            {
                path.SetAttribute("id", this.TagPrefix + Tag);
            }

            if (!string.IsNullOrEmpty(this.Tag) && this.linkDestinations.TryGetValue(this.Tag, out string destination) && !string.IsNullOrEmpty(destination))
            {
                XmlElement aElement = Document.CreateElement("a", SVGNamespace);
                aElement.SetAttribute("href", destination);
                currentElement.AppendChild(aElement);
                currentElement = aElement;
            }

            currentElement.AppendChild(path);

            currentElement = currElement;

            currentPath = new SVGPathObject();
            currentFigure = new SVGFigure();

        }

        public void FillText(string text, double x, double y)
        {
            if (!TextToPaths)
            {
                if (!UsedFontFamilies.ContainsKey(Font.FontFamily.FileName))
                {
                    UsedFontFamilies.Add(Font.FontFamily.FileName, Font.FontFamily);
                    UsedChars.Add(Font.FontFamily.FileName, new HashSet<char>());
                }

                UsedChars[Font.FontFamily.FileName].UnionWith(text);

                Font.DetailedFontMetrics metrics = Font.MeasureTextAdvanced(text);

                double[,] currTransform = null;
                double[,] deltaTransform = MatrixUtils.Identity;

                switch (TextBaseline)
                {
                    case TextBaselines.Baseline:
                        currTransform = MatrixUtils.Translate(_transform, x - metrics.LeftSideBearing, y);
                        deltaTransform = MatrixUtils.Translate(deltaTransform, x - metrics.LeftSideBearing, y);
                        break;
                    case TextBaselines.Top:
                        currTransform = MatrixUtils.Translate(_transform, x - metrics.LeftSideBearing, y + metrics.Top);
                        deltaTransform = MatrixUtils.Translate(deltaTransform, x - metrics.LeftSideBearing, y + metrics.Top);
                        break;
                    case TextBaselines.Bottom:
                        currTransform = MatrixUtils.Translate(_transform, x - metrics.LeftSideBearing, y + metrics.Bottom);
                        deltaTransform = MatrixUtils.Translate(deltaTransform, x -
```

```
metrics.LeftSideBearing, y + metrics.Bottom);
                        break;
                    case TextBaselines.Middle:
                        currTransform = MatrixUtils.Translate(_transform, x - metrics.LeftSideBearing, y + (metrics.Top + metrics.Bottom) * 0.5);
                        deltaTransform = MatrixUtils.Translate(deltaTransform, x - metrics.LeftSideBearing, y + (metrics.Top + metrics.Bottom) * 0.5);
                        break;
                    default:
                        currTransform = MatrixUtils.Translate(_transform, x - metrics.LeftSideBearing, y);
                        deltaTransform = MatrixUtils.Translate(deltaTransform, x - metrics.LeftSideBearing, y);
                        break;
                }

                XmlElement currElement = currentElement;

                if (!string.IsNullOrEmpty(_currClipPath))
                {
                    currentElement = Document.CreateElement("g", SVGNamespace);
                    currentElement.SetAttribute("clip-path", _currClipPath);
                    currElement.AppendChild(currentElement);
                }

                XmlElement textElement = Document.CreateElement("text", SVGNamespace);

                string gradientName = null;

                {
                    if (FillStyle is LinearGradientBrush linearGradient)
                    {
                        if (!string.IsNullOrEmpty(Tag))
                        {
                            gradientName = this.TagPrefix + "textGradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Tag;
                        }
                        else
                        {
                            gradientName = this.TagPrefix + "textGradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Guid.NewGuid().ToString("N");
                        }

                        XmlElement gradientElement = linearGradient.ToLinearGradient(Document, gradientName);

                        deltaTransform = MatrixUtils.Invert(deltaTransform);

                        gradientElement.SetAttribute("gradientTransform", "matrix(" + deltaTransform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + deltaTransform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + deltaTransform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + deltaTransform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + deltaTransform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + deltaTransform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");

                        this.definitions.AppendChild(gradientElement);

                        gradientCount++;
                    }
                    else if (FillStyle is RadialGradientBrush radialGradient)
                    {
                        if (!string.IsNullOrEmpty(Tag))
                        {
                            gradientName = this.TagPrefix + "textGradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Tag;
                        }
                        else
                        {
                            gradientName = this.TagPrefix + "textGradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Guid.NewGuid().ToString("N");
                        }

                        XmlElement gradientElement = radialGradient.ToRadialGradient(Document, gradientName);

                        deltaTransform = MatrixUtils.Invert(deltaTransform);

                        gradientElement.SetAttribute("gradientTransform", "matrix(" + deltaTransform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + deltaTransform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + deltaTransform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + deltaTransform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + deltaTransform[0,
```

```
2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + deltaTransform[1,
2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");

                        this.definitions.AppendChild(gradientElement);

                        gradientCount++;
                    }
                }

                textElement.SetAttribute("x", "0");
                textElement.SetAttribute("y", "0");

                textElement.SetAttribute("transform", "matrix(" + currTransform[0,
0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + currTransform[1,
0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + currTransform[0,
1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + currTransform[1,
1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + currTransform[0,
2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + currTransform[1,
2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");

                if (!UseStyles)
                {
                    textElement.SetAttribute("stroke", "none");

                    if (FillStyle is SolidColourBrush solid)
                    {
                        textElement.SetAttribute("fill", solid.Colour.ToCSSString(false));
                        textElement.SetAttribute("fill-opacity",
solid.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    }
                    else if (FillStyle is LinearGradientBrush linearGradient)
                    {
                        textElement.SetAttribute("fill", "url(#" + gradientName + ")");
                    }
                    else if (FillStyle is RadialGradientBrush radialGradient)
                    {
                        textElement.SetAttribute("fill", "url(#" + gradientName + ")");
                    }

                    textElement.SetAttribute("font-size",
Font.FontSize.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    textElement.SetAttribute("font-family", Font.FontFamily.FileName);

                    if (Font.FontFamily.IsBold)
                    {
                        textElement.SetAttribute("font-weight", "bold");
                    }
                    else
                    {
                        textElement.SetAttribute("font-weight", "regular");
                    }

                    if (Font.FontFamily.IsItalic)
                    {
                        textElement.SetAttribute("font-style", "italic");
                    }
                    else
                    {
                        textElement.SetAttribute("font-style", "normal");
                    }

                    if (Font.FontFamily.IsOblique)
                    {
                        textElement.SetAttribute("font-style", "oblique");
                    }
                }
                else
                {
                    Dictionary<string, string> style = new Dictionary<string, string>();

                    style.Add("stroke", "none");

                    if (FillStyle is SolidColourBrush solid)
                    {
                        style.Add("fill", solid.Colour.ToCSSString(false));
                        style.Add("fill-opacity",
solid.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    }
                    else if (FillStyle is LinearGradientBrush linearGradient)
                    {
                        style.Add("fill", "url(#" + gradientName + ")");
                    }
                    else if (FillStyle is RadialGradientBrush radialGradient)
                    {
                        style.Add("fill", "url(#" + gradientName + ")");
```

```
                    }

                    style.Add("font-size", Font.FontSize.ToString(System.Globalization.CultureInfo.InvariantCulture) + "px");
                    style.Add("font-family", Font.FontFamily.FileName);

                    if (Font.FontFamily.IsBold)
                    {
                        style.Add("font-weight", "bold");
                    }
                    else
                    {
                        style.Add("font-weight", "regular");
                    }

                    if (Font.FontFamily.IsItalic)
                    {
                        style["font-style"] = "italic";
                    }
                    else
                    {
                        style["font-style"] = "normal";
                    }

                    if (Font.FontFamily.IsOblique)
                    {
                        style["font-style"] = "oblique";
                    }

                    string className = GetClass(style);

                    textElement.SetAttribute("class", className);
                }

                ProcessText(text, textElement);

                if (!string.IsNullOrEmpty(Tag))
                {
                    textElement.SetAttribute("id", this.TagPrefix + Tag);
                }

                if (!string.IsNullOrEmpty(this.Tag) && this.linkDestinations.TryGetValue(this.Tag, out string destination) && !string.IsNullOrEmpty(destination))
                {
                    XmlElement aElement = Document.CreateElement("a", SVGNamespace);
                    aElement.SetAttribute("href", destination);
                    currentElement.AppendChild(aElement);
                    currentElement = aElement;
                }

                currentElement.AppendChild(textElement);

                currentElement = currElement;
            }
            else
            {
                PathText(text, x, y);
                Fill();
            }
        }

        public void LineTo(double x, double y)
        {
            if (currentFigure.PointCount == 0)
            {
                currentFigure.StartPoint = new Point(x, y);
            }

            currentFigure.CurrentPoint = new Point(x, y);
            currentFigure.Data += "L " + x.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ";
            currentFigure.PointCount++;
        }

        public void MoveTo(double x, double y)
        {
            if (currentFigure.PointCount > 0)
            {
                currentPath.Figures.Add(currentFigure);
            }

            currentFigure = new SVGFigure();
            currentFigure.CurrentPoint = new Point(x, y);
            currentFigure.StartPoint = new Point(x, y);
            currentFigure.Data += "M " + x.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " ";
            currentFigure.PointCount = 1;
```

```
        }

        public void Rectangle(double x0, double y0, double width, double height)
        {
            MoveTo(x0, y0);
            LineTo(x0 + width, y0);
            LineTo(x0 + width, y0 + height);
            LineTo(x0, y0 + height);
            Close();
        }

        public void Restore()
        {
            _transform = states.Pop();
            _currClipPath = clipPaths.Pop();
            currentPath = new SVGPathObject();
            currentFigure = new SVGFigure();
        }

        public void Rotate(double angle)
        {
            _transform = MatrixUtils.Rotate(_transform, angle);
            currentPath = new SVGPathObject(); currentPath = new SVGPathObject();
            currentFigure = new SVGFigure();
        }

        public void Save()
        {
            states.Push((double[,])_transform.Clone());
            clipPaths.Push(_currClipPath);
        }

        public void Scale(double scaleX, double scaleY)
        {
            _transform = MatrixUtils.Scale(_transform, scaleX, scaleY);
            currentPath = new SVGPathObject();
            currentFigure = new SVGFigure();
        }

        public void SetFillStyle((int r, int g, int b, double a) style)
        {
            FillStyle = Colour.FromRgba(style);
        }

        public void SetFillStyle(Brush style)
        {
            FillStyle = style;
        }

        public void SetLineDash(LineDash dash)
        {
            _lineDash = dash;
        }

        public void SetStrokeStyle((int r, int g, int b, double a) style)
        {
            StrokeStyle = Colour.FromRgba(style);
        }

        public void SetStrokeStyle(Brush style)
        {
            StrokeStyle = style;
        }

        public void Stroke()
        {
            if (currentFigure.PointCount > 0)
            {
                currentPath.Figures.Add(currentFigure);
            }

            XmlElement currElement = currentElement;

            if (!string.IsNullOrEmpty(_currClipPath))
            {
                currentElement = Document.CreateElement("g", SVGNamespace);
                currentElement.SetAttribute("clip-path", _currClipPath);
                currElement.AppendChild(currentElement);
            }

            XmlElement path = Document.CreateElement("path", SVGNamespace);
            path.SetAttribute("d", currentPath.Figures.Aggregate("", (a, b) => a + b.Data));

            string gradientName = null;

            {
                if (StrokeStyle is LinearGradientBrush linearGradient)
```

```
                {
                    if (!ReuseGradients || !gradients.TryGetValue(linearGradient, out gradientName))
                    {
                        if (!string.IsNullOrEmpty(Tag))
                        {
                            gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Tag;
                        }
                        else
                        {
                            gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Guid.NewGuid().ToString("N");
                        }

                        XmlElement gradientElement = linearGradient.ToLinearGradient(Document, gradientName);
                        this.definitions.AppendChild(gradientElement);

                        if (ReuseGradients)
                        {
                            gradients.Add(linearGradient, gradientName);
                        }

                        gradientCount++;
                    }
                }
                else if (StrokeStyle is RadialGradientBrush radialGradient)
                {
                    if (!ReuseGradients || !gradients.TryGetValue(radialGradient, out gradientName))
                    {
                        if (!string.IsNullOrEmpty(Tag))
                        {
                            gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Tag;
                        }
                        else
                        {
                            gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Guid.NewGuid().ToString("N");
                        }

                        XmlElement gradientElement = radialGradient.ToRadialGradient(Document, gradientName);
                        this.definitions.AppendChild(gradientElement);

                        if (ReuseGradients)
                        {
                            gradients.Add(radialGradient, gradientName);
                        }

                        gradientCount++;
                    }
                }
            }

            if (!UseStyles)
            {

                if (StrokeStyle is SolidColourBrush solid)
                {
                    path.SetAttribute("stroke", solid.Colour.ToCSSString(false));
                    path.SetAttribute("stroke-opacity", solid.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
                }
                else if (StrokeStyle is LinearGradientBrush linearGradient)
                {
                    path.SetAttribute("stroke", "url(#" + gradientName + ")");
                }
                else if (StrokeStyle is RadialGradientBrush radialGradient)
                {
                    path.SetAttribute("stroke", "url(#" + gradientName + ")");
                }

                path.SetAttribute("stroke-width", LineWidth.ToString(System.Globalization.CultureInfo.InvariantCulture));

                switch (LineCap)
                {
                    case LineCaps.Butt:
                        path.SetAttribute("stroke-linecap", "butt");
                        break;
                    case LineCaps.Round:
                        path.SetAttribute("stroke-linecap", "round");
                        break;
                    case LineCaps.Square:
                        path.SetAttribute("stroke-linecap", "square");
                        break;
```

```
                }

                switch (LineJoin)
                {
                    case LineJoins.Bevel:
                        path.SetAttribute("stroke-linejoin", "bevel");
                        break;
                    case LineJoins.Round:
                        path.SetAttribute("stroke-linejoin", "round");
                        break;
                    case LineJoins.Miter:
                        path.SetAttribute("stroke-linejoin", "miter");
                        break;
                }

                if (_lineDash.Phase != 0 || _lineDash.UnitsOn != 0 || _lineDash.UnitsOff != 0)
                {
                    path.SetAttribute("stroke-dasharray", _lineDash.UnitsOn.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + _lineDash.UnitsOff.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    path.SetAttribute("stroke-dashoffset", _lineDash.Phase.ToString(System.Globalization.CultureInfo.InvariantCulture));
                }

                path.SetAttribute("fill", "none");
                path.SetAttribute("transform", "matrix(" + _transform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");

            }
            else
            {
                Dictionary<string, string> style = new Dictionary<string, string>();

                if (StrokeStyle is SolidColourBrush solid)
                {
                    style.Add("stroke", solid.Colour.ToCSSString(false));
                    style.Add("stroke-opacity", solid.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
                }
                else if (StrokeStyle is LinearGradientBrush linearGradient)
                {
                    style.Add("stroke", "url(#" + gradientName + ")");
                }
                else if (StrokeStyle is RadialGradientBrush radialGradient)
                {
                    style.Add("stroke", "url(#" + gradientName + ")");
                }

                style.Add("stroke-width", LineWidth.ToString(System.Globalization.CultureInfo.InvariantCulture) + "px");

                switch (LineCap)
                {
                    case LineCaps.Butt:
                        style.Add("stroke-linecap", "butt");
                        break;
                    case LineCaps.Round:
                        style.Add("stroke-linecap", "round");
                        break;
                    case LineCaps.Square:
                        style.Add("stroke-linecap", "square");
                        break;
                }

                switch (LineJoin)
                {
                    case LineJoins.Bevel:
                        style.Add("stroke-linejoin", "bevel");
                        break;
                    case LineJoins.Round:
                        style.Add("stroke-linejoin", "round");
                        break;
                    case LineJoins.Miter:
                        style.Add("stroke-linejoin", "miter");
                        break;
                }

                if (_lineDash.Phase != 0 || _lineDash.UnitsOn != 0 || _lineDash.UnitsOff != 0)
                {
                    style.Add("stroke-dasharray", _lineDash.UnitsOn.ToString(System.Globalization.CultureInfo.InvariantCulture) + "px " + _lineDash.UnitsOff.ToString(System.Globalization.CultureInfo.InvariantCulture) + "px");
```

```
                    style.Add("stroke-dashoffset", _lineDash.Phase.ToString(System.Globalization.CultureInfo.InvariantCulture) + "px");
                }

                style.Add("fill", "none");
                style.Add("transform", "matrix(" + _transform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");

                string className = GetClass(style);
                path.SetAttribute("class", className);

            }

            if (!string.IsNullOrEmpty(Tag))
            {
                path.SetAttribute("id", this.TagPrefix + Tag);
            }

            if (!string.IsNullOrEmpty(this.Tag) && this.linkDestinations.TryGetValue(this.Tag, out string destination) && !string.IsNullOrEmpty(destination))
            {
                XmlElement aElement = Document.CreateElement("a", SVGNamespace);
                aElement.SetAttribute("href", destination);
                currentElement.AppendChild(aElement);
                currentElement = aElement;
            }

            currentElement.AppendChild(path);

            currentElement = currElement;

            currentPath = new SVGPathObject();
            currentFigure = new SVGFigure();
        }

        public void StrokeText(string text, double x, double y)
        {
            if (!TextToPaths)
            {
                if (!UsedFontFamilies.ContainsKey(Font.FontFamily.FileName))
                {
                    UsedFontFamilies.Add(Font.FontFamily.FileName, Font.FontFamily);
                    UsedChars.Add(Font.FontFamily.FileName, new HashSet<char>());
                }

                UsedChars[Font.FontFamily.FileName].UnionWith(text);

                Font.DetailedFontMetrics metrics = Font.MeasureTextAdvanced(text);

                double[,] currTransform = null;
                double[,] deltaTransform = MatrixUtils.Identity;

                switch (TextBaseline)
                {
                    case TextBaselines.Baseline:
                        currTransform = MatrixUtils.Translate(_transform, x - metrics.LeftSideBearing, y);
                        deltaTransform = MatrixUtils.Translate(deltaTransform, x - metrics.LeftSideBearing, y);
                        break;
                    case TextBaselines.Top:
                        currTransform = MatrixUtils.Translate(_transform, x - metrics.LeftSideBearing, y + metrics.Top);
                        deltaTransform = MatrixUtils.Translate(deltaTransform, x - metrics.LeftSideBearing, y + metrics.Top);
                        break;
                    case TextBaselines.Bottom:
                        currTransform = MatrixUtils.Translate(_transform, x - metrics.LeftSideBearing, y + metrics.Bottom);
                        deltaTransform = MatrixUtils.Translate(deltaTransform, x - metrics.LeftSideBearing, y + metrics.Bottom);
                        break;
                    case TextBaselines.Middle:
                        currTransform = MatrixUtils.Translate(_transform, x - metrics.LeftSideBearing, y + (metrics.Top + metrics.Bottom) * 0.5);
                        deltaTransform = MatrixUtils.Translate(deltaTransform, x - metrics.LeftSideBearing, y + (metrics.Top + metrics.Bottom) * 0.5);
                        break;
                    default:
                        currTransform = MatrixUtils.Translate(_transform, x - metrics.LeftSideBearing, y);
                        deltaTransform = MatrixUtils.Translate(deltaTransform, x -
```

```
metrics.LeftSideBearing, y);
                        break;
                }

                XmlElement currElement = currentElement;

                if (!string.IsNullOrEmpty(_currClipPath))
                {
                    currentElement = Document.CreateElement("g", SVGNamespace);
                    currentElement.SetAttribute("clip-path", _currClipPath);
                    currElement.AppendChild(currentElement);
                }

                XmlElement textElement = Document.CreateElement("text", SVGNamespace);

                string gradientName = null;

                {
                    if (StrokeStyle is LinearGradientBrush linearGradient)
                    {
                        if (!string.IsNullOrEmpty(Tag))
                        {
                            gradientName = this.TagPrefix + "textGradient" + (gradientCount +
1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Tag;
                        }
                        else
                        {
                            gradientName = this.TagPrefix + "textGradient" + (gradientCount +
1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Guid.NewGuid().ToString("N");
                        }

                        XmlElement gradientElement = linearGradient.ToLinearGradient(Document,
gradientName);

                        deltaTransform = MatrixUtils.Invert(deltaTransform);

                        gradientElement.SetAttribute("gradientTransform", "matrix(" +
deltaTransform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," +
deltaTransform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + deltaTransform[0,
1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + deltaTransform[1,
1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + deltaTransform[0,
2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + deltaTransform[1,
2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");

                        this.definitions.AppendChild(gradientElement);

                        gradientCount++;
                    }
                    else if (StrokeStyle is RadialGradientBrush radialGradient)
                    {
                        if (!string.IsNullOrEmpty(Tag))
                        {
                            gradientName = this.TagPrefix + "textGradient" + (gradientCount +
1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Tag;
                        }
                        else
                        {
                            gradientName = this.TagPrefix + "textGradient" + (gradientCount +
1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Guid.NewGuid().ToString("N");
                        }

                        XmlElement gradientElement = radialGradient.ToRadialGradient(Document,
gradientName);

                        deltaTransform = MatrixUtils.Invert(deltaTransform);

                        gradientElement.SetAttribute("gradientTransform", "matrix(" +
deltaTransform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," +
deltaTransform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + deltaTransform[0,
1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + deltaTransform[1,
1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + deltaTransform[0,
2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + deltaTransform[1,
2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");

                        this.definitions.AppendChild(gradientElement);

                        gradientCount++;
                    }
                }

                textElement.SetAttribute("x", "0");
                textElement.SetAttribute("y", "0");

```

```
                textElement.SetAttribute("transform", "matrix(" + currTransform[0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + currTransform[1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + currTransform[0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + currTransform[1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                        "," + currTransform[0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + currTransform[1].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");

                if (!UseStyles)
                {
                    if (StrokeStyle is SolidColourBrush solid)
                    {
                        textElement.SetAttribute("stroke", solid.Colour.ToCSSString(false));
                        textElement.SetAttribute("stroke-opacity", solid.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    }
                    else if (StrokeStyle is LinearGradientBrush linearGradient)
                    {
                        textElement.SetAttribute("stroke", "url(#" + gradientName + ")");
                    }
                    else if (StrokeStyle is RadialGradientBrush radialGradient)
                    {
                        textElement.SetAttribute("stroke", "url(#" + gradientName + ")");
                    }

                    textElement.SetAttribute("stroke-width", LineWidth.ToString(System.Globalization.CultureInfo.InvariantCulture));

                    switch (LineCap)
                    {
                        case LineCaps.Butt:
                            textElement.SetAttribute("stroke-linecap", "butt");
                            break;
                        case LineCaps.Round:
                            textElement.SetAttribute("stroke-linecap", "round");
                            break;
                        case LineCaps.Square:
                            textElement.SetAttribute("stroke-linecap", "square");
                            break;
                    }

                    switch (LineJoin)
                    {
                        case LineJoins.Bevel:
                            textElement.SetAttribute("stroke-linejoin", "bevel");
                            break;
                        case LineJoins.Round:
                            textElement.SetAttribute("stroke-linejoin", "round");
                            break;
                        case LineJoins.Miter:
                            textElement.SetAttribute("stroke-linejoin", "miter");
                            break;
                    }

                    if (_lineDash.Phase != 0 || _lineDash.UnitsOn != 0 || _lineDash.UnitsOff != 0)
                    {
                        textElement.SetAttribute("stroke-dasharray", _lineDash.UnitsOn.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + _lineDash.UnitsOff.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        textElement.SetAttribute("stroke-dashoffset", _lineDash.Phase.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    }
                    textElement.SetAttribute("fill", "none");

                    textElement.SetAttribute("font-size", Font.FontSize.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    textElement.SetAttribute("font-family", Font.FontFamily.FileName);

                    if (Font.FontFamily.IsBold)
                    {
                        textElement.SetAttribute("font-weight", "bold");
                    }
                    else
                    {
                        textElement.SetAttribute("font-weight", "regular");
                    }

                    if (Font.FontFamily.IsItalic)
                    {
                        textElement.SetAttribute("font-style", "italic");
                    }
                    else
                    {
                        textElement.SetAttribute("font-style", "normal");
                    }
```

```
                    if (Font.FontFamily.IsOblique)
                    {
                        textElement.SetAttribute("font-style", "oblique");
                    }
                }
                else
                {
                    Dictionary<string, string> style = new Dictionary<string, string>();

                    if (StrokeStyle is SolidColourBrush solid)
                    {
                        style.Add("stroke", solid.Colour.ToCSSString(false));
                        style.Add("stroke-opacity", solid.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    }
                    else if (StrokeStyle is LinearGradientBrush linearGradient)
                    {
                        style.Add("stroke", "url(#" + gradientName + ")");
                    }
                    else if (StrokeStyle is RadialGradientBrush radialGradient)
                    {
                        style.Add("stroke", "url(#" + gradientName + ")");
                    }

                    style.Add("stroke-width", LineWidth.ToString(System.Globalization.CultureInfo.InvariantCulture) + "px");

                    switch (LineCap)
                    {
                        case LineCaps.Butt:
                            style.Add("stroke-linecap", "butt");
                            break;
                        case LineCaps.Round:
                            style.Add("stroke-linecap", "round");
                            break;
                        case LineCaps.Square:
                            style.Add("stroke-linecap", "square");
                            break;
                    }

                    switch (LineJoin)
                    {
                        case LineJoins.Bevel:
                            style.Add("stroke-linejoin", "bevel");
                            break;
                        case LineJoins.Round:
                            style.Add("stroke-linejoin", "round");
                            break;
                        case LineJoins.Miter:
                            style.Add("stroke-linejoin", "miter");
                            break;
                    }

                    if (_lineDash.Phase != 0 || _lineDash.UnitsOn != 0 || _lineDash.UnitsOff != 0)
                    {
                        style.Add("stroke-dasharray", _lineDash.UnitsOn.ToString(System.Globalization.CultureInfo.InvariantCulture) + "px " + _lineDash.UnitsOff.ToString(System.Globalization.CultureInfo.InvariantCulture) + "px");
                        style.Add("stroke-dashoffset", _lineDash.Phase.ToString(System.Globalization.CultureInfo.InvariantCulture) + "px");
                    }
                    style.Add("fill", "none");

                    style.Add("font-size", Font.FontSize.ToString(System.Globalization.CultureInfo.InvariantCulture) + "px");
                    style.Add("font-family", Font.FontFamily.FileName);

                    if (Font.FontFamily.IsBold)
                    {
                        style.Add("font-weight", "bold");
                    }
                    else
                    {
                        style.Add("font-weight", "regular");
                    }

                    if (Font.FontFamily.IsItalic)
                    {
                        style["font-style"] = "italic";
                    }
                    else
                    {
                        style["font-style"] = "normal";
                    }

                    if (Font.FontFamily.IsOblique)
```

```
                    {
                        style["font-style"] = "oblique";
                    }

                    string className = GetClass(style);

                    textElement.SetAttribute("class", className);
                }

                ProcessText(text, textElement);

                if (!string.IsNullOrEmpty(Tag))
                {
                    textElement.SetAttribute("id", this.TagPrefix + Tag);
                }

                if (!string.IsNullOrEmpty(this.Tag) && this.linkDestinations.TryGetValue(this.Tag, out string destination) && !string.IsNullOrEmpty(destination))
                {
                    XmlElement aElement = Document.CreateElement("a", SVGNamespace);
                    aElement.SetAttribute("href", destination);
                    currentElement.AppendChild(aElement);
                    currentElement = aElement;
                }

                currentElement.AppendChild(textElement);

                currentElement = currElement;
            }
            else
            {
                PathText(text, x, y);
                Stroke();
            }
        }

        public void Transform(double a, double b, double c, double d, double e, double f)
        {
            double[,] transfMatrix = new double[3, 3] { { a, c, e }, { b, d, f }, { 0, 0, 1 } };
            _transform = MatrixUtils.Multiply(_transform, transfMatrix);

            currentPath = new SVGPathObject();
            currentFigure = new SVGFigure();
        }

        public void Translate(double x, double y)
        {
            _transform = MatrixUtils.Translate(_transform, x, y);

            currentPath = new SVGPathObject();
            currentFigure = new SVGFigure();
        }

        public void SetClippingPath()
        {
            if (currentFigure.PointCount > 0)
            {
                currentPath.Figures.Add(currentFigure);
            }

            XmlElement clipPath = Document.CreateElement("clipPath", SVGNamespace);
            string id = !string.IsNullOrEmpty(Tag) ? (this.TagPrefix + Tag) : Guid.NewGuid().ToString("N");
            clipPath.SetAttribute("id", id);

            if (!string.IsNullOrEmpty(_currClipPath))
            {
                clipPath.SetAttribute("clip-path", _currClipPath);
            }

            XmlElement path = Document.CreateElement("path", SVGNamespace);
            path.SetAttribute("id", id + "_clipPath");
            path.SetAttribute("d", currentPath.Figures.Aggregate("", (a, b) => a + b.Data));

            if (StrokeStyle is SolidColourBrush solid)
            {
                path.SetAttribute("stroke", solid.Colour.ToCSSString(false));
                path.SetAttribute("stroke-opacity", solid.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
            }
            else if (StrokeStyle is LinearGradientBrush linearGradient)
            {
                string gradientName;

                if (!ReuseGradients || !gradients.TryGetValue(linearGradient, out gradientName))
                {
                    if (!string.IsNullOrEmpty(Tag))
```

```
                    {
                        gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Tag;
                    }
                    else
                    {
                        gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Guid.NewGuid().ToString("N");
                    }

                    XmlElement gradientElement = linearGradient.ToLinearGradient(Document, gradientName);
                    this.definitions.AppendChild(gradientElement);

                    if (ReuseGradients)
                    {
                        gradients.Add(linearGradient, gradientName);
                    }

                    gradientCount++;
                }

                path.SetAttribute("stroke", "url(#" + gradientName + ")");
            }
            else if (StrokeStyle is RadialGradientBrush radialGradient)
            {
                string gradientName;

                if (!ReuseGradients || !gradients.TryGetValue(radialGradient, out gradientName))
                {
                    if (!string.IsNullOrEmpty(Tag))
                    {
                        gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Tag;
                    }
                    else
                    {
                        gradientName = this.TagPrefix + "gradient" + (gradientCount + 1).ToString(System.Globalization.CultureInfo.InvariantCulture) + "_" + Guid.NewGuid().ToString("N");
                    }

                    XmlElement gradientElement = radialGradient.ToRadialGradient(Document, gradientName);
                    this.definitions.AppendChild(gradientElement);

                    if (ReuseGradients)
                    {
                        gradients.Add(radialGradient, gradientName);
                    }

                    gradientCount++;
                }

                path.SetAttribute("stroke", "url(#" + gradientName + ")");
            }

            path.SetAttribute("stroke-width", LineWidth.ToString(System.Globalization.CultureInfo.InvariantCulture));

            switch (LineCap)
            {
                case LineCaps.Butt:
                    path.SetAttribute("stroke-linecap", "butt");
                    break;
                case LineCaps.Round:
                    path.SetAttribute("stroke-linecap", "round");
                    break;
                case LineCaps.Square:
                    path.SetAttribute("stroke-linecap", "square");
                    break;
            }

            switch (LineJoin)
            {
                case LineJoins.Bevel:
                    path.SetAttribute("stroke-linejoin", "bevel");
                    break;
                case LineJoins.Round:
                    path.SetAttribute("stroke-linejoin", "round");
                    break;
                case LineJoins.Miter:
                    path.SetAttribute("stroke-linejoin", "miter");
                    break;
            }

            if (_lineDash.Phase != 0 || _lineDash.UnitsOn != 0 || _lineDash.UnitsOff != 0)
            {
```

```
                path.SetAttribute("stroke-dasharray", _lineDash.UnitsOn.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + _lineDash.UnitsOff.ToString(System.Globalization.CultureInfo.InvariantCulture));
                path.SetAttribute("stroke-dashoffset", _lineDash.Phase.ToString(System.Globalization.CultureInfo.InvariantCulture));
            }

            path.SetAttribute("fill", "none");
            path.SetAttribute("transform", "matrix(" + _transform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                "," + _transform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                "," + _transform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");

            clipPath.AppendChild(path);

            currentElement.AppendChild(clipPath);

            _currClipPath = "url(#" + id + ")";

            currentPath = new SVGPathObject();
            currentFigure = new SVGFigure();
        }

        public void DrawRasterImage(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image)
        {
            Save();

            MoveTo(destinationX, destinationY);
            LineTo(destinationX + destinationWidth, destinationY);
            LineTo(destinationX + destinationWidth, destinationY + destinationHeight);
            LineTo(destinationX, destinationY + destinationHeight);
            Close();
            SetClippingPath();

            double sourceRectX = (double)sourceX / image.Width;
            double sourceRectY = (double)sourceY / image.Height;
            double sourceRectWidth = (double)sourceWidth / image.Width;
            double sourceRectHeight = (double)sourceHeight / image.Height;

            double scaleX = destinationWidth / sourceRectWidth;
            double scaleY = destinationHeight / sourceRectHeight;

            double translationX = destinationX / scaleX - sourceRectX;
            double translationY = destinationY / scaleY - sourceRectY;

            Scale(scaleX, scaleY);
            Translate(translationX, translationY);

            XmlElement currElement = currentElement;

            if (!string.IsNullOrEmpty(_currClipPath))
            {
                currentElement = Document.CreateElement("g", SVGNamespace);
                currentElement.SetAttribute("clip-path", _currClipPath);
                currElement.AppendChild(currentElement);
            }

            XmlElement img = Document.CreateElement("image", SVGNamespace);
            img.SetAttribute("x", "0");
            img.SetAttribute("y", "0");

            img.SetAttribute("width", "1");
            img.SetAttribute("height", "1");

            img.SetAttribute("preserveAspectRatio", "none");

            if (image.Interpolate)
            {
                img.SetAttribute("image-rendering", "optimizeQuality");
            }
            else
            {
                img.SetAttribute("image-rendering", "pixelated");
            }

            img.SetAttribute("transform", "matrix(" + _transform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                "," + _transform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                "," + _transform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");
```

```

            if (!string.IsNullOrEmpty(Tag))
            {
                img.SetAttribute("id", this.TagPrefix + Tag);
            }

            img.SetAttribute("href", "http://www.w3.org/1999/xlink", "" + Convert.ToBase64String(image.PNGStream.ToArray()));

            if (!string.IsNullOrEmpty(this.Tag) && this.linkDestinations.TryGetValue(this.Tag, out string destination) && !string.IsNullOrEmpty(destination))
            {
                XmlElement aElement = Document.CreateElement("a", SVGNamespace);
                aElement.SetAttribute("href", destination);
                currentElement.AppendChild(aElement);
                currentElement = aElement;
            }

            currentElement.AppendChild(img);

            currentElement = currElement;

            Restore();
        }

        private void ProcessText(string text, XmlNode parent)
        {
            if (Font.EnableKerning && this.TextOption == SVGContextInterpreter.TextOptions.SubsetFonts)
            {
                List<(string, Point)> tSpans = new List<(string, Point)>();

                StringBuilder currentRun = new StringBuilder();
                Point currentKerning = new Point();

                Point currentGlyphPlacementDelta = new Point();
                Point currentGlyphAdvanceDelta = new Point();
                Point nextGlyphPlacementDelta = new Point();
                Point nextGlyphAdvanceDelta = new Point();

                for (int i = 0; i < text.Length; i++)
                {
                    if (i < text.Length - 1)
                    {
                        currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                        currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                        nextGlyphAdvanceDelta = new Point();
                        nextGlyphPlacementDelta = new Point();

                        TrueTypeFile.PairKerning kerning = Font.FontFamily.TrueTypeFile.Get1000EmKerning(text[i], text[i + 1]);

                        if (kerning != null)
                        {
                            currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                            currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                            nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                            nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                        }
                    }

                    if (currentGlyphPlacementDelta.X != 0 || currentGlyphPlacementDelta.Y != 0 || currentGlyphAdvanceDelta.X != 0 || currentGlyphAdvanceDelta.Y != 0)
                    {
                        if (currentRun.Length > 0)
                        {
                            tSpans.Add((currentRun.ToString(), currentKerning));

                            tSpans.Add((text[i].ToString(), new Point(currentGlyphPlacementDelta.X * Font.FontSize / 1000, currentGlyphPlacementDelta.Y * Font.FontSize / 1000)));

                            currentRun.Clear();
                            currentKerning = new Point((currentGlyphAdvanceDelta.X - currentGlyphPlacementDelta.X) * Font.FontSize / 1000, (currentGlyphAdvanceDelta.Y - currentGlyphPlacementDelta.Y) * Font.FontSize / 1000);
                        }
                        else
                        {
                            tSpans.Add((text[i].ToString(), new Point(currentGlyphPlacementDelta.X * Font.FontSize / 1000 + currentKerning.X, currentGlyphPlacementDelta.Y * Font.FontSize / 1000 + currentKerning.Y)));

```

```
                            currentRun.Clear();
                            currentKerning = new Point((currentGlyphAdvanceDelta.X - currentGlyphPlacementDelta.X) * Font.FontSize / 1000, (currentGlyphAdvanceDelta.Y - currentGlyphPlacementDelta.Y) * Font.FontSize / 1000);
                        }
                    }
                    else
                    {
                        currentRun.Append(text[i]);
                    }
                }

                if (currentRun.Length > 0)
                {
                    tSpans.Add((currentRun.ToString(), currentKerning));
                }

                for (int i = 0; i < tSpans.Count; i++)
                {
                    XmlElement tspanElement = Document.CreateElement("tspan", SVGNamespace);
                    tspanElement.InnerText = tSpans[i].Item1.Replace(" ", "\u00A0");

                    if (tSpans[i].Item2.X != 0)
                    {
                        tspanElement.SetAttribute("dx", tSpans[i].Item2.X.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    }

                    if (tSpans[i].Item2.Y != 0)
                    {
                        tspanElement.SetAttribute("dy", tSpans[i].Item2.Y.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    }

                    parent.AppendChild(tspanElement);
                }

            }
            else
            {
                parent.InnerText = text.Replace(" ", "\u00A0");
            }
        }

        public void DrawFilteredGraphics(Graphics graphics, IFilter filter)
        {
            if (FilterOption.Operation == SVGContextInterpreter.FilterOption.FilterOperations.IgnoreAll)
            {
                graphics.CopyToIGraphicsContext(this);
            }
            else if (FilterOption.Operation == SVGContextInterpreter.FilterOption.FilterOperations.SkipAll)
            {

            }
            else
            {
                bool rasterisationNeeded = false;
                bool justDraw = false;

                if (FilterOption.Operation == SVGContextInterpreter.FilterOption.FilterOperations.RasteriseAll)
                {
                    rasterisationNeeded = true;
                    justDraw = false;
                }
                else
                {
                    if (FilterOption.Operation == SVGContextInterpreter.FilterOption.FilterOperations.RasteriseIfNecessary)
                    {
                        justDraw = false;
                        rasterisationNeeded = true;
                    }
                    else if (FilterOption.Operation == SVGContextInterpreter.FilterOption.FilterOperations.NeverRasteriseAndIgnore)
                    {
                        rasterisationNeeded = false;
                        justDraw = true;
                    }
                    else if (FilterOption.Operation == SVGContextInterpreter.FilterOption.FilterOperations.NeverRasteriseAndSkip)
                    {
                        rasterisationNeeded = false;
                        justDraw = false;
                    }
```

```

                    if (filter is MaskFilter mask)
                    {
                        rasterisationNeeded = false;
                        justDraw = false;

                        XmlElement currElement = currentElement;

                        if (!string.IsNullOrEmpty(_currClipPath))
                        {
                            currentElement = Document.CreateElement("g", SVGNamespace);
                            currentElement.SetAttribute("clip-path", _currClipPath);
                            currElement.AppendChild(currentElement);
                        }

                        XmlElement currentElement2 = currentElement;

                        string filterGuid = !string.IsNullOrEmpty(Tag) ? (this.TagPrefix + Tag + "@filter") : Guid.NewGuid().ToString("N");

                        Rectangle bounds = graphics.GetBounds();

                        Point p1 = new Point(bounds.Location.X, bounds.Location.Y);
                        Point p2 = new Point(bounds.Location.X + bounds.Size.Width, bounds.Location.Y);
                        Point p3 = new Point(bounds.Location.X + bounds.Size.Width, bounds.Location.Y + bounds.Size.Height);
                        Point p4 = new Point(bounds.Location.X, bounds.Location.Y + bounds.Size.Height);

                        /*p1 = MatrixUtils.Multiply(_transform, p1);
                        p2 = MatrixUtils.Multiply(_transform, p2);
                        p3 = MatrixUtils.Multiply(_transform, p3);
                        p4 = MatrixUtils.Multiply(_transform, p4);*/

                        bounds = Point.Bounds(p1, p2, p3, p4);

                        XmlElement maskElement = Document.CreateElement("mask", SVGNamespace);
                        maskElement.SetAttribute("id", filterGuid);
                        maskElement.SetAttribute("maskUnits", "userSpaceOnUse");
                        maskElement.SetAttribute("x", bounds.Location.X.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        maskElement.SetAttribute("y", bounds.Location.Y.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        maskElement.SetAttribute("width", bounds.Size.Width.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        maskElement.SetAttribute("height", bounds.Size.Height.ToString(System.Globalization.CultureInfo.InvariantCulture));

                        this.definitions.AppendChild(maskElement);

                        currentElement = maskElement;

                        double[,] currTransform = _transform;
                        _transform = MatrixUtils.Identity;

                        mask.Mask.CopyToIGraphicsContext(this);

                        _transform = currTransform;

                        currentElement = Document.CreateElement("g", SVGNamespace);

                        if (!string.IsNullOrEmpty(this.Tag))
                        {
                            currentElement.SetAttribute("id", this.TagPrefix + Tag);
                        }

                        currentElement.SetAttribute("mask", "url(#" + filterGuid + ")");
                        currentElement2.AppendChild(currentElement);

                        currentElement.SetAttribute("transform", "matrix(" + _transform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");
                        currTransform = _transform;
                        _transform = MatrixUtils.Identity;

                        graphics.CopyToIGraphicsContext(this);

                        _transform = currTransform;

                        currentElement = currElement;
                    }
                    if (filter is GaussianBlurFilter gauss)
```

```
                    {
                        rasterisationNeeded = false;
                        justDraw = false;
                        XmlElement currElement = currentElement;

                        if (!string.IsNullOrEmpty(_currClipPath))
                        {
                            currentElement = Document.CreateElement("g", SVGNamespace);
                            currentElement.SetAttribute("clip-path", _currClipPath);
                            currElement.AppendChild(currentElement);
                        }

                        Rectangle bounds = graphics.GetBounds();

                        Point p1 = new Point(bounds.Location.X - gauss.StandardDeviation * 3, bounds.Location.Y - gauss.StandardDeviation * 3);
                        Point p2 = new Point(bounds.Location.X + bounds.Size.Width + gauss.StandardDeviation * 3, bounds.Location.Y - gauss.StandardDeviation * 3);
                        Point p3 = new Point(bounds.Location.X + bounds.Size.Width + gauss.StandardDeviation * 3, bounds.Location.Y + bounds.Size.Height + gauss.StandardDeviation * 3);
                        Point p4 = new Point(bounds.Location.X - gauss.StandardDeviation * 3, bounds.Location.Y + bounds.Size.Height + gauss.StandardDeviation * 3);

                        /*p1 = MatrixUtils.Multiply(_transform, p1);
                        p2 = MatrixUtils.Multiply(_transform, p2);
                        p3 = MatrixUtils.Multiply(_transform, p3);
                        p4 = MatrixUtils.Multiply(_transform, p4);*/

                        bounds = Point.Bounds(p1, p2, p3, p4);

                        string filterGuid = !string.IsNullOrEmpty(Tag) ? (this.TagPrefix + Tag + "@filter") : Guid.NewGuid().ToString("N");

                        XmlElement filterElement = Document.CreateElement("filter", SVGNamespace);
                        filterElement.SetAttribute("id", filterGuid);
                        filterElement.SetAttribute("color-interpolation-filters", "sRGB");
                        filterElement.SetAttribute("filterUnits", "userSpaceOnUse");
                        filterElement.SetAttribute("x", bounds.Location.X.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        filterElement.SetAttribute("y", bounds.Location.Y.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        filterElement.SetAttribute("width", bounds.Size.Width.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        filterElement.SetAttribute("height", bounds.Size.Height.ToString(System.Globalization.CultureInfo.InvariantCulture));

                        this.definitions.AppendChild(filterElement);

                        XmlElement feElement = Document.CreateElement("feGaussianBlur", SVGNamespace);
                        feElement.SetAttribute("stdDeviation", gauss.StandardDeviation.ToString(System.Globalization.CultureInfo.InvariantCulture));

                        if (!string.IsNullOrEmpty(Tag))
                        {
                            feElement.SetAttribute("id", this.TagPrefix + Tag + "@feGaussianBlur");
                        }

                        filterElement.AppendChild(feElement);

                        XmlElement currentElement2 = currentElement;

                        currentElement = Document.CreateElement("g", SVGNamespace);

                        if (!string.IsNullOrEmpty(this.Tag))
                        {
                            currentElement.SetAttribute("id", this.TagPrefix + Tag);
                        }

                        currentElement.SetAttribute("filter", "url(#" + filterGuid + ")");
                        currentElement2.AppendChild(currentElement);

                        currentElement.SetAttribute("transform", "matrix(" + _transform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");
                        double[,] currTransform = _transform;
                        _transform = MatrixUtils.Identity;

                        graphics.CopyToIGraphicsContext(this);

                        currentElement = currElement;
                    }
                    else if (filter is ColourMatrixFilter cmf)
                    {
```

```
                        rasterisationNeeded = false;
                        justDraw = false;
                        XmlElement currElement = currentElement;

                        if (!string.IsNullOrEmpty(_currClipPath))
                        {
                            currentElement = Document.CreateElement("g", SVGNamespace);
                            currentElement.SetAttribute("clip-path", _currClipPath);
                            currElement.AppendChild(currentElement);
                        }

                        Rectangle bounds = graphics.GetBounds();

                        Point p1 = new Point(bounds.Location.X, bounds.Location.Y);
                        Point p2 = new Point(bounds.Location.X + bounds.Size.Width, bounds.Location.Y);
                        Point p3 = new Point(bounds.Location.X + bounds.Size.Width, bounds.Location.Y + bounds.Size.Height);
                        Point p4 = new Point(bounds.Location.X, bounds.Location.Y + bounds.Size.Height);

                        /* p1 = MatrixUtils.Multiply(_transform, p1);
                         p2 = MatrixUtils.Multiply(_transform, p2);
                         p3 = MatrixUtils.Multiply(_transform, p3);
                         p4 = MatrixUtils.Multiply(_transform, p4);*/

                        bounds = Point.Bounds(p1, p2, p3, p4);

                        string filterGuid = !string.IsNullOrEmpty(Tag) ? (this.TagPrefix + Tag + "@filter") : Guid.NewGuid().ToString("N");

                        XmlElement filterElement = Document.CreateElement("filter", SVGNamespace);
                        filterElement.SetAttribute("id", filterGuid);
                        filterElement.SetAttribute("color-interpolation-filters", "sRGB");
                        filterElement.SetAttribute("filterUnits", "userSpaceOnUse");
                        filterElement.SetAttribute("x", bounds.Location.X.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        filterElement.SetAttribute("y", bounds.Location.Y.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        filterElement.SetAttribute("width", bounds.Size.Width.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        filterElement.SetAttribute("height", bounds.Size.Height.ToString(System.Globalization.CultureInfo.InvariantCulture));

                        this.definitions.AppendChild(filterElement);

                        XmlElement feElement = Document.CreateElement("feColorMatrix", SVGNamespace);
                        feElement.SetAttribute("type", "matrix");

                        if (!string.IsNullOrEmpty(Tag))
                        {
                            feElement.SetAttribute("id", this.TagPrefix + Tag + "@feColorMatrix");
                        }

                        StringBuilder matrix = new StringBuilder();

                        for (int i = 0; i < 4; i++)
                        {
                            for (int j = 0; j < 5; j++)
                            {
                                matrix.Append(cmf.ColourMatrix[i, j].ToString(System.Globalization.CultureInfo.InvariantCulture));
                                if (i != 3 || j != 4)
                                {
                                    matrix.Append(" ");
                                }
                            }
                        }

                        feElement.SetAttribute("values", matrix.ToString());
                        filterElement.AppendChild(feElement);

                        XmlElement currentElement2 = currentElement;

                        currentElement = Document.CreateElement("g", SVGNamespace);

                        if (!string.IsNullOrEmpty(this.Tag))
                        {
                            currentElement.SetAttribute("id", this.TagPrefix + Tag);
                        }

                        currentElement.SetAttribute("filter", "url(#" + filterGuid + ")");
                        currentElement2.AppendChild(currentElement);

                        currentElement.SetAttribute("transform", "matrix(" + _transform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
```

```
                    "," + _transform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                    "," + _transform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");
                        double[,] currTransform = _transform;
                        _transform = MatrixUtils.Identity;

                        graphics.CopyToIGraphicsContext(this);

                        currentElement = currElement;
                    }
                    else if (filter is CompositeLocationInvariantFilter comp)
                    {
                        bool allSupported = true;

                        foreach (IFilter filter2 in comp.Filters)
                        {
                            if (!(filter2 is GaussianBlurFilter) && !(filter2 is ColourMatrixFilter))
                            {
                                allSupported = false;
                                break;
                            }
                        }

                        if (allSupported)
                        {
                            rasterisationNeeded = false;
                            justDraw = false;

                            XmlElement currElement = currentElement;

                            if (!string.IsNullOrEmpty(_currClipPath))
                            {
                                currentElement = Document.CreateElement("g", SVGNamespace);
                                currentElement.SetAttribute("clip-path", _currClipPath);
                                currElement.AppendChild(currentElement);
                            }

                            Rectangle bounds = graphics.GetBounds();

                            Point p1 = new Point(bounds.Location.X - comp.TopLeftMargin.X, bounds.Location.Y - comp.TopLeftMargin.Y);
                            Point p2 = new Point(bounds.Location.X + bounds.Size.Width + comp.BottomRightMargin.X, bounds.Location.Y - comp.TopLeftMargin.Y);
                            Point p3 = new Point(bounds.Location.X + bounds.Size.Width + comp.BottomRightMargin.X, bounds.Location.Y + bounds.Size.Height + comp.BottomRightMargin.Y);
                            Point p4 = new Point(bounds.Location.X - comp.TopLeftMargin.X, bounds.Location.Y + bounds.Size.Height + comp.BottomRightMargin.Y);

                            /* p1 = MatrixUtils.Multiply(_transform, p1);
                             p2 = MatrixUtils.Multiply(_transform, p2);
                             p3 = MatrixUtils.Multiply(_transform, p3);
                             p4 = MatrixUtils.Multiply(_transform, p4);*/

                            bounds = Point.Bounds(p1, p2, p3, p4);

                            string filterGuid = !string.IsNullOrEmpty(Tag) ? (this.TagPrefix + Tag + "@filter") : Guid.NewGuid().ToString("N");

                            XmlElement filterElement = Document.CreateElement("filter", SVGNamespace);
                            filterElement.SetAttribute("id", filterGuid);
                            filterElement.SetAttribute("color-interpolation-filters", "sRGB");
                            filterElement.SetAttribute("filterUnits", "userSpaceOnUse");
                            filterElement.SetAttribute("x", bounds.Location.X.ToString(System.Globalization.CultureInfo.InvariantCulture));
                            filterElement.SetAttribute("y", bounds.Location.Y.ToString(System.Globalization.CultureInfo.InvariantCulture));
                            filterElement.SetAttribute("width", bounds.Size.Width.ToString(System.Globalization.CultureInfo.InvariantCulture));
                            filterElement.SetAttribute("height", bounds.Size.Height.ToString(System.Globalization.CultureInfo.InvariantCulture));

                            this.definitions.AppendChild(filterElement);

                            int index = 0;

                            foreach (IFilter filter2 in comp.Filters)
                            {
                                if (filter2 is GaussianBlurFilter gauss2)
                                {
                                    XmlElement feElement = Document.CreateElement("feGaussianBlur", SVGNamespace);
                                    feElement.SetAttribute("stdDeviation", gauss2.StandardDeviation.ToString(System.Globalization.CultureInfo.InvariantCulture));

                                    if (!string.IsNullOrEmpty(Tag))
                                    {
```

```
                                        filterElement.SetAttribute("id", this.TagPrefix + Tag + "@feGaussianBlur" + index.ToString());
                                    }

                                    filterElement.AppendChild(feElement);
                                }
                                else if (filter2 is ColourMatrixFilter cmf2)
                                {
                                    XmlElement feElement = Document.CreateElement("feColorMatrix", SVGNamespace);
                                    feElement.SetAttribute("type", "matrix");
                                    if (!string.IsNullOrEmpty(Tag))
                                    {
                                        filterElement.SetAttribute("id", this.TagPrefix + Tag + "@feColorMatrix" + index.ToString());
                                    }

                                    StringBuilder matrix = new StringBuilder();

                                    for (int i = 0; i < 4; i++)
                                    {
                                        for (int j = 0; j < 5; j++)
                                        {
                                            matrix.Append(cmf2.ColourMatrix[i, j].ToString(System.Globalization.CultureInfo.InvariantCulture));
                                            if (i != 3 || j != 4)
                                            {
                                                matrix.Append(" ");
                                            }
                                        }
                                    }

                                    feElement.SetAttribute("values", matrix.ToString());
                                    filterElement.AppendChild(feElement);
                                }

                                index++;
                            }

                            XmlElement currentElement2 = currentElement;

                            currentElement = Document.CreateElement("g", SVGNamespace);

                            if (!string.IsNullOrEmpty(this.Tag))
                            {
                                currentElement.SetAttribute("id", this.TagPrefix + Tag);
                            }

                            currentElement.SetAttribute("filter", "url(#" + filterGuid + ")");
                            currentElement2.AppendChild(currentElement);

                            currentElement.SetAttribute("transform", "matrix(" + _transform[0, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 0].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                                "," + _transform[0, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 1].ToString(System.Globalization.CultureInfo.InvariantCulture) +
                                "," + _transform[0, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + "," + _transform[1, 2].ToString(System.Globalization.CultureInfo.InvariantCulture) + ")");
                            double[,] currTransform = _transform;
                            _transform = MatrixUtils.Identity;

                            graphics.CopyToIGraphicsContext(this);

                            currentElement = currElement;


                        }
                    }
                }

                if (rasterisationNeeded)
                {
                    double scale = FilterOption.RasterisationResolution;

                    Rectangle bounds = graphics.GetBounds();

                    bounds = new Rectangle(bounds.Location.X - filter.TopLeftMargin.X, bounds.Location.Y - filter.TopLeftMargin.Y, bounds.Size.Width + filter.TopLeftMargin.X + filter.BottomRightMargin.X, bounds.Size.Height + filter.TopLeftMargin.Y + filter.BottomRightMargin.Y);

                    if (bounds.Size.Width > 0 && bounds.Size.Height > 0)
                    {
                        if (!FilterOption.RasterisationResolutionRelative)
                        {
                            scale = scale / Math.Min(bounds.Size.Width, bounds.Size.Height);
                        }
```

```
                        if (graphics.TryRasterise(bounds, scale, true, out RasterImage rasterised))
                        {
                            RasterImage filtered = null;

                            if (filter is ILocationInvariantFilter locInvFilter)
                            {
                                filtered = locInvFilter.Filter(rasterised, scale);
                            }
                            else if (filter is IFilterWithLocation filterWithLoc)
                            {
                                filtered = filterWithLoc.Filter(rasterised, bounds, scale);
                            }

                            if (filtered != null)
                            {
                                rasterised.Dispose();

                                DrawRasterImage(0, 0, filtered.Width, filtered.Height, bounds.Location.X, bounds.Location.Y, bounds.Size.Width, bounds.Size.Height, filtered);
                            }
                        }
                        else
                        {
                            throw new NotImplementedException(@"The filter could not be rasterised! You can avoid this error by doing one of the following:
• Add a reference to VectSharp.Raster or VectSharp.Raster.ImageSharp (you may also need to add a using directive somewhere to force the assembly to be loaded).
• Provide your own implementation of Graphics.RasterisationMethod.
• Set the FilterOption.Operation to ""NeverRasteriseAndIgnore"", ""NeverRasteriseAndSkip"", ""IgnoreAll"" or ""SkipAll"".");
                        }
                    }
                }

                if (justDraw)
                {
                    graphics.CopyToIGraphicsContext(this);
                }
            }
        }
    }


    /// <summary>
    /// Contains methods to render a <see cref="Page"/> as an SVG file.
    /// </summary>
    public static class SVGContextInterpreter
    {

        /// <summary>
        /// Render the page to an SVG file.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="fileName">The full path to the file to save. If it exists, it will be overwritten.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the file.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <param name="useStyles">If this is <see langword="false"/>, presentation attributes are set as attributes on SVG elements. If this is <see langword="true"/>, CSS classes are used to set presentation attributes.</param>
        public static void SaveAsSVG(this Page page, string fileName, TextOptions textOption = TextOptions.SubsetFonts, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default, bool useStyles = false)
        {
            using (FileStream sr = new FileStream(fileName, FileMode.Create))
            {
                page.SaveAsSVG(sr, textOption, linkDestinations, filterOption, useStyles);
            }
        }

        /// <summary>
        /// Defines whether the used fonts should be included in the file.
        /// </summary>
        public enum TextOptions
        {
            /// <summary>
            /// Embeds the full font files.
            /// </summary>
            EmbedFonts,
```

```
            /// <summary>
            /// Embeds subsetted font files containing only the glyphs for the characters that have been used.
            /// </summary>
            SubsetFonts,

            /// <summary>
            /// Does not embed any font file and converts all text items into paths.
            /// </summary>
            ConvertIntoPaths,

            /// <summary>
            /// Does not embed any font file, but still encodes text items as such.
            /// </summary>
            DoNotEmbed
        }

        /// <summary>
        /// Determines how and whether image filters are rasterised.
        /// </summary>
        public class FilterOption
        {
            /// <summary>
            /// Defines whether image filters should be rasterised or not.
            /// </summary>
            public enum FilterOperations
            {
                /// <summary>
                /// Image filters will always be rasterised.
                /// </summary>
                RasteriseAll,

                /// <summary>
                /// Image filters will only be rasterised if they are not supported natively by the output file format.
                /// </summary>
                RasteriseIfNecessary,

                /// <summary>
                /// Image filters will never be rasterised; for filters that are not supported, the filter will be ignored.
                /// </summary>
                NeverRasteriseAndIgnore,

                /// <summary>
                /// Image filters will never be rasterised; if an image should be drawn with an unsupported filter, the image will not be drawn at all.
                /// </summary>
                NeverRasteriseAndSkip,

                /// <summary>
                /// All image filters (supported and unsupported) will be ignored.
                /// </summary>
                IgnoreAll,

                /// <summary>
                /// All the images that should be drawn with a filter will be ignored.
                /// </summary>
                SkipAll
            }

            /// <summary>
            /// Defines whether image filters should be rasterised or not.
            /// </summary>
            public FilterOperations Operation { get; } = FilterOperations.RasteriseIfNecessary;

            /// <summary>
            /// The resolution that will be used to rasterise image filters. Depending on the value of <see cref="RasterisationResolutionRelative"/>, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.
            /// </summary>
            public double RasterisationResolution { get; } = 1;

            /// <summary>
            /// Determines whether the value of <see cref="RasterisationResolution"/> is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).
            /// </summary>
            public bool RasterisationResolutionRelative { get; } = true;

            /// <summary>
            /// The default options for image filter rasterisation.
            /// </summary>
            public static FilterOption Default = new FilterOption(FilterOperations.RasteriseIfNecessary, 1, true);

```

```
            /// <summary>
            /// Create a new <see cref="FilterOption"/> object.
            /// </summary>
            /// <param name="operation">Defines whether image filters should be rasterised or not.</param>
            /// <param name="rasterisationResolution">The resolution that will be used to rasterise image filters. Depending on the value of <see cref="RasterisationResolutionRelative"/>, this can either be an absolute resolution (i.e. a size in pixel), or a scale factor that is applied to the image size in graphics units.</param>
            /// <param name="rasterisationResolutionRelative">Determines whether the value of <see cref="RasterisationResolution"/> is absolute (i.e. a size in pixel), or relative (i.e. a scale factor that is applied to the image size in graphics units).</param>
            public FilterOption(FilterOperations operation, double rasterisationResolution, bool rasterisationResolutionRelative)
            {
                this.Operation = operation;
                this.RasterisationResolution = rasterisationResolution;
                this.RasterisationResolutionRelative = rasterisationResolutionRelative;
            }
        }

        /// <summary>
        /// Render the page to an SVG stream.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="stream">The stream to which the SVG data will be written.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the file.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <param name="useStyles">If this is <see langword="false"/>, presentation attributes are set as attributes on SVG elements. If this is <see langword="true"/>, CSS classes are used to set presentation attributes.</param>
        public static void SaveAsSVG(this Page page, Stream stream, TextOptions textOption = TextOptions.SubsetFonts, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default, bool useStyles = false)
        {
            XmlDocument doc = page.SaveAsSVG(textOption, linkDestinations, filterOption, useStyles);
            WriteXMLToStream(doc.DocumentElement, stream);
        }

        /// <summary>
        /// Render the page to an SVG document.
        /// </summary>
        /// <param name="page">The <see cref="Page"/> to render.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the file.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <param name="useStyles">If this is <see langword="false"/>, presentation attributes are set as attributes on SVG elements. If this is <see langword="true"/>, CSS classes are used to set presentation attributes.</param>
        /// <returns>An <see cref="XmlDocument"/> containing the rendered SVG image.</returns>
        public static XmlDocument SaveAsSVG(this Page page, TextOptions textOption = TextOptions.SubsetFonts, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default, bool useStyles = false)
        {
            return CreateSVGDocument(page, "", textOption, linkDestinations, filterOption, useStyles, true);
        }

        private static XmlDocument CreateSVGDocument(this Page page, string tagPrefix, TextOptions textOption, Dictionary<string, string> linkDestinations, FilterOption filterOption, bool useStyles, bool reuseGradients)
        {
            if (linkDestinations == null)
            {
                linkDestinations = new Dictionary<string, string>();
            }

            if (filterOption == null)
            {
                filterOption = FilterOption.Default;
            }

            bool textToPaths = textOption == TextOptions.ConvertIntoPaths;

```

```
            SVGContext ctx = new SVGContext(page.Width, page.Height, textToPaths, textOption, linkDestinations, filterOption, useStyles, tagPrefix, reuseGradients);

            ctx.Rectangle(0, 0, page.Width, page.Height);
            ctx.SetFillStyle(page.Background);
            ctx.Fill();
            ctx.SetFillStyle((0, 0, 0, 1));

            page.Graphics.CopyToIGraphicsContext(ctx);

            if (!textToPaths && textOption != TextOptions.DoNotEmbed)
            {
                bool subsetFonts = textOption == TextOptions.SubsetFonts;

                StringBuilder cssFonts = new StringBuilder();

                Dictionary<string, string> newFontFamilies = new Dictionary<string, string>();

                foreach (KeyValuePair<string, FontFamily> kvp in ctx.UsedFontFamilies)
                {
                    TrueTypeFile subsettedFont;

                    if (subsetFonts)
                    {
                        newFontFamilies[kvp.Key] = kvp.Value.TrueTypeFile.GetFontFamilyName() + "-" + Guid.NewGuid().ToString();
                        subsettedFont = kvp.Value.TrueTypeFile.SubsetFont(new string(ctx.UsedChars[kvp.Key].ToArray()));
                    }
                    else
                    {
                        newFontFamilies[kvp.Key] = kvp.Value.TrueTypeFile.GetFontName();
                        subsettedFont = kvp.Value.TrueTypeFile;
                    }

                    byte[] fontBytes;

                    using (MemoryStream fontStream = new MemoryStream((int)subsettedFont.FontStream.Length))
                    {
                        subsettedFont.FontStream.Seek(0, SeekOrigin.Begin);
                        subsettedFont.FontStream.CopyTo(fontStream);

                        fontBytes = fontStream.ToArray();
                    }


                    cssFonts.Append("\n    @font-face\n    {\n      font-family: \"" + newFontFamilies[kvp.Key] + "\";\n      src: url(\"data:font/ttf;charset=utf-8base64,");
                    cssFonts.Append(Convert.ToBase64String(fontBytes));
                    cssFonts.Append("\");\n    }\n  ");
                }

                XmlElement style = ctx.Document.CreateElement("style", SVGContext.SVGNamespace);
                style.InnerText = cssFonts.ToString();

                XmlNode svgElement = ctx.Document.GetElementsByTagName("svg")[0];

                svgElement.InsertBefore(style, svgElement.FirstChild);

                HashSet<string> updatedClasses = new HashSet<string>();

                foreach (XmlNode text in ctx.Document.GetElementsByTagName("text"))
                {
                    string fontFamily;

                    if (!useStyles)
                    {
                        fontFamily = text.Attributes["font-family"].Value;
                    }
                    else
                    {
                        fontFamily = ctx.Styles[text.Attributes["class"].Value]["font-family"];
                    }

                    if (!useStyles || updatedClasses.Add(text.Attributes["class"].Value))
                    {

                        string fallbackFontFamily = "";

                        switch (fontFamily)
                        {
                            case "Helvetica":
                            case "Helvetica-Bold":
                            case "Helvetica-Oblique":
                            case "Helvetica-BoldOblique":
                                fallbackFontFamily = "sans-serif";
```

```
                                break;

                            case "Times-Roman":
                            case "Times-Bold":
                            case "Times-Italic":
                            case "Times-BoldItalic":
                                fallbackFontFamily = "serif";
                                break;

                            case "Courier":
                            case "Courier-Bold":
                            case "Courier-Oblique":
                            case "Courier-BoldOblique":
                                fallbackFontFamily = "monospace";
                                break;

                            default:
                                if (ctx.UsedFontFamilies[fontFamily].TrueTypeFile.IsFixedPitch())
                                {
                                    fallbackFontFamily = "monospace";
                                }
                                else if (ctx.UsedFontFamilies[fontFamily].TrueTypeFile.IsScript() ||
ctx.UsedFontFamilies[fontFamily].TrueTypeFile.IsSerif())
                                {
                                    fallbackFontFamily = "serif";
                                }
                                else
                                {
                                    fallbackFontFamily = "sans-serif";
                                }
                                break;
                        }

                        if (!useStyles)
                        {
                            if (!string.IsNullOrEmpty(fallbackFontFamily))
                            {
                                ((XmlElement)text).SetAttribute("font-family",
newFontFamilies[fontFamily] + ", " + fallbackFontFamily);
                            }
                            else
                            {
                                ((XmlElement)text).SetAttribute("font-family",
newFontFamilies[fontFamily]);
                            }
                        }
                        else
                        {
                            if (!string.IsNullOrEmpty(fallbackFontFamily))
                            {
                                ctx.Styles[text.Attributes["class"].Value]["font-family"] =
newFontFamilies[fontFamily] + ", " + fallbackFontFamily;
                            }
                            else
                            {
                                ctx.Styles[text.Attributes["class"].Value]["font-family"] =
newFontFamilies[fontFamily];
                            }
                        }
                    }
                }
            }
            else if (!textToPaths && textOption == TextOptions.DoNotEmbed)
            {
                HashSet<string> updatedClasses = new HashSet<string>();

                foreach (XmlNode text in ctx.Document.GetElementsByTagName("text"))
                {
                    string fontFamily;

                    if (!useStyles)
                    {
                        fontFamily = text.Attributes["font-family"].Value;
                    }
                    else
                    {
                        fontFamily = ctx.Styles[text.Attributes["class"].Value]["font-family"];
                    }

                    if (!useStyles || updatedClasses.Add(text.Attributes["class"].Value))
                    {

                        string newFontFamily =
ctx.UsedFontFamilies[fontFamily].TrueTypeFile.GetFontFamilyName();

                        switch (fontFamily)
                        {
```

```
                            case "Helvetica":
                            case "Helvetica-Bold":
                            case "Helvetica-Oblique":
                            case "Helvetica-BoldOblique":
                                newFontFamily = newFontFamily + ", sans-serif";
                                break;

                            case "Times-Roman":
                            case "Times-Bold":
                            case "Times-Italic":
                            case "Times-BoldItalic":
                                newFontFamily = newFontFamily + ", serif";
                                break;

                            case "Courier":
                            case "Courier-Bold":
                            case "Courier-Oblique":
                            case "Courier-BoldOblique":
                                newFontFamily = newFontFamily + ", monospace";
                                break;

                            default:
                                if (ctx.UsedFontFamilies[fontFamily].TrueTypeFile.IsFixedPitch())
                                {
                                    newFontFamily = newFontFamily + ", monospace";
                                }
                                else if (ctx.UsedFontFamilies[fontFamily].TrueTypeFile.IsScript())
                                {
                                    newFontFamily = newFontFamily + ", cursive";
                                }
                                else if (ctx.UsedFontFamilies[fontFamily].TrueTypeFile.IsSerif())
                                {
                                    newFontFamily = newFontFamily + ", serif";
                                }
                                else
                                {
                                    newFontFamily = newFontFamily + ", sans-serif";
                                }
                                break;
                        }

                        if (!useStyles)
                        {
                            ((XmlElement)text).SetAttribute("font-family", newFontFamily);
                        }
                        else
                        {
                            ctx.Styles[text.Attributes["class"].Value]["font-family"] = newFontFamily;
                        }
                    }
                }
            }

            if (useStyles)
            {
                StringBuilder classStyle = new StringBuilder();

                if (ctx.Styles.Count > 0)
                {
                    classStyle.Append("\n  ");
                }

                foreach (KeyValuePair<string, Dictionary<string, string» style in ctx.Styles)
                {
                    classStyle.Append("  .");
                    classStyle.Append(style.Key);
                    classStyle.Append("\n");
                    classStyle.Append("    {\n");

                    foreach (KeyValuePair<string, string> kvp in style.Value)
                    {
                        classStyle.Append("      ");
                        classStyle.Append(kvp.Key);
                        classStyle.Append(": ");
                        classStyle.Append(kvp.Value);
                        classStyle.Append(";\n");
                    }

                    classStyle.Append("    }\n  ");
                }


                XmlElement styleElement = ctx.Document.CreateElement("style",
SVGContext.SVGNamespace);
                styleElement.InnerText = classStyle.ToString();

                XmlNode svgElement = ctx.Document.GetElementsByTagName("svg")[0];
```

```
                svgElement.InsertBefore(styleElement, svgElement.FirstChild);
            }



            ctx.Document.DocumentElement.SetAttribute("style", "font-synthesis: none;");

            return ctx.Document;
        }

        /// <summary>
        /// Render the animation to an SVG document, using SVG animations.
        /// </summary>
        /// <param name="animation">The <see cref="Animation"/> to render.</param>
        /// <param name="includeControls">If this is <see langword="true"/>, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the file.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="XmlDocument"/> containing the animated SVG image.</returns>
        public static XmlDocument SaveAsAnimatedSVG(this Animation animation, bool includeControls = false, double durationScaling = 1, TextOptions textOption = TextOptions.SubsetFonts, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default)
        {
            (XmlDocument, double)[] frames = new (XmlDocument, double)[animation.Frames.Count];
            double[] durations = new double[animation.Frames.Count];

            (XmlDocument, XmlDocument, Transition)[] transitions = new (XmlDocument, XmlDocument, Transition)[animation.Frames.Count - 1];


            string repeatCount = animation.RepeatCount <= 0 ? "indefinite" : animation.RepeatCount.ToString();

            double currentTime = 0;

            for (int i = 0; i < frames.Length; i++)
            {
                Page pag = new Page(animation.Width, animation.Height) { Background = animation.Background };
                pag.Graphics.DrawGraphics(0, 0, animation.Frames[i].Graphics);

                if (i > 0)
                {
                    frames[i] = (pag.CreateSVGDocument("frame" + i.ToString() + "://", textOption, linkDestinations, filterOption, false, false), animation.Frames[i].Duration);
                    durations[i] = animation.Frames[i].Duration * durationScaling;

                    if (animation.Transitions[i - 1].Duration > 0)
                    {
                        double epsilon = 1e-5 * animation.Transitions[i - 1].Duration;

                        Page startPage = animation.GetFrameAtAbsolute(currentTime + epsilon);
                        Page endPage = animation.GetFrameAtAbsolute(currentTime + animation.Transitions[i - 1].Duration - epsilon);

                        transitions[i - 1] = (startPage.CreateSVGDocument("transition" + i.ToString() + "://", textOption, linkDestinations, filterOption, false, false), endPage.CreateSVGDocument("transition" + i.ToString() + "://", textOption, linkDestinations, filterOption, false, false), animation.Transitions[i - 1]);
                    }
                    else
                    {
                        transitions[i - 1] = (null, null, animation.Transitions[i - 1]);
                    }

                    currentTime += animation.Frames[i].Duration + animation.Transitions[i - 1].Duration;
                }
                else
                {
                    frames[i] = (pag.CreateSVGDocument("frame" + i.ToString() + "://", textOption, linkDestinations, filterOption, false, false), animation.Frames[i].Duration);
                    durations[i] = animation.Frames[i].Duration * durationScaling;
                    currentTime += animation.Frames[i].Duration;
                }
```

```
            }

            double totalDuration = animation.Duration * durationScaling;

            XmlDocument Document = new XmlDocument();

            Document.InsertBefore(Document.CreateXmlDeclaration("1.0", "UTF-8", null), Document.DocumentElement);

            XmlElement currentElement = Document.CreateElement(null, "svg", SVGContext.SVGNamespace);
            currentElement.SetAttribute("xmlns:xlink", "http://www.w3.org/1999/xlink");
            currentElement.SetAttribute("viewBox", "0 0 " + animation.Width.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + animation.Height.ToString(System.Globalization.CultureInfo.InvariantCulture));
            currentElement.SetAttribute("version", "1.1");
            Document.AppendChild(currentElement);

            currentTime = 0;

            for (int i = 0; i < frames.Length; i++)
            {
                if (i > 0 && transitions[i - 1].Item3.Duration > 0)
                {
                    XmlNode clonedSVG = Document.ImportNode(transitions[i - 1].Item1.GetElementsByTagName("svg")[0], true);

                    {
                        XmlElement animate = Document.CreateElement("animate", SVGContext.SVGNamespace);
                        animate.SetAttribute("attributeName", "display");
                        animate.SetAttribute("values", "none;none;block;block;none;none");
                        animate.SetAttribute("dur", totalDuration.ToString() + "ms");
                        animate.SetAttribute("repeatCount", repeatCount);
                        animate.SetAttribute("fill", "freeze");
                        animate.SetAttribute("keyTimes", "0;" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + transitions[i - 1].Item3.Duration * durationScaling) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + transitions[i - 1].Item3.Duration * durationScaling) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";1");
                        clonedSVG.InsertBefore(animate, clonedSVG.FirstChild);
                    }

                    currentElement.AppendChild(clonedSVG);

                    Dictionary<string, XmlElement> taggedStart = GetTaggedElements(clonedSVG);
                    Dictionary<string, XmlElement> taggedEnd = GetTaggedElements(transitions[i - 1].Item2.GetElementsByTagName("svg")[0]);

                    string overallEasing = null;

                    if (transitions[i - 1].Item3.OverallEasing is SplineEasing spline)
                    {
                        overallEasing = "0.5 0.5 0.5 0.5; " + spline.ControlPoint1.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + spline.ControlPoint1.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + spline.ControlPoint2.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + spline.ControlPoint2.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + "; 0.5 0.5 0.5 0.5";
                    }

                    foreach (KeyValuePair<string, XmlElement> kvp in taggedStart)
                    {
                        if (taggedEnd.TryGetValue(kvp.Key, out XmlElement endElement))
                        {
                            List<(string, string, string)> attributesToChange = new List<(string, string, string)>();

                            foreach (XmlAttribute attr in kvp.Value.Attributes)
                            {
                                if (endElement.HasAttribute(attr.Name))
                                {
                                    string oldValue = attr.Value;
                                    string newValue = endElement.GetAttribute(attr.Name);

                                    if (oldValue != newValue)
                                    {
                                        attributesToChange.Add((attr.Name, oldValue, newValue));
                                    }
                                }
                            }

                            string easing = overallEasing;

                            string itemId = kvp.Key.Substring(kvp.Key.IndexOf("://") + 3);
```

```
                            if (kvp.Value.Name == "linearGradient" || kvp.Value.Name == "radialGradient")
                            {
                                itemId = itemId.Substring(itemId.IndexOf("_") + 1);
                            }
                            else if (kvp.Value.Name == "stop")
                            {
                                itemId = itemId.Substring(itemId.IndexOf("_") + 1);
                                itemId = itemId.Substring(0, itemId.IndexOf("/"));
                            }

                            if (transitions[i - 1].Item3.Easings != null && transitions[i - 1].Item3.Easings.TryGetValue(itemId, out IEasing currEasing) && currEasing is SplineEasing currSpline)
                            {
                                easing = "0.5 0.5 0.5 0.5; " + currSpline.ControlPoint1.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + currSpline.ControlPoint1.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + currSpline.ControlPoint2.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + currSpline.ControlPoint2.Y.ToString(System.Globalization.CultureInfo.InvariantCulture) + "; 0.5 0.5 0.5 0.5";
                            }

                            for (int j = 0; j < attributesToChange.Count; j++)
                            {
                                kvp.Value.RemoveAttribute(attributesToChange[j].Item1);

                                if (attributesToChange[j].Item1 != "transform")
                                {
                                    XmlElement animate = Document.CreateElement("animate", SVGContext.SVGNamespace);
                                    animate.SetAttribute("attributeName", attributesToChange[j].Item1);
                                    animate.SetAttribute("values", attributesToChange[j].Item2 + ";" + attributesToChange[j].Item2 + ";" + attributesToChange[j].Item3 + ";" + attributesToChange[j].Item3);
                                    animate.SetAttribute("dur", totalDuration.ToString() + "ms");
                                    animate.SetAttribute("repeatCount", repeatCount);
                                    animate.SetAttribute("fill", "freeze");
                                    animate.SetAttribute("keyTimes", "0;" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + transitions[i - 1].Item3.Duration * durationScaling) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";1");

                                    if (!string.IsNullOrEmpty(easing))
                                    {
                                        animate.SetAttribute("calcMode", "spline");
                                        animate.SetAttribute("keySplines", easing);
                                    }

                                    if (kvp.Value.HasChildNodes)
                                    {
                                        kvp.Value.InsertBefore(animate, kvp.Value.FirstChild);
                                    }
                                    else
                                    {
                                        kvp.Value.AppendChild(animate);
                                    }
                                }
                                else
                                {
                                    if (attributesToChange[j].Item2.StartsWith("matrix") && attributesToChange[j].Item3.StartsWith("matrix"))
                                    {
                                        (Point, double, Size, Size) transformStart = DecomposeMatrix(attributesToChange[j].Item2);
                                        (Point, double, Size, Size) transformEnd = DecomposeMatrix(attributesToChange[j].Item3);

                                        if (transformStart.Item4.Height != 0 || transformEnd.Item4.Height != 0)
                                        {
                                            string skewStart = transformStart.Item4.Height.ToString(System.Globalization.CultureInfo.InvariantCulture);
                                            string skewEnd = transformEnd.Item4.Height.ToString(System.Globalization.CultureInfo.InvariantCulture);

                                            XmlElement animate = Document.CreateElement("animateTransform", SVGContext.SVGNamespace);
                                            animate.SetAttribute("attributeName", attributesToChange[j].Item1);
                                            animate.SetAttribute("type", "skewY");
                                            animate.SetAttribute("values", skewStart + ";" + skewStart + ";" + skewEnd + ";" + skewEnd);
                                            animate.SetAttribute("dur", totalDuration.ToString() + "ms");
```

```
                                        animate.SetAttribute("repeatCount", repeatCount);
                                        animate.SetAttribute("fill", "freeze");
                                        animate.SetAttribute("additive", "sum");
                                        animate.SetAttribute("keyTimes", "0;" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + transitions[i - 1].Item3.Duration * durationScaling) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";1");
                                        if (!string.IsNullOrEmpty(easing))
                                        {
                                            animate.SetAttribute("calcMode", "spline");
                                            animate.SetAttribute("keySplines", easing);
                                        }

                                        if (kvp.Value.HasChildNodes)
                                        {
                                            kvp.Value.InsertBefore(animate, kvp.Value.FirstChild);
                                        }
                                        else
                                        {
                                            kvp.Value.AppendChild(animate);
                                        }
                                    }

                                    if (transformStart.Item4.Width != 0 || transformEnd.Item4.Width != 0)
                                    {
                                        string skewStart = transformStart.Item4.Width.ToString(System.Globalization.CultureInfo.InvariantCulture);
                                        string skewEnd = transformEnd.Item4.Width.ToString(System.Globalization.CultureInfo.InvariantCulture);

                                        XmlElement animate = Document.CreateElement("animateTransform", SVGContext.SVGNamespace);
                                        animate.SetAttribute("attributeName", attributesToChange[j].Item1);
                                        animate.SetAttribute("type", "skewX");
                                        animate.SetAttribute("values", skewStart + ";" + skewStart + ";" + skewEnd + ";" + skewEnd);
                                        animate.SetAttribute("dur", totalDuration.ToString() + "ms");
                                        animate.SetAttribute("repeatCount", repeatCount);
                                        animate.SetAttribute("fill", "freeze");
                                        animate.SetAttribute("additive", "sum");
                                        animate.SetAttribute("keyTimes", "0;" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + transitions[i - 1].Item3.Duration * durationScaling) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";1");
                                        if (!string.IsNullOrEmpty(easing))
                                        {
                                            animate.SetAttribute("calcMode", "spline");
                                            animate.SetAttribute("keySplines", easing);
                                        }

                                        if (kvp.Value.HasChildNodes)
                                        {
                                            kvp.Value.InsertBefore(animate, kvp.Value.FirstChild);
                                        }
                                        else
                                        {
                                            kvp.Value.AppendChild(animate);
                                        }
                                    }

                                    if (transformStart.Item3.Width != 0 || transformStart.Item3.Height != 0 || transformEnd.Item3.Width != 0 || transformEnd.Item3.Height != 0)
                                    {
                                        string scaleStart = transformStart.Item3.Width.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + transformStart.Item3.Height.ToString(System.Globalization.CultureInfo.InvariantCulture);
                                        string scaleEnd = transformEnd.Item3.Width.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + transformEnd.Item3.Height.ToString(System.Globalization.CultureInfo.InvariantCulture);

                                        XmlElement animate = Document.CreateElement("animateTransform", SVGContext.SVGNamespace);
                                        animate.SetAttribute("attributeName", attributesToChange[j].Item1);
                                        animate.SetAttribute("type", "scale");
                                        animate.SetAttribute("values", scaleStart + ";" + scaleStart + ";" + scaleEnd + ";" + scaleEnd);
                                        animate.SetAttribute("dur", totalDuration.ToString() + "ms");
                                        animate.SetAttribute("repeatCount", repeatCount);
                                        animate.SetAttribute("fill", "freeze");
                                        animate.SetAttribute("additive", "sum");
                                        animate.SetAttribute("keyTimes", "0;" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime +
```

```
transitions[i - 1].Item3.Duration * durationScaling) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";1");
                                        if (!string.IsNullOrEmpty(easing))
                                        {
                                            animate.SetAttribute("calcMode", "spline");
                                            animate.SetAttribute("keySplines", easing);
                                        }

                                        if (kvp.Value.HasChildNodes)
                                        {
                                            kvp.Value.InsertBefore(animate, kvp.Value.FirstChild);
                                        }
                                        else
                                        {
                                            kvp.Value.AppendChild(animate);
                                        }
                                    }

                                    if (transformStart.Item2 != 0 || transformEnd.Item2 != 0)
                                    {
                                        string rotateStart = (transformStart.Item2 * 180 / Math.PI).ToString(System.Globalization.CultureInfo.InvariantCulture);
                                        string rotateEnd = (transformEnd.Item2 * 180 / Math.PI).ToString(System.Globalization.CultureInfo.InvariantCulture);

                                        XmlElement animate = Document.CreateElement("animateTransform", SVGContext.SVGNamespace);
                                        animate.SetAttribute("attributeName", attributesToChange[j].Item1);
                                        animate.SetAttribute("type", "rotate");
                                        animate.SetAttribute("values", rotateStart + ";" + rotateStart + ";" + rotateEnd + ";" + rotateEnd);
                                        animate.SetAttribute("dur", totalDuration.ToString() + "ms");
                                        animate.SetAttribute("repeatCount", repeatCount);
                                        animate.SetAttribute("fill", "freeze");
                                        animate.SetAttribute("additive", "sum");
                                        animate.SetAttribute("keyTimes", "0;" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + transitions[i - 1].Item3.Duration * durationScaling) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";1");
                                        if (!string.IsNullOrEmpty(easing))
                                        {
                                            animate.SetAttribute("calcMode", "spline");
                                            animate.SetAttribute("keySplines", easing);
                                        }

                                        if (kvp.Value.HasChildNodes)
                                        {
                                            kvp.Value.InsertBefore(animate, kvp.Value.FirstChild);
                                        }
                                        else
                                        {
                                            kvp.Value.AppendChild(animate);
                                        }
                                    }

                                    if (transformStart.Item1.X != 0 || transformStart.Item1.Y != 0 || transformEnd.Item1.X != 0 || transformEnd.Item1.Y != 0)
                                    {
                                        string translateStart = transformStart.Item1.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + transformStart.Item1.Y.ToString(System.Globalization.CultureInfo.InvariantCulture);
                                        string translateEnd = transformEnd.Item1.X.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + transformEnd.Item1.Y.ToString(System.Globalization.CultureInfo.InvariantCulture);

                                        XmlElement animate = Document.CreateElement("animateTransform", SVGContext.SVGNamespace);
                                        animate.SetAttribute("attributeName", attributesToChange[j].Item1);
                                        animate.SetAttribute("type", "translate");
                                        animate.SetAttribute("values", translateStart + ";" + translateStart + ";" + translateEnd + ";" + translateEnd);
                                        animate.SetAttribute("dur", totalDuration.ToString() + "ms");
                                        animate.SetAttribute("repeatCount", repeatCount);
                                        animate.SetAttribute("fill", "freeze");
                                        animate.SetAttribute("additive", "sum");
                                        animate.SetAttribute("keyTimes", "0;" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + transitions[i - 1].Item3.Duration * durationScaling) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";1");
                                        if (!string.IsNullOrEmpty(easing))
                                        {
                                            animate.SetAttribute("calcMode", "spline");
                                            animate.SetAttribute("keySplines", easing);
```

```
                                            }

                                            if (kvp.Value.HasChildNodes)
                                            {
                                                kvp.Value.InsertBefore(animate, kvp.Value.FirstChild);
                                            }
                                            else
                                            {
                                                kvp.Value.AppendChild(animate);
                                            }
                                        }
                                    }
                                    else
                                    {
                                        throw new NotImplementedException("Invalid transform!");
                                    }
                                }
                            }
                        }
                    }

                    currentTime += transitions[i - 1].Item3.Duration * durationScaling;
                }

                {
                    XmlNode clonedSVG = Document.ImportNode(frames[i].Item1.GetElementsByTagName("svg")[0], true);

                    XmlElement animate = Document.CreateElement("animate", SVGContext.SVGNamespace);
                    animate.SetAttribute("attributeName", "display");
                    animate.SetAttribute("values", "none;none;block;block;none;none");
                    animate.SetAttribute("dur", totalDuration.ToString() + "ms");
                    animate.SetAttribute("repeatCount", repeatCount);

                    if (i < frames.Length - 1)
                    {
                        animate.SetAttribute("fill", "freeze");
                    }

                    animate.SetAttribute("keyTimes", "0;" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + durations[i]) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + durations[i]) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";1");
                    clonedSVG.InsertBefore(animate, clonedSVG.FirstChild);

                    currentElement.AppendChild(clonedSVG);
                }

                currentTime += durations[i];
            }

            if (includeControls)
            {
                double controlsHeight = animation.Height * 0.08;

                double controlsWidth = animation.Width * 0.8;

                if (controlsWidth < controlsHeight * 5)
                {
                    controlsHeight = controlsWidth / 5;
                }

                currentElement.AppendChild(Document.ImportNode(GetAnimationControls(animation.Width, animation.Height, controlsWidth, controlsHeight, animation.Duration * durationScaling, animation.RepeatCount).GetElementsByTagName("svg")[0], true));
            }

            return Document;
        }

        private static XmlDocument GetAnimationControls(double width, double height, double controlsWidth, double controlsHeight, double totalDuration, int repeatCount)
        {
            double controlsMargin = controlsHeight / 12;
            double controlRadius = controlsHeight / 6;

            Page pag = new Page(width, height);

            Graphics gpr = pag.Graphics;

            GraphicsPath background = new GraphicsPath().MoveTo(width * 0.5 - controlsWidth * 0.5 + controlRadius, height - controlsMargin - controlsHeight).LineTo(width * 0.5 + controlsWidth * 0.5 - controlRadius, height - controlsMargin - controlsHeight);
            background.Arc(width * 0.5 + controlsWidth * 0.5 - controlRadius, height - controlsMargin - controlsHeight + controlRadius, controlRadius, -Math.PI / 2, 0).LineTo(width * 0.5 + controlsWidth
```

```
* 0.5, height - controlsMargin - controlRadius);
            background.Arc(width * 0.5 + controlsWidth * 0.5 - controlRadius, height - controlsMargin - controlRadius, controlRadius, 0, Math.PI / 2).LineTo(width * 0.5 - controlsWidth * 0.5 + controlRadius, height - controlsMargin);
            background.Arc(width * 0.5 - controlsWidth * 0.5 + controlRadius, height - controlsMargin - controlRadius, controlRadius, Math.PI / 2, Math.PI).LineTo(width * 0.5 - controlsWidth * 0.5, height - controlsMargin - controlsHeight + controlRadius);
            background.Arc(width * 0.5 - controlsWidth * 0.5 + controlRadius, height - controlsMargin - controlsHeight + controlRadius, controlRadius, Math.PI, 3 * Math.PI / 2).Close();

            gpr.FillPath(background, Colour.FromRgba(0, 0, 0, 0.5), tag: "timeLineBackground");

            double roundedCornerSize = 0.25;

            LinearGradientBrush buttonBrush = new LinearGradientBrush(new Point(0, controlsHeight), new Point(0, 0), new GradientStop(Colours.White, 1), new GradientStop(Colour.FromRgb(220, 220, 220), 0));

            gpr.Save();
            gpr.Translate(width * 0.5 - controlsWidth * 0.5 + controlsHeight * 0.5, height - controlsHeight - controlsMargin);
            GraphicsPath playButton = new GraphicsPath().MoveTo(0, controlsHeight * 0.2 + controlsHeight * 0.6 * roundedCornerSize).LineTo(0, controlsHeight * 0.2 + controlsHeight * 0.6 * (1 - roundedCornerSize)).CubicBezierTo(0, controlsHeight * 0.8, 0, controlsHeight * 0.8, controlsHeight * 0.5 * roundedCornerSize, controlsHeight * 0.5 + controlsHeight * 0.3 * (1 - roundedCornerSize)).LineTo(controlsHeight * 0.5 * (1 - roundedCornerSize), controlsHeight * 0.5 + controlsHeight * 0.3 * roundedCornerSize);
            playButton.CubicBezierTo(controlsHeight * 0.5, controlsHeight * 0.5, controlsHeight * 0.5, controlsHeight * 0.5, controlsHeight * 0.5 * (1 - roundedCornerSize), controlsHeight * 0.5 - controlsHeight * 0.3 * roundedCornerSize).LineTo(controlsHeight * 0.5 * roundedCornerSize, controlsHeight * 0.5 - controlsHeight * 0.3 * (1 - roundedCornerSize));
            playButton.CubicBezierTo(0, controlsHeight * 0.2, 0, controlsHeight * 0.2, 0, controlsHeight * 0.2 + controlsHeight * 0.6 * roundedCornerSize).Close();
            gpr.FillPath(playButton, buttonBrush, "playButton");

            GraphicsPath pauseButton = new GraphicsPath().MoveTo(controlsHeight * 0.2 * roundedCornerSize, controlsHeight * 0.25).LineTo(controlsHeight * 0.2 * (1 - roundedCornerSize), controlsHeight * 0.25).CubicBezierTo(controlsHeight * 0.2, controlsHeight * 0.25, controlsHeight * 0.2, controlsHeight * 0.25, controlsHeight * 0.2, controlsHeight * 0.25 + controlsHeight * 0.5 * roundedCornerSize);
            pauseButton.LineTo(controlsHeight * 0.2, controlsHeight * 0.25 + controlsHeight * 0.5 * (1 - roundedCornerSize)).CubicBezierTo(controlsHeight * 0.2, controlsHeight * 0.75, controlsHeight * 0.2, controlsHeight * 0.75, controlsHeight * 0.2 * (1 - roundedCornerSize), controlsHeight * 0.75).LineTo(controlsHeight * 0.2 * roundedCornerSize, controlsHeight * 0.75);
            pauseButton.CubicBezierTo(0, controlsHeight * 0.75, 0, controlsHeight * 0.75, 0, controlsHeight * 0.25 + controlsHeight * 0.5 * (1 - roundedCornerSize)).LineTo(0, controlsHeight * 0.25 + controlsHeight * 0.5 * roundedCornerSize).CubicBezierTo(0, controlsHeight * 0.25, 0, controlsHeight * 0.25, controlsHeight * 0.2 * roundedCornerSize, controlsHeight * 0.25).Close();
            pauseButton.MoveTo(controlsHeight * 0.3 + controlsHeight * 0.2 * roundedCornerSize, controlsHeight * 0.25).LineTo(controlsHeight * 0.3 + controlsHeight * 0.2 * (1 - roundedCornerSize), controlsHeight * 0.25).CubicBezierTo(controlsHeight * 0.3 + controlsHeight * 0.2, controlsHeight * 0.25, controlsHeight * 0.3 + controlsHeight * 0.2, controlsHeight * 0.25, controlsHeight * 0.3 + controlsHeight * 0.2, controlsHeight * 0.25 + controlsHeight * 0.5 * roundedCornerSize);
            pauseButton.LineTo(controlsHeight * 0.3 + controlsHeight * 0.2, controlsHeight * 0.25 + controlsHeight * 0.5 * (1 - roundedCornerSize)).CubicBezierTo(controlsHeight * 0.3 + controlsHeight * 0.2, controlsHeight * 0.75, controlsHeight * 0.3 + controlsHeight * 0.2, controlsHeight * 0.75, controlsHeight * 0.3 + controlsHeight * 0.2 * (1 - roundedCornerSize), controlsHeight * 0.75).LineTo(controlsHeight * 0.3 + controlsHeight * 0.2 * roundedCornerSize, controlsHeight * 0.75);
            pauseButton.CubicBezierTo(controlsHeight * 0.3, controlsHeight * 0.75, controlsHeight * 0.3, controlsHeight * 0.75, controlsHeight * 0.3, controlsHeight * 0.25 + controlsHeight * 0.5 * (1 - roundedCornerSize)).LineTo(controlsHeight * 0.3, controlsHeight * 0.25 + controlsHeight * 0.5 * roundedCornerSize).CubicBezierTo(controlsHeight * 0.3, controlsHeight * 0.25, controlsHeight * 0.3, controlsHeight * 0.25, controlsHeight * 0.3 + controlsHeight * 0.2 * roundedCornerSize, controlsHeight * 0.25).Close();
            gpr.FillPath(pauseButton, buttonBrush, "pauseButton");

            gpr.FillRectangle(0, controlsHeight * 0.2, controlsHeight * 0.5, controlsHeight * 0.6, Colour.FromRgba(0, 0, 0, 0), tag: "playButtonHitbox");
            gpr.FillRectangle(0, controlsHeight * 0.2, controlsHeight * 0.5, controlsHeight * 0.6, Colour.FromRgba(0, 0, 0, 0), tag: "pauseButtonHitbox");

            double timeLineStartX = controlsHeight * 0.5 + controlsHeight * 0.5 + controlsHeight / 3;

            gpr.Translate(timeLineStartX, 0);

            GraphicsPath timeLine = new GraphicsPath().MoveTo(controlsHeight * 0.1, controlsHeight * 0.4).LineTo(controlsWidth - timeLineStartX - controlsHeight - controlsHeight * 0.1 - controlsHeight / 3, controlsHeight * 0.4);
            timeLine.Arc(controlsWidth - timeLineStartX - controlsHeight - controlsHeight * 0.1 - controlsHeight / 3, controlsHeight * 0.5, controlsHeight * 0.1, -Math.PI / 2, Math.PI / 2).LineTo(controlsHeight * 0.1, controlsHeight * 0.6);
            timeLine.Arc(controlsHeight * 0.1, controlsHeight * 0.5, controlsHeight * 0.1, Math.PI / 2, 3 * Math.PI / 2).Close();

            gpr.FillPath(timeLine, new LinearGradientBrush(new Point(0, controlsHeight * 0.6), new Point(0, controlsHeight * 0.4), new GradientStop(Colour.FromRgba(255, 255, 255, 80), 0), new GradientStop(Colour.FromRgba(255, 255, 255, 40), 1)), tag: "timeLine");
```

```
            gpr.StrokePath(timeLine, Colour.FromRgba(255, 255, 255, 180));

            gpr.Restore();

            GraphicsPath thumbPath = new GraphicsPath().MoveTo(-controlsHeight * 0.2, controlsHeight * 0.3).LineTo(controlsHeight * 0.2, controlsHeight * 0.3);
            thumbPath.Arc(controlsHeight * 0.2, controlsHeight * 0.5, controlsHeight * 0.2, -Math.PI / 2, Math.PI / 2).LineTo(-controlsHeight * 0.2, controlsHeight * 0.7);
            thumbPath.Arc(-controlsHeight * 0.2, controlsHeight * 0.5, controlsHeight * 0.2, Math.PI / 2, 3 * Math.PI / 2).Close();

            Graphics thumbStart = new Graphics();
            thumbStart.Translate(width * 0.5 - controlsWidth * 0.5 + controlsHeight * 0.5 + timeLineStartX, height - controlsHeight - controlsMargin);
            thumbStart.FillPath(thumbPath, buttonBrush, tag: "thumb");

            Graphics thumbEnd = new Graphics();
            thumbEnd.Translate(width * 0.5 + controlsWidth * 0.5 - controlsHeight * 0.5 - controlsHeight * 0.4, height - controlsHeight - controlsMargin);
            thumbEnd.FillPath(thumbPath, buttonBrush, tag: "thumb");

            Animation thumbAnimation = new Animation(width, height, 1) { RepeatCount = repeatCount };
            thumbAnimation.AddFrame(new Frame(thumbStart, 0));
            thumbAnimation.AddFrame(new Frame(thumbEnd, 0), new Transition(totalDuration));

            XmlDocument animationDoc = thumbAnimation.SaveAsAnimatedSVG();

            XmlNodeList svgNodes = animationDoc.GetElementsByTagName("svg");

            for (int i = 1; i < svgNodes.Count; i++)
            {
                ((XmlElement)svgNodes[i]).RemoveChild(((XmlElement)svgNodes[i]).GetElementsByTagName("path")[0]);
            }

            ((XmlElement)animationDoc.GetElementsByTagName("svg")[0]).SetAttribute("style", "pointer-events: none");

            XmlDocument doc = pag.SaveAsSVG();

            XmlNodeList paths = doc.GetElementsByTagName("path");

            for (int i = 0; i < paths.Count; i++)
            {
                if (((XmlElement)paths[i]).GetAttribute("id") == "timeLine")
                {
                    ((XmlElement)paths[i]).SetAttribute("style", "cursor: pointer;");
                }

                if (((XmlElement)paths[i]).GetAttribute("id") == "playButtonHitbox" || ((XmlElement)paths[i]).GetAttribute("id") == "pauseButtonHitbox")
                {
                    ((XmlElement)paths[i]).SetAttribute("style", "cursor: pointer");
                }
            }

            XmlNode animationNode = doc.ImportNode(animationDoc.GetElementsByTagName("svg")[0], true);
            doc.GetElementsByTagName("svg")[0].AppendChild(animationNode);
            ((XmlElement)doc.GetElementsByTagName("svg")[0]).SetAttribute("id", "animationControls");

            ((XmlElement)doc.GetElementsByTagName("svg")[0]).SetAttribute("style", "font-synthesis: none; transition: opacity 500ms;");

            using (Stream sr = Assembly.GetExecutingAssembly().GetManifestResourceStream("VectSharp.SVG.AnimationControls.js"))
            using (StreamReader reader = new StreamReader(sr))
            {
                string javascript = reader.ReadToEnd();
                javascript = javascript.Replace("@@totalLength@@", (totalDuration / 1000).ToString(System.Globalization.CultureInfo.InvariantCulture));

                XmlElement script = doc.CreateElement("script", SVGContext.SVGNamespace);
                script.SetAttribute("type", "text/javascript");
                script.InnerText = javascript;

                doc.GetElementsByTagName("svg")[0].AppendChild(script);
            }

            return doc;
        }

        /// <summary>
        /// Render the animation to an SVG stream, using SVG animations.
        /// </summary>
        /// <param name="animation">The <see cref="Animation"/> to render.</param>
        /// <param name="stream">The <see cref="Stream"/> on which the SVG document will be written.</param>
```

```
        /// <param name="includeControls">If this is <see langword="true"/>, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the file.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        public static void SaveAsAnimatedSVG(this Animation animation, Stream stream, bool includeControls = false, double durationScaling = 1, TextOptions textOption = TextOptions.SubsetFonts, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default)
        {
            XmlDocument document = SaveAsAnimatedSVG(animation, includeControls, durationScaling, textOption, linkDestinations, filterOption);
            WriteXMLToStream(document.DocumentElement, stream);
        }

        /// <summary>
        /// Render the animation to an SVG file, using SVG animations.
        /// </summary>
        /// <param name="animation">The <see cref="Animation"/> to render.</param>
        /// <param name="fileName">The output file that will be created.</param>
        /// <param name="includeControls">If this is <see langword="true"/>, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the file.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        public static void SaveAsAnimatedSVG(this Animation animation, string fileName, bool includeControls = false, double durationScaling = 1, TextOptions textOption = TextOptions.SubsetFonts, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default)
        {
            using (FileStream fs = File.Create(fileName))
            {
                SaveAsAnimatedSVG(animation, fs, includeControls, durationScaling, textOption, linkDestinations, filterOption);
            }
        }

        /// <summary>
        /// Render the animation to an SVG document, encoding discrete frames.
        /// </summary>
        /// <param name="animation">The <see cref="Animation"/> to render.</param>
        /// <param name="includeControls">If this is <see langword="true"/>, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second (fps).</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the file.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        /// <returns>An <see cref="XmlDocument"/> containing the animated SVG image.</returns>
        public static XmlDocument SaveAsAnimatedSVGWithFrames(this Animation animation, bool includeControls = false, double frameRate = 60, double durationScaling = 1, TextOptions textOption = TextOptions.SubsetFonts, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default)
        {
            int frameCount = (int)Math.Ceiling(animation.Duration * frameRate * durationScaling / 1000);

```

```
            (XmlDocument, double)[] frames = new (XmlDocument, double)[frameCount];

            Parallel.For(0, frameCount, i =>
            {
                double frameTime = i / frameRate / durationScaling * 1000;

                double frameDuration = Math.Min((animation.Duration - frameTime) * durationScaling, 1000 / frameRate);

                Page pag = animation.GetFrameAtAbsolute(frameTime);

                frames[i] = (pag.CreateSVGDocument("frame" + i.ToString() + "://", textOption, linkDestinations, filterOption, false, true), frameDuration);
            });

            string repeatCount = animation.RepeatCount <= 0 ? "indefinite" : animation.RepeatCount.ToString();

            double totalDuration = animation.Duration * durationScaling;

            XmlDocument Document = new XmlDocument();

            Document.InsertBefore(Document.CreateXmlDeclaration("1.0", "UTF-8", null), Document.DocumentElement);

            XmlElement currentElement = Document.CreateElement(null, "svg", SVGContext.SVGNamespace);
            currentElement.SetAttribute("xmlns:xlink", "http://www.w3.org/1999/xlink");
            currentElement.SetAttribute("viewBox", "0 0 " + animation.Width.ToString(System.Globalization.CultureInfo.InvariantCulture) + " " + animation.Height.ToString(System.Globalization.CultureInfo.InvariantCulture));
            currentElement.SetAttribute("version", "1.1");
            Document.AppendChild(currentElement);

            double currentTime = 0;

            for (int i = 0; i < frames.Length; i++)
            {
                XmlNode clonedSVG = Document.ImportNode(frames[i].Item1.GetElementsByTagName("svg")[0], true);

                XmlElement animate = Document.CreateElement("animate", SVGContext.SVGNamespace);
                animate.SetAttribute("attributeName", "display");
                animate.SetAttribute("values", "none;none;block;block;none;none");
                animate.SetAttribute("dur", totalDuration.ToString() + "ms");
                animate.SetAttribute("repeatCount", repeatCount);
                animate.SetAttribute("keyTimes", "0;" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + (currentTime / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + frames[i].Item2) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";" + ((currentTime + frames[i].Item2) / totalDuration).ToString(System.Globalization.CultureInfo.InvariantCulture) + ";1");
                clonedSVG.InsertBefore(animate, clonedSVG.FirstChild);

                currentElement.AppendChild(clonedSVG);

                currentTime += frames[i].Item2;
            }

            if (includeControls)
            {
                double controlsHeight = animation.Height * 0.08;

                double controlsWidth = animation.Width * 0.8;

                if (controlsWidth < controlsHeight * 5)
                {
                    controlsHeight = controlsWidth / 5;
                }

                currentElement.AppendChild(Document.ImportNode(GetAnimationControls(animation.Width, animation.Height, controlsWidth, controlsHeight, animation.Duration * durationScaling, animation.RepeatCount).GetElementsByTagName("svg")[0], true));
            }

            return Document;
        }

        /// <summary>
        /// Render the animation to an SVG stream, encoding discrete frames.
        /// </summary>
        /// <param name="animation">The <see cref="Animation"/> to render.</param>
        /// <param name="includeControls">If this is <see langword="true"/>, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.</param>
        /// <param name="stream">The <see cref="Stream"/> on which the SVG document will be written.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second
```

```
(fps).</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the file.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        public static void SaveAsAnimatedSVGWithFrames(this Animation animation, Stream stream, bool includeControls = false, double frameRate = 60, double durationScaling = 1, TextOptions textOption = TextOptions.SubsetFonts, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default)
        {
            XmlDocument document = SaveAsAnimatedSVGWithFrames(animation, includeControls, frameRate, durationScaling, textOption, linkDestinations, filterOption);
            WriteXMLToStream(document.DocumentElement, stream);
        }

        /// <summary>
        /// Render the animation to an SVG file, encoding discrete frames.
        /// </summary>
        /// <param name="animation">The <see cref="Animation"/> to render.</param>
        /// <param name="includeControls">If this is <see langword="true"/>, the generated SVG file will contain playback controls that use Javascript to play/pause the animation and change the current time.</param>
        /// <param name="fileName">The output file that will be created.</param>
        /// <param name="frameRate">The target frame rate of the animation, in frames-per-second (fps).</param>
        /// <param name="durationScaling">A scaling factor that will be applied to all durations in the animation. Values greater than 1 slow down the animation, values smaller than 1 accelerate it. Note that this does not affect the frame rate of the animation.</param>
        /// <param name="textOption">Defines whether the used fonts should be included in the file.</param>
        /// <param name="linkDestinations">A dictionary associating element tags to link targets. If this is provided, objects that have been drawn with a tag contained in the dictionary will become hyperlink to the destination specified in the dictionary. If the destination starts with a hash (#), it is interpreted as the tag of another object in the current document; otherwise, it is interpreted as an external URI.</param>
        /// <param name="filterOption">Defines how and whether image filters should be rasterised when rendering the image.</param>
        public static void SaveAsAnimatedSVGWithFrames(this Animation animation, string fileName, bool includeControls = false, double frameRate = 60, double durationScaling = 1, TextOptions textOption = TextOptions.SubsetFonts, Dictionary<string, string> linkDestinations = null, FilterOption filterOption = default)
        {
            using (FileStream fs = File.Create(fileName))
            {
                SaveAsAnimatedSVGWithFrames(animation, fs, includeControls, frameRate, durationScaling, textOption, linkDestinations, filterOption);
            }
        }

        // Adapted from https://math.stackexchange.com/a/2888105
        private static (Point, double, Size, Size) DecomposeMatrix(string transformMatrix)
        {
            transformMatrix = transformMatrix.Trim().Substring(7);
            transformMatrix = transformMatrix.Substring(0, transformMatrix.Length - 1);

            double[] matrixElements = (from el in transformMatrix.Split(',') select double.Parse(el, System.Globalization.CultureInfo.InvariantCulture)).ToArray();

            double a = matrixElements[0];
            double b = matrixElements[1];
            double c = matrixElements[2];
            double d = matrixElements[3];
            double e = matrixElements[4];
            double f = matrixElements[5];

            double delta = a * d - b * c;

            Point translation = new Point(e, f);
            double rotation = 0;
            Size scale = new Size(0, 0);
            Size skew = new Size(0, 0);

            if (a != 0 || b != 0)
            {
                double r = Math.Sqrt(a * a + b * b);
                rotation = b > 0 ? Math.Acos(a / r) : -Math.Acos(a / r);
                scale = new Size(r, delta / r);
                new Size(Math.Atan((a * c + b * d) / (r * r)), 0);
            }
```

```
            else if (c != 0 || d != 0)
            {
                double s = Math.Sqrt(c * c + d * d);
                rotation = Math.PI / 2 - (d > 0 ? Math.Acos(-c / s) : -Math.Acos(c / s));
                scale = new Size(delta / s, s);
                skew = new Size(0, Math.Atan((a * c + b * d) / (s * s)));
            }
            else
            {
                // a = b = c = d = 0
            }

            return (translation, rotation, scale, skew);
        }

        private static Dictionary<string, XmlElement> GetTaggedElements(XmlNode node)
        {
            Dictionary<string, XmlElement> tbr = new Dictionary<string, XmlElement>();

            foreach (XmlNode childNode in node)
            {
                if (childNode is XmlElement child)
                {
                    string id = child.GetAttribute("id");
                    if (!string.IsNullOrEmpty(id))
                    {
                        tbr[id] = child;
                    }

                    foreach (KeyValuePair<string, XmlElement> kvp in GetTaggedElements(child))
                    {
                        tbr[kvp.Key] = kvp.Value;
                    }
                }
            }

            return tbr;
        }

        internal static XmlElement ToLinearGradient(this LinearGradientBrush brush, XmlDocument document, string gradientId)
        {
            XmlElement gradient = document.CreateElement("linearGradient", SVGContext.SVGNamespace);

            gradient.SetAttribute("id", gradientId);

            gradient.SetAttribute("gradientUnits", "userSpaceOnUse");

            gradient.SetAttribute("x1", brush.StartPoint.X.ToString(System.Globalization.CultureInfo.InvariantCulture));
            gradient.SetAttribute("y1", brush.StartPoint.Y.ToString(System.Globalization.CultureInfo.InvariantCulture));
            gradient.SetAttribute("x2", brush.EndPoint.X.ToString(System.Globalization.CultureInfo.InvariantCulture));
            gradient.SetAttribute("y2", brush.EndPoint.Y.ToString(System.Globalization.CultureInfo.InvariantCulture));

            int index = 0;
            foreach (GradientStop stop in brush.GradientStops)
            {
                XmlElement gradientStop = document.CreateElement("stop", SVGContext.SVGNamespace);

                gradientStop.SetAttribute("offset", stop.Offset.ToString(System.Globalization.CultureInfo.InvariantCulture));
                gradientStop.SetAttribute("stop-color", stop.Colour.ToCSSString(false));
                gradientStop.SetAttribute("stop-opacity", stop.Colour.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
                gradientStop.SetAttribute("id", gradientId + "/stop" + index.ToString());

                gradient.AppendChild(gradientStop);
                index++;
            }

            return gradient;
        }

        internal static XmlElement ToRadialGradient(this RadialGradientBrush brush, XmlDocument document, string gradientId)
        {
            XmlElement gradient = document.CreateElement("radialGradient", SVGContext.SVGNamespace);

            gradient.SetAttribute("id", gradientId);

            gradient.SetAttribute("gradientUnits", "userSpaceOnUse");

            gradient.SetAttribute("cx", brush.Centre.X.ToString(System.Globalization.CultureInfo.InvariantCulture));
```

```
            gradient.SetAttribute("cy", brush.Centre.Y.ToString(System.Globalization.CultureInfo.InvariantCulture));
            gradient.SetAttribute("r", brush.Radius.ToString(System.Globalization.CultureInfo.InvariantCulture));
            gradient.SetAttribute("fx", brush.FocalPoint.X.ToString(System.Globalization.CultureInfo.InvariantCulture));
            gradient.SetAttribute("fy", brush.FocalPoint.Y.ToString(System.Globalization.CultureInfo.InvariantCulture));

            int index = 0;
            foreach (GradientStop stop in brush.GradientStops)
            {
                XmlElement gradientStop = document.CreateElement("stop", SVGContext.SVGNamespace);

                gradientStop.SetAttribute("offset", stop.Offset.ToString(System.Globalization.CultureInfo.InvariantCulture));
                gradientStop.SetAttribute("stop-color", stop.Colour.ToCSSString(false));
                gradientStop.SetAttribute("stop-opacity", stop.Colour.A.ToString(System.Globalization.CultureInfo.InvariantCulture));
                gradientStop.SetAttribute("id", gradientId + "/stop" + index.ToString());

                gradient.AppendChild(gradientStop);
                index++;
            }

            return gradient;
        }

        // Adapted from http://www.ericwhite.com/blog/2011/05/09/custom-formatting-of-xml-using-linq-to-xml-2/
        private static void WriteStartElement(XmlWriter writer, XmlElement e)
        {
            writer.WriteStartElement(e.Prefix, e.LocalName, e.NamespaceURI);

            foreach (XmlAttribute a in e.Attributes)
            {
                writer.WriteAttributeString(a.Prefix, a.LocalName, a.NamespaceURI, a.Value);
            }
        }

        private static void WriteElement(XmlWriter writer, XmlElement e)
        {
            if (e.Name == "text")
            {
                XmlWriterSettings settings = new XmlWriterSettings();
                settings.Indent = false;
                settings.OmitXmlDeclaration = true;
                settings.ConformanceLevel = ConformanceLevel.Fragment;
                settings.NamespaceHandling = NamespaceHandling.OmitDuplicates;

                WriteStartElement(writer, e);

                StringBuilder sb = new StringBuilder();

                using (XmlWriter newWriter = XmlWriter.Create(sb, settings))
                {
                    foreach (XmlNode n in e.ChildNodes)
                    {
                        n.WriteTo(newWriter);
                    }
                }

                writer.WriteRaw(sb.ToString().Replace(" xmlns=\"http://www.w3.org/2000/svg\">", ">"));

                writer.WriteEndElement();
            }
            else
            {
                WriteStartElement(writer, e);
                foreach (XmlNode n in e.ChildNodes)
                {
                    if (n is XmlElement element)
                    {
                        WriteElement(writer, element);
                    }
                    else
                    {
                        n.WriteTo(writer);
                    }
                }
                writer.WriteEndElement();
            }
        }

        private static void WriteXMLToStream(XmlElement element, Stream output)
        {
            XmlWriterSettings settings = new XmlWriterSettings();
```

```
            settings.Indent = true;

            using (XmlWriter writer = XmlWriter.Create(output, settings))
            {
                WriteElement(writer, element);
            }
        }
    }
}
```

8.24 SVGParser.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using ExCSS;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.Xml;
using VectSharp.Filters;

namespace VectSharp.SVG
{
    /// <summary>
    /// Contains methods to read an SVG image file.
    /// </summary>
    public static class Parser
    {
        static Parser()
        {
            ParseImageURI = ParseSVGURI;
        }

        /// <summary>
        /// A function that takes as input an image URI and a boolean value indicating whether the image should be interpolated, and returns a <see cref="Page"/> object containing the image.
        /// By default, this is equal to <see cref="ParseSVGURI"/>, i.e. it is only able to parse SVG images. If you wish to enable the parsing of other formats, you should install the "VectSharp.MuPDFUtils" NuGet package
        /// and enable the parser in your program by doing something like:
        /// <code>VectSharp.SVG.Parser.ParseImageURI = VectSharp.MuPDFUtils.ImageURIParser.Parser(VectSharp.SVG.Parser.ParseSVGURI);</code>
        /// </summary>
        public static Func<string, bool, Page> ParseImageURI;

        /// <summary>
        /// Parses an SVG image URI.
        /// </summary>
        /// <param name="uri">The image URI to parse.</param>
        /// <param name="ignored">This value is ignored and is only needed for compatibility.</param>
        /// <returns>A <see cref="Page"/> containing the parsed SVG image, or null.</returns>
        public static Page ParseSVGURI(string uri, bool ignored = false)
        {
            if (uri.StartsWith("data:"))
            {
                string mimeType = uri.Substring(uri.IndexOf(":") + 1, uri.IndexOf(";") - uri.IndexOf(":") - 1);

                string type = uri.Substring(uri.IndexOf(";") + 1, uri.IndexOf(",") - uri.IndexOf(";") - 1);

                if (mimeType == "image/svg+xml")
                {
                    int offset = uri.IndexOf(",") + 1;

```

```
                    string data;

                    switch (type)
                    {
                        case "base64":
                            data = Encoding.UTF8.GetString(Convert.FromBase64String(uri.Substring(offset)));
                            break;
                        case "":
                        case "charset=utf-8":
                        case "utf-8":
                        case "utf8":
                            data = System.Web.HttpUtility.UrlDecode(uri.Substring(offset));
                            break;
                        case "charset=ascii":
                        case "ascii":
                            data = System.Web.HttpUtility.UrlDecode(uri.Substring(offset));
                            break;
                        default:
                            throw new InvalidDataException("Unknown data stream type!");
                    }

                    try
                    {
                        StringReader sr = new StringReader(data);
                        string firstLine = sr.ReadLine();
                        sr.Dispose();

                        if (firstLine.StartsWith("<?xml") && firstLine.EndsWith("?>"))
                        {
                            data = data.Substring(firstLine.Length + 1);
                        }
                    }
                    catch { }

                    return FromString(data);
                }
                else
                {
                    return null;
                }
            }

            return null;
        }

        /// <summary>
        /// Parses SVG source into a <see cref="Page"/> containing the image represented by the code.
        /// </summary>
        /// <param name="svgSource">The SVG source code.</param>
        /// <returns>A <see cref="Page"/> containing the image represented by the <paramref name="svgSource"/>.</returns>
        public static Page FromString(string svgSource)
        {
            XmlDocument svgDoc = new XmlDocument();
            svgDoc.LoadXml(svgSource);

            Dictionary<string, FontFamily> embeddedFonts = new Dictionary<string, FontFamily>();

            StylesheetParser parser = new StylesheetParser();

            List<Stylesheet> styleSheets = new List<Stylesheet>();

            foreach (XmlNode styleNode in svgDoc.GetElementsByTagName("style"))
            {
                foreach (KeyValuePair<string, FontFamily> fnt in GetEmbeddedFonts(styleNode.InnerText))
                {
                    embeddedFonts.Add(fnt.Key, fnt.Value);
                }

                try
                {
                    Stylesheet sheet = parser.Parse(styleNode.InnerText);
                    styleSheets.Add(sheet);
                }
                catch { }
            }

            Dictionary<string, Brush> gradients = new Dictionary<string, Brush>();
            Dictionary<string, IFilter> filters = new Dictionary<string, IFilter>();
            Dictionary<string, XmlNode> masks = new Dictionary<string, XmlNode>();

            foreach (XmlNode definitionsNode in svgDoc.GetElementsByTagName("defs"))
            {
                foreach (KeyValuePair<string, Brush> fnt in GetGradients(definitionsNode, styleSheets))
```

```
                {
                    gradients.Add(fnt.Key, fnt.Value);
                }

                foreach (KeyValuePair<string, IFilter> filt in GetFilters(definitionsNode, styleSheets))
                {
                    filters.Add(filt.Key, filt.Value);
                }

                foreach (KeyValuePair<string, XmlNode> mask in GetMasks(definitionsNode, styleSheets))
                {
                    masks.Add(mask.Key, mask.Value);
                }
            }

            Graphics gpr = new Graphics();

            Size pageSize = InterpretSVGObject(svgDoc.GetElementsByTagName("svg")[0], gpr, new PresentationAttributes() { EmbeddedFonts = embeddedFonts }, styleSheets, gradients, filters, masks);

            Page pg = new Page(pageSize.Width, pageSize.Height);

            pg.Graphics = gpr;

            return pg;
        }

        /// <summary>
        /// Parses an SVG image file into a <see cref="Page"/> containing the image.
        /// </summary>
        /// <param name="fileName">The path to the SVG image file.</param>
        /// <returns>A <see cref="Page"/> containing the image represented by the file.</returns>
        public static Page FromFile(string fileName)
        {
            return FromString(File.ReadAllText(fileName));
        }

        /// <summary>
        /// Parses an stream containing SVG source code into a <see cref="Page"/> containing the image represented by the code.
        /// </summary>
        /// <param name="svgSourceStream">The stream containing SVG source code.</param>
        /// <returns>A <see cref="Page"/> containing the image represented by the <paramref name="svgSourceStream"/>.</returns>
        public static Page FromStream(Stream svgSourceStream)
        {
            using (StreamReader sr = new StreamReader(svgSourceStream))
            {
                return FromString(sr.ReadToEnd());
            }
        }

        private static Size InterpretSVGObject(XmlNode svgObject, Graphics gpr, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients, Dictionary<string, IFilter> filters, Dictionary<string, XmlNode> masks)
        {
            double[] viewBox = ParseListOfDoubles(svgObject.Attributes?["viewBox"]?.Value);

            double width, height, x, y;

            string widthAttribute = svgObject.Attributes?["width"]?.Value?.Replace("px", "");

            if (!double.TryParse(widthAttribute, System.Globalization.NumberStyles.Any, System.Globalization.CultureInfo.InvariantCulture, out width)) { width = double.NaN; }

            string heightAttribute = svgObject.Attributes?["height"]?.Value?.Replace("px", "");
            if (!double.TryParse(heightAttribute, System.Globalization.NumberStyles.Any, System.Globalization.CultureInfo.InvariantCulture, out height)) { height = double.NaN; }

            string xAttribute = svgObject.Attributes?["x"]?.Value;
            double.TryParse(xAttribute, System.Globalization.NumberStyles.Any, System.Globalization.CultureInfo.InvariantCulture, out x);

            string yAttribute = svgObject.Attributes?["y"]?.Value;
            double.TryParse(yAttribute, System.Globalization.NumberStyles.Any, System.Globalization.CultureInfo.InvariantCulture, out y);

            double scaleX = 1;
            double scaleY = 1;

            double postTranslateX = 0;
            double postTranslateY = 0;

            if (viewBox != null)
            {
                if (!double.IsNaN(width) && !double.IsNaN(height))
```

```
                {
                    scaleX = width / viewBox[2];
                    scaleY = height / viewBox[3];
                }
                else if (!double.IsNaN(width) && double.IsNaN(height))
                {
                    scaleX = width / viewBox[2];
                    scaleY = scaleX;
                    height = scaleY * viewBox[3];
                }
                else if (double.IsNaN(width) && !double.IsNaN(height))
                {
                    scaleY = height / viewBox[3];
                    scaleX = scaleY;
                    width = scaleX * viewBox[2];
                }
                else if (double.IsNaN(width) && double.IsNaN(height))
                {
                    width = viewBox[2];
                    height = viewBox[3];
                }

                postTranslateX = -viewBox[0];
                postTranslateY = -viewBox[1];
            }
            else
            {
                viewBox = new double[4];

                if (!double.IsNaN(width))
                {
                    viewBox[2] = width;
                }

                if (!double.IsNaN(height))
                {
                    viewBox[3] = height;
                }
            }

            double diagonal = Math.Sqrt(viewBox[2] * viewBox[2] + viewBox[3] * viewBox[3]) / Math.Sqrt(2);

            Size tbrSize = new Size(width, height);

            gpr.Save();
            gpr.Translate(x, y);
            gpr.Scale(scaleX, scaleY);
            gpr.Translate(postTranslateX, postTranslateY);

            attributes = InterpretPresentationAttributes(svgObject, attributes, viewBox[2], viewBox[3], diagonal, gpr, styleSheets, gradients);

            foreach (KeyValuePair<string, XmlNode> mask in masks)
            {
                Graphics maskGpr = new Graphics();
                InterpretGObject(mask.Value, maskGpr, viewBox[2], viewBox[3], diagonal, attributes, styleSheets, gradients, filters);

                filters.Add(mask.Key, new MaskFilter(maskGpr));
            }

            InterpretSVGChildren(svgObject, gpr, attributes, viewBox[2], viewBox[3], diagonal, styleSheets, gradients, filters);

            gpr.Restore();

            return tbrSize;
        }

        private static void InterpretSVGChildren(XmlNode svgObject, Graphics gpr, PresentationAttributes attributes, double width, double height, double diagonal, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients, Dictionary<string, IFilter> filters)
        {
            foreach (XmlNode child in svgObject.ChildNodes)
            {
                InterpretSVGElement(child, gpr, attributes, width, height, diagonal, styleSheets, gradients, filters);
            }
        }

        private static void InterpretSVGElement(XmlNode currObject, Graphics gpr, PresentationAttributes attributes, double width, double height, double diagonal, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients, Dictionary<string, IFilter> filters)
        {
```

```
            if (currObject.NodeType == XmlNodeType.EntityReference)
            {
                InterpretSVGChildren(currObject, gpr, attributes, width, height, diagonal, styleSheets, gradients, filters);
            }
            else if (currObject.Name.Equals("svg", StringComparison.OrdinalIgnoreCase))
            {
                InterpretSVGObject(currObject, gpr, attributes, styleSheets, gradients, filters, new Dictionary<string, XmlNode>());
            }
            else if (currObject.Name.Equals("line", StringComparison.OrdinalIgnoreCase))
            {
                InterpretLineObject(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);
            }
            else if (currObject.Name.Equals("circle", StringComparison.OrdinalIgnoreCase))
            {
                InterpretCircleObject(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);
            }
            else if (currObject.Name.Equals("ellipse", StringComparison.OrdinalIgnoreCase))
            {
                InterpretEllipseObject(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);
            }
            else if (currObject.Name.Equals("path", StringComparison.OrdinalIgnoreCase))
            {
                InterpretPathObject(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);
            }
            else if (currObject.Name.Equals("polyline", StringComparison.OrdinalIgnoreCase))
            {
                InterpretPolyLineObject(currObject, false, gpr, width, height, diagonal, attributes, styleSheets, gradients);
            }
            else if (currObject.Name.Equals("polygon", StringComparison.OrdinalIgnoreCase))
            {
                InterpretPolyLineObject(currObject, true, gpr, width, height, diagonal, attributes, styleSheets, gradients);
            }
            else if (currObject.Name.Equals("rect", StringComparison.OrdinalIgnoreCase))
            {
                InterpretRectObject(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);
            }
            else if (currObject.Name.Equals("use", StringComparison.OrdinalIgnoreCase))
            {
                InterpretUseObject(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients, filters);
            }
            else if (currObject.Name.Equals("g", StringComparison.OrdinalIgnoreCase) || currObject.Name.Equals("symbol", StringComparison.OrdinalIgnoreCase))
            {
                InterpretGObject(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients, filters);
            }
            else if (currObject.Name.Equals("text", StringComparison.OrdinalIgnoreCase))
            {
                double x = 0;
                double y = 0;

                InterpretTextObject(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients, ref x, ref y);
            }
            else if (currObject.Name.Equals("image", StringComparison.OrdinalIgnoreCase))
            {
                InterpretImageObject(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);
            }
        }

        private static void InterpretImageObject(XmlNode currObject, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients)
        {
            PresentationAttributes currAttributes = InterpretPresentationAttributes(currObject, attributes, width, height, diagonal, gpr, styleSheets, gradients);

            double x = ParseLengthOrPercentage(currObject.Attributes?["x"]?.Value, width, currAttributes.X);
            double y = ParseLengthOrPercentage(currObject.Attributes?["y"]?.Value, height, currAttributes.Y);

            double w = ParseLengthOrPercentage(currObject.Attributes?["width"]?.Value, width, currAttributes.Width);
            double h = ParseLengthOrPercentage(currObject.Attributes?["height"]?.Value, height, currAttributes.Height);
```

```

            bool interpolate = !(currObject.Attributes?["image-rendering"]?.Value == "pixelated" || currObject.Attributes?["image-rendering"]?.Value == "optimizeSpeed");

            string href = currObject.Attributes?["href"]?.Value;

            if (string.IsNullOrEmpty(href))
            {
                href = currObject.Attributes?["xlink:href"]?.Value;
            }

            bool hadClippingPath = ApplyClipPath(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);

            string tag = currObject.Attributes?["id"]?.Value;

            if (!string.IsNullOrEmpty(href) && w > 0 && h > 0)
            {
                Page image = ParseImageURI(href, interpolate);

                if (image != null)
                {
                    gpr.Save();

                    double scaleX = w / image.Width;
                    double scaleY = h / image.Height;

                    gpr.Scale(scaleX, scaleY);

                    gpr.DrawGraphics(x / scaleX, y / scaleY, image.Graphics, tag: tag);

                    gpr.Restore();
                }
                else
                {
                    gpr.StrokeRectangle(x, y, w, h, Colours.Red, 0.1, tag: tag);
                    gpr.StrokePath(new GraphicsPath().MoveTo(x, y).LineTo(x + w, y + h).MoveTo(x + w, y).LineTo(x, y + h), Colours.Red, 0.1, tag: tag);
                }
            }

            if (hadClippingPath)
            {
                gpr.Restore();
            }

            if (currAttributes.NeedsRestore)
            {
                gpr.Restore();
            }
        }

        private static void InterpretTextObject(XmlNode currObject, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients, ref double x, ref double y, double fontSize = double.NaN, string fontFamily = null, string textAlign = null)
        {
            PresentationAttributes currAttributes = InterpretPresentationAttributes(currObject, attributes, width, height, diagonal, gpr, styleSheets, gradients);

            x = ParseLengthOrPercentage(currObject.Attributes?["x"]?.Value, width, x);
            y = ParseLengthOrPercentage(currObject.Attributes?["y"]?.Value, height, y);

            double dx = ParseLengthOrPercentage(currObject.Attributes?["dx"]?.Value, width, 0);
            double dy = ParseLengthOrPercentage(currObject.Attributes?["dy"]?.Value, height, 0);

            x += dx;
            y += dy;

            fontFamily = currObject.Attributes?["font-family"]?.Value ?? fontFamily;
            fontSize = ParseLengthOrPercentage(currObject.Attributes?["font-size"]?.Value, width, fontSize);
            textAlign = currObject.Attributes?["text-align"]?.Value ?? textAlign;

            bool hadClippingPath = ApplyClipPath(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);

            string tag = currObject.Attributes?["id"]?.Value;

            if (currObject.ChildNodes.OfType<XmlNode>().Any(a => a.NodeType != XmlNodeType.Text))
            {
                foreach (XmlNode child in currObject.ChildNodes)
                {
                    InterpretTextObject(child, gpr, width, height, diagonal, currAttributes, styleSheets, gradients, ref x, ref y, fontSize, fontFamily, textAlign);
                }

```

```
                if (hadClippingPath)
                {
                    gpr.Restore();
                }

                if (currAttributes.NeedsRestore)
                {
                    gpr.Restore();
                }
            }
            else
            {
                string text = currObject.InnerText;

                if (!double.IsNaN(fontSize) && !string.IsNullOrEmpty(text))
                {
                    text = text.Replace("\u00A0", " ");

                    FontFamily parsedFontFamily = ParseFontFamily(fontFamily, currAttributes.EmbeddedFonts);
                    string fontWeight = currObject.Attributes?["font-weight"]?.Value;
                    string fontStyle = currObject.Attributes?["font-style"]?.Value;

                    if (fontWeight != null && (fontWeight.Equals("bold", StringComparison.OrdinalIgnoreCase) || fontWeight.Equals("bolder", StringComparison.OrdinalIgnoreCase) || (int.TryParse(fontWeight, out int weight) && weight >= 500)))
                    {
                        parsedFontFamily = GetBoldFontFamily(parsedFontFamily);
                    }

                    if (fontStyle != null && (fontStyle.Equals("italic", StringComparison.OrdinalIgnoreCase) || fontStyle.Equals("oblique", StringComparison.OrdinalIgnoreCase)))
                    {
                        parsedFontFamily = GetItalicFontFamily(parsedFontFamily);
                    }

                    Font fnt = new Font(parsedFontFamily, fontSize);

                    double endX = x;

                    if (fnt.FontFamily.TrueTypeFile != null)
                    {
                        Font.DetailedFontMetrics metrics = fnt.MeasureTextAdvanced(text);
                        x += metrics.LeftSideBearing;

                        if (!string.IsNullOrEmpty(textAlign) && (textAlign.Equals("right", StringComparison.OrdinalIgnoreCase) || textAlign.Equals("end", StringComparison.OrdinalIgnoreCase)))
                        {
                            x -= metrics.Width + metrics.LeftSideBearing;
                        }
                        else if (!string.IsNullOrEmpty(textAlign) && textAlign.Equals("center", StringComparison.OrdinalIgnoreCase))
                        {
                            x -= metrics.Width * 0.5;
                        }

                        endX += metrics.AdvanceWidth;
                    }

                    TextBaselines baseline = TextBaselines.Baseline;

                    string textBaseline = currObject.Attributes?["alignment-baseline"]?.Value;

                    if (textBaseline != null)
                    {
                        if (textBaseline.Equals("text-bottom", StringComparison.OrdinalIgnoreCase) || textBaseline.Equals("bottom", StringComparison.OrdinalIgnoreCase))
                        {
                            baseline = TextBaselines.Bottom;
                        }
                        if (textBaseline.Equals("middle", StringComparison.OrdinalIgnoreCase) || textBaseline.Equals("central", StringComparison.OrdinalIgnoreCase) || textBaseline.Equals("center", StringComparison.OrdinalIgnoreCase))
                        {
                            baseline = TextBaselines.Middle;
                        }
                        if (textBaseline.Equals("text-top", StringComparison.OrdinalIgnoreCase) || textBaseline.Equals("top", StringComparison.OrdinalIgnoreCase) || textBaseline.Equals("hanging", StringComparison.OrdinalIgnoreCase))
                        {
                            baseline = TextBaselines.Top;
                        }
                    }

                    if (currAttributes.StrokeFirst)
                    {
```

```
                        if (currAttributes.Stroke != null)
                        {
                            Brush strokeColour = currAttributes.Stroke.MultiplyOpacity(currAttributes.Opacity * currAttributes.StrokeOpacity);
                            gpr.StrokeText(x, y, text, fnt, strokeColour, baseline, currAttributes.StrokeThickness, currAttributes.LineCap, currAttributes.LineJoin, currAttributes.LineDash, tag: tag);
                        }

                        if (currAttributes.Fill != null)
                        {
                            Brush fillColour = currAttributes.Fill.MultiplyOpacity(currAttributes.Opacity * currAttributes.FillOpacity);
                            gpr.FillText(x, y, text, fnt, fillColour, baseline, tag: tag);
                        }
                    }
                    else
                    {
                        if (currAttributes.Fill != null)
                        {
                            Brush fillColour = currAttributes.Fill.MultiplyOpacity(currAttributes.Opacity * currAttributes.FillOpacity);
                            gpr.FillText(x, y, text, fnt, fillColour, baseline, tag: tag);
                        }

                        if (currAttributes.Stroke != null)
                        {
                            Brush strokeColour = currAttributes.Stroke.MultiplyOpacity(currAttributes.Opacity * currAttributes.StrokeOpacity);
                            gpr.StrokeText(x, y, text, fnt, strokeColour, baseline, currAttributes.StrokeThickness, currAttributes.LineCap, currAttributes.LineJoin, currAttributes.LineDash, tag: tag);
                        }
                    }

                    x = endX;
                }

                if (hadClippingPath)
                {
                    gpr.Restore();
                }

                if (currAttributes.NeedsRestore)
                {
                    gpr.Restore();
                }
            }
        }

        private static FontFamily GetBoldFontFamily(FontFamily fontFamily)
        {
            switch (fontFamily.FileName)
            {
                case "Times-Roman":
                case "Times-Bold":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesBold);
                case "Times-Italic":
                case "Times-BoldItalic":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesBoldItalic);
                case "Helvetica":
                case "Helvetica-Bold":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBold);
                case "Helvetica-Oblique":
                case "Helvetica-BoldOblique":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBoldOblique);
                case "Courier":
                case "Courier-Bold":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBold);
                case "Courier-Oblique":
                case "Courier-BoldOblique":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBoldOblique);
                default:
                    return fontFamily;
            }
        }

        private static FontFamily GetItalicFontFamily(FontFamily fontFamily)
        {
            switch (fontFamily.FileName)
            {
                case "Times-Roman":
                case "Times-Italic":
```

```
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesItalic);
                case "Times-Bold":
                case "Times-BoldItalic":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesBoldItalic);
                case "Helvetica":
                case "Helvetica-Oblique":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaOblique);
                case "Helvetica-Bold":
                case "Helvetica-BoldOblique":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBoldOblique);
                case "Courier":
                case "Courier-Oblique":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierOblique);
                case "Courier-Bold":
                case "Courier-BoldOblique":
                    return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBoldOblique);
                default:
                    return fontFamily;
            }
        }

        private static FontFamily ParseFontFamily(string fontFamily, Dictionary<string, FontFamily> embeddedFonts)
        {
            string[] fontFamilies = Regexes.FontFamilySeparator.Split(fontFamily);

            foreach (string fam in fontFamilies)
            {
                string family = fam.Trim().Trim(',', '"').Trim();

                if (embeddedFonts.TryGetValue(family, out FontFamily tbr))
                {
                    return tbr;
                }

                List<(string, int)> matchedFamilies = new List<(string, int)>();

                for (int i = 0; i < FontFamily.StandardFamilies.Length; i++)
                {
                    if (family.StartsWith(FontFamily.StandardFamilies[i]))
                    {
                        matchedFamilies.Add((FontFamily.StandardFamilies[i], FontFamily.StandardFamilies[i].Length));
                    }
                }

                if (matchedFamilies.Count > 0)
                {
                    return FontFamily.ResolveFontFamily((from el in matchedFamilies orderby el.Item2 descending select el.Item1).First());
                }
                else
                {
                    if (family.Equals("serif", StringComparison.OrdinalIgnoreCase))
                    {
                        return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesRoman);
                    }
                    else if (family.Equals("sans-serif", StringComparison.OrdinalIgnoreCase))
                    {
                        return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica);
                    }
                    else if (family.Equals("monospace", StringComparison.OrdinalIgnoreCase))
                    {
                        return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Courier);
                    }
                    else if (family.Equals("cursive", StringComparison.OrdinalIgnoreCase))
                    {
                        return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesItalic);
                    }
                    else if (family.Equals("system-ui", StringComparison.OrdinalIgnoreCase))
                    {
                        return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica);
                    }
                    else if (family.Equals("ui-serif", StringComparison.OrdinalIgnoreCase))
                    {
                        return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesRoman);
                    }
```

```
                    else if (family.Equals("ui-sans-serif", StringComparison.OrdinalIgnoreCase))
                    {
                        return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica);
                    }
                    else if (family.Equals("ui-monospace", StringComparison.OrdinalIgnoreCase))
                    {
                        return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Courier);
                    }
                    else if (family.Equals("StandardSymbolsPS", StringComparison.OrdinalIgnoreCase))
                    {
                        return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Symbol);
                    }
                    else if (family.Equals("D050000L", StringComparison.OrdinalIgnoreCase))
                    {
                        return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.ZapfDingbats);
                    }
                }
            }

            return FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica);
        }

        private static void InterpretGObject(XmlNode currObject, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients, Dictionary<string, IFilter> filters)
        {
            PresentationAttributes currAttributes = InterpretPresentationAttributes(currObject, attributes, width, height, diagonal, gpr, styleSheets, gradients);

            bool hadClippingPath = ApplyClipPath(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);

            string filter = currObject.Attributes?["filter"]?.Value ?? currObject.Attributes?["mask"]?.Value;

            if (!string.IsNullOrEmpty(filter) && filter.StartsWith("url(#"))
            {
                filter = filter.Substring(5, filter.Length - 6);
            }

            if (!string.IsNullOrEmpty(filter) && filters.ContainsKey(filter))
            {
                Graphics filteredGraphics = new Graphics();

                InterpretSVGChildren(currObject, filteredGraphics, currAttributes, width, height, diagonal, styleSheets, gradients, filters);
                gpr.DrawGraphics(0, 0, filteredGraphics, filters[filter]);
            }
            else
            {
                InterpretSVGChildren(currObject, gpr, currAttributes, width, height, diagonal, styleSheets, gradients, filters);
            }

            if (hadClippingPath)
            {
                gpr.Restore();
            }

            if (currAttributes.NeedsRestore)
            {
                gpr.Restore();
            }
        }

        private static void InterpretUseObject(XmlNode currObject, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients, Dictionary<string, IFilter> filters)
        {
            double x, y, w, h;

            x = ParseLengthOrPercentage(currObject.Attributes?["x"]?.Value, width);
            y = ParseLengthOrPercentage(currObject.Attributes?["y"]?.Value, height);
            w = ParseLengthOrPercentage(currObject.Attributes?["width"]?.Value, width, double.NaN);
            h = ParseLengthOrPercentage(currObject.Attributes?["height"]?.Value, height, double.NaN);

            string id = currObject.Attributes?["href"]?.Value ?? currObject.Attributes?["xlink:href"]?.Value;

            if (id != null && id.StartsWith("#"))
            {
                id = id.Substring(1);

                XmlNode element = currObject.OwnerDocument.SelectSingleNode(string.Format("//*[@id='{0}']", id));
```

```
                if (element != null)
                {
                    XmlNode clone = element.Clone();

                    currObject.AppendChild(clone);


                    PresentationAttributes currAttributes = InterpretPresentationAttributes(currObject, attributes, width, height, diagonal, gpr, styleSheets, gradients);


                    gpr.Save();
                    gpr.Translate(x, y);

                    ((XmlElement)clone).SetAttribute("x", "0");
                    ((XmlElement)clone).SetAttribute("y", "0");

                    if (clone.Attributes?["viewBox"] != null)
                    {
                        ((XmlElement)clone).SetAttribute("width", w.ToString(System.Globalization.CultureInfo.InvariantCulture));
                        ((XmlElement)clone).SetAttribute("height", h.ToString(System.Globalization.CultureInfo.InvariantCulture));
                    }

                    InterpretSVGElement(clone, gpr, currAttributes, width, height, diagonal, styleSheets, gradients, filters);

                    gpr.Restore();

                    if (currAttributes.NeedsRestore)
                    {
                        gpr.Restore();
                    }
                }
            }
        }

        private static bool ApplyClipPath(XmlNode currObject, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients)
        {
            string id = currObject.Attributes?["clip-path"]?.Value;

            if (id != null && id.StartsWith("url(#"))
            {
                id = id.Substring(5);
                id = id.Substring(0, id.Length - 1);

                XmlNode element = currObject.OwnerDocument.SelectSingleNode(string.Format("//*[@id='{0}']", id));

                if (element != null && element.ChildNodes.Count == 1 && element.ChildNodes[0].Name.Equals("path", StringComparison.OrdinalIgnoreCase))
                {
                    bool hasParentClipPath = ApplyClipPath(element, gpr, width, height, diagonal, attributes, styleSheets, gradients);

                    Graphics pathGraphics = new Graphics();
                    InterpretPathObject(element.ChildNodes[0], pathGraphics, width, height, diagonal, attributes, styleSheets, gradients);

                    PathTransformerGraphicsContext ptgc = new PathTransformerGraphicsContext();
                    pathGraphics.CopyToIGraphicsContext(ptgc);

                    if (!hasParentClipPath)
                    {
                        gpr.Save();
                    }

                    gpr.SetClippingPath(ptgc.CurrentPath);

                    return true;
                }

                return false;
            }
            else
            {
                return false;
            }
        }

        private static void InterpretRectObject(XmlNode currObject, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets,
```

```
Dictionary<string, Brush> gradients)
        {
            double x, y, w, h, rx, ry;

            x = ParseLengthOrPercentage(currObject.Attributes?["x"]?.Value, width);
            y = ParseLengthOrPercentage(currObject.Attributes?["y"]?.Value, height);
            w = ParseLengthOrPercentage(currObject.Attributes?["width"]?.Value, width);
            h = ParseLengthOrPercentage(currObject.Attributes?["height"]?.Value, height);
            rx = ParseLengthOrPercentage(currObject.Attributes?["rx"]?.Value, width, double.NaN);
            ry = ParseLengthOrPercentage(currObject.Attributes?["ry"]?.Value, height, double.NaN);

            if (w > 0 && h > 0)
            {
                if (double.IsNaN(rx) && !double.IsNaN(ry))
                {
                    rx = ry;
                }
                else if (!double.IsNaN(rx) && double.IsNaN(ry))
                {
                    ry = rx;
                }

                if (double.IsNaN(rx))
                {
                    rx = 0;
                }

                if (double.IsNaN(ry))
                {
                    ry = 0;
                }

                rx = Math.Min(rx, w / 2);
                ry = Math.Min(ry, h / 2);

                GraphicsPath path = new GraphicsPath();

                path.MoveTo(x + rx, y);
                path.LineTo(x + w - rx, y);

                if (rx > 0 && ry > 0)
                {
                    path.EllipticalArc(rx, ry, 0, false, true, new Point(x + w, y + ry));
                }

                path.LineTo(x + w, y + h - ry);

                if (rx > 0 && ry > 0)
                {
                    path.EllipticalArc(rx, ry, 0, false, true, new Point(x + w - rx, y + h));
                }

                path.LineTo(x + rx, y + h);

                if (rx > 0 && ry > 0)
                {
                    path.EllipticalArc(rx, ry, 0, false, true, new Point(x, y + h - ry));
                }

                path.LineTo(x, y + ry);

                if (rx > 0 && ry > 0)
                {
                    path.EllipticalArc(rx, ry, 0, false, true, new Point(x + rx, y));
                }

                path.Close();

                PresentationAttributes currAttributes = InterpretPresentationAttributes(currObject,
attributes, width, height, diagonal, gpr, styleSheets, gradients);

                bool hadClippingPath = ApplyClipPath(currObject, gpr, width, height, diagonal,
attributes, styleSheets, gradients);

                string tag = currObject.Attributes?["id"]?.Value;

                if (currAttributes.StrokeFirst)
                {
                    if (currAttributes.Stroke != null)
                    {
                        Brush strokeColour =
currAttributes.Stroke.MultiplyOpacity(currAttributes.Opacity * currAttributes.StrokeOpacity);
                        gpr.StrokePath(path, strokeColour, currAttributes.StrokeThickness,
currAttributes.LineCap, currAttributes.LineJoin, currAttributes.LineDash, tag: tag);
                    }

                    if (currAttributes.Fill != null)
```

```
                    {
                        Brush fillColour = currAttributes.Fill.MultiplyOpacity(currAttributes.Opacity * currAttributes.FillOpacity);
                        gpr.FillPath(path, fillColour, tag: tag);
                    }
                }
                else
                {
                    if (currAttributes.Fill != null)
                    {
                        Brush fillColour = currAttributes.Fill.MultiplyOpacity(currAttributes.Opacity * currAttributes.FillOpacity);
                        gpr.FillPath(path, fillColour, tag: tag);
                    }

                    if (currAttributes.Stroke != null)
                    {
                        Brush strokeColour = currAttributes.Stroke.MultiplyOpacity(currAttributes.Opacity * currAttributes.StrokeOpacity);
                        gpr.StrokePath(path, strokeColour, currAttributes.StrokeThickness, currAttributes.LineCap, currAttributes.LineJoin, currAttributes.LineDash, tag: tag);
                    }
                }

                if (hadClippingPath)
                {
                    gpr.Restore();
                }

                if (currAttributes.NeedsRestore)
                {
                    gpr.Restore();
                }
            }
        }

        private static void InterpretPolyLineObject(XmlNode currObject, bool isPolygon, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients)
        {
            string points = currObject.Attributes?["points"]?.Value;

            if (points != null)
            {
                double[] coordinates = ParseListOfDoubles(points);

                GraphicsPath path = new GraphicsPath();

                for (int i = 0; i < coordinates.Length; i += 2)
                {
                    path.LineTo(coordinates[i], coordinates[i + 1]);
                }

                if (isPolygon)
                {
                    path.Close();
                }

                PresentationAttributes currAttributes = InterpretPresentationAttributes(currObject, attributes, width, height, diagonal, gpr, styleSheets, gradients);

                bool hadClippingPath = ApplyClipPath(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);

                string tag = currObject.Attributes?["id"]?.Value;

                if (currAttributes.StrokeFirst)
                {
                    if (currAttributes.Stroke != null)
                    {
                        Brush strokeColour = currAttributes.Stroke.MultiplyOpacity(currAttributes.Opacity * currAttributes.StrokeOpacity);
                        gpr.StrokePath(path, strokeColour, currAttributes.StrokeThickness, currAttributes.LineCap, currAttributes.LineJoin, currAttributes.LineDash, tag: tag);
                    }

                    if (currAttributes.Fill != null)
                    {
                        Brush fillColour = currAttributes.Fill.MultiplyOpacity(currAttributes.Opacity * currAttributes.FillOpacity);
                        gpr.FillPath(path, fillColour, tag: tag);
                    }
                }
                else
                {
                    if (currAttributes.Fill != null)
                    {
```

```
                        Brush fillColour = currAttributes.Fill.MultiplyOpacity(currAttributes.Opacity * currAttributes.FillOpacity);
                        gpr.FillPath(path, fillColour, tag: tag);
                    }

                    if (currAttributes.Stroke != null)
                    {
                        Brush strokeColour = currAttributes.Stroke.MultiplyOpacity(currAttributes.Opacity * currAttributes.StrokeOpacity);
                        gpr.StrokePath(path, strokeColour, currAttributes.StrokeThickness, currAttributes.LineCap, currAttributes.LineJoin, currAttributes.LineDash, tag: tag);
                    }
                }

                if (hadClippingPath)
                {
                    gpr.Restore();
                }

                if (currAttributes.NeedsRestore)
                {
                    gpr.Restore();
                }
            }
        }

        private static void InterpretPathObject(XmlNode currObject, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients)
        {
            string d = currObject.Attributes?["d"]?.Value;

            if (d != null)
            {
                List<string> pathData = TokenisePathData(d);

                GraphicsPath path = new GraphicsPath();

                Point lastPoint = new Point();
                Point? figureStartPoint = null;

                char lastCommand = '\0';
                Point lastCtrlPoint = new Point();

                for (int i = 0; i < pathData.Count; i++)
                {
                    Point delta = new Point();

                    bool isAbsolute = char.IsUpper(pathData[i][0]);

                    if (!isAbsolute)
                    {
                        delta = lastPoint;
                    }

                    switch (pathData[i][0])
                    {
                        case 'M':
                        case 'm':
                            lastPoint = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                            path.MoveTo(lastPoint);
                            figureStartPoint = lastPoint;
                            i += 2;
                            lastCommand = 'M';
                            while (i < pathData.Count - 1 && !char.IsLetter(pathData[i + 1][0]))
                            {
                                if (!isAbsolute)
                                {
                                    delta = lastPoint;
                                }
                                else
                                {
                                    delta = new Point();
                                }

                                lastPoint = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                                path.LineTo(lastPoint);

                                i += 2;
                                lastCommand = 'L';
                            }
                            break;
                        case 'L':
```

```
                        case 'l':
                            lastPoint = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                            path.LineTo(lastPoint);
                            if (figureStartPoint == null)
                            {
                                figureStartPoint = lastPoint;
                            }
                            i += 2;
                            lastCommand = 'L';
                            while (i < pathData.Count - 1 && !char.IsLetter(pathData[i + 1][0]))
                            {
                                if (!isAbsolute)
                                {
                                    delta = lastPoint;
                                }
                                else
                                {
                                    delta = new Point();
                                }

                                lastPoint = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                                path.LineTo(lastPoint);

                                i += 2;
                            }
                            break;
                        case 'H':
                        case 'h':
                            lastPoint = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), lastPoint.Y);
                            path.LineTo(lastPoint);
                            if (figureStartPoint == null)
                            {
                                figureStartPoint = lastPoint;
                            }
                            i++;
                            lastCommand = 'L';
                            while (i < pathData.Count - 1 && !char.IsLetter(pathData[i + 1][0]))
                            {
                                if (!isAbsolute)
                                {
                                    delta = lastPoint;
                                }
                                else
                                {
                                    delta = new Point();
                                }

                                lastPoint = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), lastPoint.Y);
                                path.LineTo(lastPoint);

                                i++;
                            }
                            break;
                        case 'V':
                        case 'v':
                            lastPoint = new Point(lastPoint.X, delta.Y + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture));
                            path.LineTo(lastPoint);
                            if (figureStartPoint == null)
                            {
                                figureStartPoint = lastPoint;
                            }
                            i++;
                            lastCommand = 'L';
                            while (i < pathData.Count - 1 && !char.IsLetter(pathData[i + 1][0]))
                            {
                                if (!isAbsolute)
                                {
                                    delta = lastPoint;
                                }
                                else
                                {
                                    delta = new Point();
                                }

                                lastPoint = new Point(lastPoint.X, delta.Y + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture));
                                path.LineTo(lastPoint);

                                i++;
                            }
```

```
                        break;
                    case 'C':
                    case 'c':
                        {
                            Point ctrlPoint1 = new Point(delta.X + double.Parse(pathData[i + 1],
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                            i += 2;

                            Point ctrlPoint2 = new Point(delta.X + double.Parse(pathData[i + 1],
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                            i += 2;

                            lastPoint = new Point(delta.X + double.Parse(pathData[i + 1],
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                            i += 2;

                            if (figureStartPoint == null)
                            {
                                figureStartPoint = lastPoint;
                            }

                            path.CubicBezierTo(ctrlPoint1, ctrlPoint2, lastPoint);

                            lastCtrlPoint = ctrlPoint2;
                            lastCommand = 'C';

                            while (i < pathData.Count - 1 && !char.IsLetter(pathData[i + 1][0]))
                            {
                                if (!isAbsolute)
                                {
                                    delta = lastPoint;
                                }
                                else
                                {
                                    delta = new Point();
                                }

                                ctrlPoint1 = new Point(delta.X + double.Parse(pathData[i + 1],
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                                i += 2;

                                ctrlPoint2 = new Point(delta.X + double.Parse(pathData[i + 1],
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                                i += 2;

                                lastPoint = new Point(delta.X + double.Parse(pathData[i + 1],
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                                i += 2;

                                path.CubicBezierTo(ctrlPoint1, ctrlPoint2, lastPoint);
                                lastCtrlPoint = ctrlPoint2;
                            }
                        }
                        break;
                    case 'S':
                    case 's':
                        {
                            Point ctrlPoint1;

                            if (lastCommand == 'C')
                            {
                                ctrlPoint1 = new Point(2 * lastPoint.X - lastCtrlPoint.X, 2 *
lastPoint.Y - lastCtrlPoint.Y);
                            }
                            else
                            {
                                ctrlPoint1 = lastPoint;
                            }

                            Point ctrlPoint2 = new Point(delta.X + double.Parse(pathData[i + 1],
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                            i += 2;

                            lastPoint = new Point(delta.X + double.Parse(pathData[i + 1],
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                            i += 2;

                            if (figureStartPoint == null)
                            {
```

```
                                    figureStartPoint = lastPoint;
                                }

                                path.CubicBezierTo(ctrlPoint1, ctrlPoint2, lastPoint);

                                lastCtrlPoint = ctrlPoint2;
                                lastCommand = 'C';

                                while (i < pathData.Count - 1 && !char.IsLetter(pathData[i + 1][0]))
                                {
                                    if (!isAbsolute)
                                    {
                                        delta = lastPoint;
                                    }
                                    else
                                    {
                                        delta = new Point();
                                    }

                                    ctrlPoint1 = new Point(2 * lastPoint.X - lastCtrlPoint.X, 2 * lastPoint.Y - lastCtrlPoint.Y);

                                    ctrlPoint2 = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                                    i += 2;

                                    lastPoint = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                                    i += 2;

                                    path.CubicBezierTo(ctrlPoint1, ctrlPoint2, lastPoint);

                                    lastCtrlPoint = ctrlPoint2;
                                }
                            }
                            break;
                        case 'Q':
                        case 'q':
                            {
                                Point ctrlPoint = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                                i += 2;

                                Point actualCP1 = new Point(lastPoint.X + 2 * (ctrlPoint.X - lastPoint.X) / 3, lastPoint.Y + 2 * (ctrlPoint.Y - lastPoint.Y) / 3);

                                lastPoint = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                                i += 2;

                                Point actualCP2 = new Point(lastPoint.X + 2 * (ctrlPoint.X - lastPoint.X) / 3, lastPoint.Y + 2 * (ctrlPoint.Y - lastPoint.Y) / 3);

                                if (figureStartPoint == null)
                                {
                                    figureStartPoint = lastPoint;
                                }

                                path.CubicBezierTo(actualCP1, actualCP2, lastPoint);

                                lastCtrlPoint = ctrlPoint;
                                lastCommand = 'Q';

                                while (i < pathData.Count - 1 && !char.IsLetter(pathData[i + 1][0]))
                                {
                                    if (!isAbsolute)
                                    {
                                        delta = lastPoint;
                                    }
                                    else
                                    {
                                        delta = new Point();
                                    }

                                    ctrlPoint = new Point(delta.X + double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                                    i += 2;

                                    actualCP1 = new Point(lastPoint.X + 2 * (ctrlPoint.X - lastPoint.X) / 3, lastPoint.Y + 2 * (ctrlPoint.Y - lastPoint.Y) / 3);

                                    lastPoint = new Point(delta.X + double.Parse(pathData[i + 1],
```

```
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                                    i += 2;

                                    actualCP2 = new Point(lastPoint.X + 2 * (ctrlPoint.X -
lastPoint.X) / 3, lastPoint.Y + 2 * (ctrlPoint.Y - lastPoint.Y) / 3);

                                    path.CubicBezierTo(actualCP1, actualCP2, lastPoint);
                                    lastCtrlPoint = ctrlPoint;
                                }


                            }
                            break;
                        case 'T':
                        case 't':
                            {
                                Point ctrlPoint;

                                if (lastCommand == 'Q')
                                {
                                    ctrlPoint = new Point(2 * lastPoint.X - lastCtrlPoint.X, 2 *
lastPoint.Y - lastCtrlPoint.Y);
                                }
                                else
                                {
                                    ctrlPoint = lastPoint;
                                }

                                Point actualCP1 = new Point(lastPoint.X + 2 * (ctrlPoint.X -
lastPoint.X) / 3, lastPoint.Y + 2 * (ctrlPoint.Y - lastPoint.Y) / 3);

                                lastPoint = new Point(delta.X + double.Parse(pathData[i + 1],
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                                i += 2;

                                Point actualCP2 = new Point(lastPoint.X + 2 * (ctrlPoint.X -
lastPoint.X) / 3, lastPoint.Y + 2 * (ctrlPoint.Y - lastPoint.Y) / 3);

                                if (figureStartPoint == null)
                                {
                                    figureStartPoint = lastPoint;
                                }

                                path.CubicBezierTo(actualCP1, actualCP2, lastPoint);
                                lastCtrlPoint = ctrlPoint;
                                lastCommand = 'Q';

                                while (i < pathData.Count - 1 && !char.IsLetter(pathData[i + 1][0]))
                                {
                                    if (!isAbsolute)
                                    {
                                        delta = lastPoint;
                                    }
                                    else
                                    {
                                        delta = new Point();
                                    }

                                    ctrlPoint = new Point(2 * lastPoint.X - lastCtrlPoint.X, 2 *
lastPoint.Y - lastCtrlPoint.Y);

                                    actualCP1 = new Point(lastPoint.X + 2 * (ctrlPoint.X -
lastPoint.X) / 3, lastPoint.Y + 2 * (ctrlPoint.Y - lastPoint.Y) / 3);

                                    lastPoint = new Point(delta.X + double.Parse(pathData[i + 1],
System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 2],
System.Globalization.CultureInfo.InvariantCulture));
                                    i += 2;

                                    actualCP2 = new Point(lastPoint.X + 2 * (ctrlPoint.X -
lastPoint.X) / 3, lastPoint.Y + 2 * (ctrlPoint.Y - lastPoint.Y) / 3);

                                    path.CubicBezierTo(actualCP1, actualCP2, lastPoint);

                                    lastCtrlPoint = ctrlPoint;
                                }
                            }
                            break;
                        case 'A':
                        case 'a':
                            {
                                Point startPoint = lastPoint;

                                if (figureStartPoint == null)
                                {
```

```
                                    figureStartPoint = lastPoint;
                                }

                                Point radii = new Point(double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                                double angle = double.Parse(pathData[i + 3], System.Globalization.CultureInfo.InvariantCulture) * Math.PI / 180;
                                bool largeArcFlag = pathData[i + 4][0] == '1';
                                bool sweepFlag = pathData[i + 5][0] == '1';

                                lastPoint = new Point(delta.X + double.Parse(pathData[i + 6], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 7], System.Globalization.CultureInfo.InvariantCulture));
                                i += 7;

                                path.EllipticalArc(radii.X, radii.Y, angle, largeArcFlag, sweepFlag, lastPoint);

                                while (i < pathData.Count - 1 && !char.IsLetter(pathData[i + 1][0]))
                                {
                                    if (!isAbsolute)
                                    {
                                        delta = lastPoint;
                                    }
                                    else
                                    {
                                        delta = new Point();
                                    }

                                    startPoint = lastPoint;
                                    radii = new Point(double.Parse(pathData[i + 1], System.Globalization.CultureInfo.InvariantCulture), double.Parse(pathData[i + 2], System.Globalization.CultureInfo.InvariantCulture));
                                    angle = double.Parse(pathData[i + 3], System.Globalization.CultureInfo.InvariantCulture) * Math.PI / 180;
                                    largeArcFlag = pathData[i + 4][0] == '1';
                                    sweepFlag = pathData[i + 5][0] == '1';

                                    lastPoint = new Point(delta.X + double.Parse(pathData[i + 6], System.Globalization.CultureInfo.InvariantCulture), delta.Y + double.Parse(pathData[i + 7], System.Globalization.CultureInfo.InvariantCulture));
                                    i += 7;

                                    path.EllipticalArc(radii.X, radii.Y, angle, largeArcFlag, sweepFlag, lastPoint);
                                }
                            }

                            break;
                        case 'Z':
                        case 'z':
                            path.Close();
                            lastPoint = figureStartPoint.Value;
                            figureStartPoint = null;
                            lastCommand = 'Z';
                            break;
                    }
                }

                PresentationAttributes currAttributes = InterpretPresentationAttributes(currObject, attributes, width, height, diagonal, gpr, styleSheets, gradients);

                bool hadClippingPath = ApplyClipPath(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);

                string tag = currObject.Attributes?["id"]?.Value;

                if (currAttributes.StrokeFirst)
                {
                    if (currAttributes.Stroke != null)
                    {
                        Brush strokeColour = currAttributes.Stroke.MultiplyOpacity(currAttributes.Opacity * currAttributes.StrokeOpacity);
                        gpr.StrokePath(path, strokeColour, currAttributes.StrokeThickness, currAttributes.LineCap, currAttributes.LineJoin, currAttributes.LineDash, tag: tag);
                    }

                    if (currAttributes.Fill != null)
                    {
                        Brush fillColour = currAttributes.Fill.MultiplyOpacity(currAttributes.Opacity * currAttributes.FillOpacity);
                        gpr.FillPath(path, fillColour, tag: tag);
                    }
                }
                else
                {
```

```
                    if (currAttributes.Fill != null)
                    {
                        Brush fillColour = currAttributes.Fill.MultiplyOpacity(currAttributes.Opacity * currAttributes.FillOpacity);
                        gpr.FillPath(path, fillColour, tag: tag);
                    }

                    if (currAttributes.Stroke != null)
                    {
                        Brush strokeColour = currAttributes.Stroke.MultiplyOpacity(currAttributes.Opacity * currAttributes.StrokeOpacity);
                        gpr.StrokePath(path, strokeColour, currAttributes.StrokeThickness, currAttributes.LineCap, currAttributes.LineJoin, currAttributes.LineDash, tag: tag);
                    }
                }

                if (hadClippingPath)
                {
                    gpr.Restore();
                }

                if (currAttributes.NeedsRestore)
                {
                    gpr.Restore();
                }
            }

        }

        private static List<string> TokenisePathData(string d)
        {
            List<string> tbr = new List<string>();

            string currToken = "";

            for (int i = 0; i < d.Length; i++)
            {
                char c = d[i];

                if (c >= '0' && c <= '9' || c == '.' || c == 'e' || c == 'E')
                {
                    currToken += c;
                }
                else if (c == '-' || c == '+')
                {
                    if (i > 0 && (d[i - 1] == 'e' || d[i - 1] == 'E'))
                    {
                        currToken += c;
                    }
                    else
                    {
                        if (!string.IsNullOrEmpty(currToken))
                        {
                            tbr.Add(currToken);
                        }
                        currToken = "" + c;
                    }
                }
                else if (char.IsWhiteSpace(c) || c == ',')
                {
                    if (!string.IsNullOrEmpty(currToken))
                    {
                        tbr.Add(currToken);
                    }
                    currToken = "";
                }
                else if (i < d.Length - 2 && (c == 'N' || c == 'n') && (d[i + 1] == 'a' || d[i + 1] == 'A') && (d[i + 2] == 'N' || d[i + 2] == 'n'))
                {
                    if (!string.IsNullOrEmpty(currToken))
                    {
                        tbr.Add(currToken);
                    }
                    tbr.Add("NaN");
                    currToken = "";
                    i += 2;
                }
                else if ("MmLlHhVvCcSsQqTtAaZz".Contains(c))
                {
                    if (!string.IsNullOrEmpty(currToken))
                    {
                        tbr.Add(currToken);
                    }
                    tbr.Add(c.ToString());
                    currToken = "";
                }
            }
```

```
            if (!string.IsNullOrEmpty(currToken))
            {
                tbr.Add(currToken);
            }

            return tbr;
        }

        private static void InterpretCircleObject(XmlNode circleObject, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients)
        {
            double cx, cy, r;

            cx = ParseLengthOrPercentage(circleObject.Attributes?["cx"]?.Value, width);
            cy = ParseLengthOrPercentage(circleObject.Attributes?["cy"]?.Value, height);
            r = ParseLengthOrPercentage(circleObject.Attributes?["r"]?.Value, diagonal);

            string tag = circleObject.Attributes?["id"]?.Value;

            PresentationAttributes circleAttributes = InterpretPresentationAttributes(circleObject, attributes, width, height, diagonal, gpr, styleSheets, gradients);

            bool hadClippingPath = ApplyClipPath(circleObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);

            if (circleAttributes.StrokeFirst)
            {
                if (circleAttributes.Stroke != null)
                {
                    Brush strokeColour = circleAttributes.Stroke.MultiplyOpacity(circleAttributes.Opacity * circleAttributes.StrokeOpacity);
                    gpr.StrokePath(new GraphicsPath().Arc(cx, cy, r, 0, 2 * Math.PI).Close(), strokeColour, circleAttributes.StrokeThickness, circleAttributes.LineCap, circleAttributes.LineJoin, circleAttributes.LineDash, tag: tag);
                }

                if (circleAttributes.Fill != null)
                {
                    Brush fillColour = circleAttributes.Fill.MultiplyOpacity(circleAttributes.Opacity * circleAttributes.FillOpacity);
                    gpr.FillPath(new GraphicsPath().Arc(cx, cy, r, 0, 2 * Math.PI).Close(), fillColour, tag: tag);
                }
            }
            else
            {
                if (circleAttributes.Fill != null)
                {
                    Brush fillColour = circleAttributes.Fill.MultiplyOpacity(circleAttributes.Opacity * circleAttributes.FillOpacity);
                    gpr.FillPath(new GraphicsPath().Arc(cx, cy, r, 0, 2 * Math.PI).Close(), fillColour, tag: tag);
                }

                if (circleAttributes.Stroke != null)
                {
                    Brush strokeColour = circleAttributes.Stroke.MultiplyOpacity(circleAttributes.Opacity * circleAttributes.StrokeOpacity);
                    gpr.StrokePath(new GraphicsPath().Arc(cx, cy, r, 0, 2 * Math.PI).Close(), strokeColour, circleAttributes.StrokeThickness, circleAttributes.LineCap, circleAttributes.LineJoin, circleAttributes.LineDash, tag: tag);
                }
            }

            if (hadClippingPath)
            {
                gpr.Restore();
            }

            if (circleAttributes.NeedsRestore)
            {
                gpr.Restore();
            }
        }

        private static void InterpretEllipseObject(XmlNode currObject, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients)
        {
            double cx, cy, rx, ry;

            cx = ParseLengthOrPercentage(currObject.Attributes?["cx"]?.Value, width);
            cy = ParseLengthOrPercentage(currObject.Attributes?["cy"]?.Value, height);
            rx = ParseLengthOrPercentage(currObject.Attributes?["rx"]?.Value, width, double.NaN);
            ry = ParseLengthOrPercentage(currObject.Attributes?["ry"]?.Value, height, double.NaN);
```

```

            string tag = currObject.Attributes?["id"]?.Value;

            if (double.IsNaN(rx) && !double.IsNaN(ry))
            {
                rx = ry;
            }
            else if (!double.IsNaN(rx) && double.IsNaN(ry))
            {
                ry = rx;
            }

            if (rx > 0 && ry > 0)
            {

                PresentationAttributes currAttributes = InterpretPresentationAttributes(currObject, attributes, width, height, diagonal, gpr, styleSheets, gradients);

                bool hadClippingPath = ApplyClipPath(currObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);

                double r = Math.Min(rx, ry);

                gpr.Save();
                gpr.Translate(cx, cy);
                gpr.Scale(rx / r, ry / r);

                if (currAttributes.StrokeFirst)
                {
                    if (currAttributes.Stroke != null)
                    {
                        Brush strokeColour = currAttributes.Stroke.MultiplyOpacity(currAttributes.Opacity * currAttributes.StrokeOpacity);
                        gpr.StrokePath(new GraphicsPath().Arc(0, 0, r, 0, 2 * Math.PI).Close(), strokeColour, currAttributes.StrokeThickness, currAttributes.LineCap, currAttributes.LineJoin, currAttributes.LineDash, tag: tag);
                    }

                    if (currAttributes.Fill != null)
                    {
                        Brush fillColour = currAttributes.Fill.MultiplyOpacity(currAttributes.Opacity * currAttributes.FillOpacity);
                        gpr.FillPath(new GraphicsPath().Arc(0, 0, r, 0, 2 * Math.PI).Close(), fillColour, tag: tag);
                    }
                }
                else
                {
                    if (currAttributes.Fill != null)
                    {
                        Brush fillColour = currAttributes.Fill.MultiplyOpacity(currAttributes.Opacity * currAttributes.FillOpacity);
                        gpr.FillPath(new GraphicsPath().Arc(0, 0, r, 0, 2 * Math.PI).Close(), fillColour, tag: tag);
                    }

                    if (currAttributes.Stroke != null)
                    {
                        Brush strokeColour = currAttributes.Stroke.MultiplyOpacity(currAttributes.Opacity * currAttributes.StrokeOpacity);
                        gpr.StrokePath(new GraphicsPath().Arc(0, 0, r, 0, 2 * Math.PI).Close(), strokeColour, currAttributes.StrokeThickness, currAttributes.LineCap, currAttributes.LineJoin, currAttributes.LineDash, tag: tag);
                    }
                }

                gpr.Restore();

                if (hadClippingPath)
                {
                    gpr.Restore();
                }

                if (currAttributes.NeedsRestore)
                {
                    gpr.Restore();
                }
            }
        }

        private static void InterpretLineObject(XmlNode lineObject, Graphics gpr, double width, double height, double diagonal, PresentationAttributes attributes, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients)
        {
            double x1, x2, y1, y2;

            x1 = ParseLengthOrPercentage(lineObject.Attributes?["x1"]?.Value, width);
```

```
            y1 = ParseLengthOrPercentage(lineObject.Attributes?["y1"]?.Value, height);
            x2 = ParseLengthOrPercentage(lineObject.Attributes?["x2"]?.Value, width);
            y2 = ParseLengthOrPercentage(lineObject.Attributes?["y2"]?.Value, height);

            PresentationAttributes lineAttributes = InterpretPresentationAttributes(lineObject, attributes, width, height, diagonal, gpr, styleSheets, gradients);

            bool hadClippingPath = ApplyClipPath(lineObject, gpr, width, height, diagonal, attributes, styleSheets, gradients);

            string tag = lineObject.Attributes?["id"]?.Value;

            if (lineAttributes.Stroke != null)
            {
                Brush strokeColour = lineAttributes.Stroke.MultiplyOpacity(lineAttributes.Opacity * lineAttributes.StrokeOpacity);
                gpr.StrokePath(new GraphicsPath().MoveTo(x1, y1).LineTo(x2, y2), strokeColour, lineAttributes.StrokeThickness, lineAttributes.LineCap, lineAttributes.LineJoin, lineAttributes.LineDash, tag: tag);
            }

            if (hadClippingPath)
            {
                gpr.Restore();
            }

            if (lineAttributes.NeedsRestore)
            {
                gpr.Restore();
            }
        }

        private static void SetStyleAttributes(XmlNode obj, IEnumerable<Stylesheet> styleSheets)
        {
            string style = obj.Attributes?["style"]?.Value;

            string classes = obj.Attributes?["class"]?.Value;

            if (!string.IsNullOrEmpty(classes))
            {
                string[] splitClasses = classes.Split(' ');

                foreach (string className in splitClasses)
                {
                    if (!string.IsNullOrEmpty(className.Trim()))
                    {
                        foreach (Stylesheet sheet in styleSheets)
                        {
                            foreach (StyleRule rule in sheet.StyleRules)
                            {
                                if (rule.SelectorText.Contains("." + className))
                                {
                                    style = rule.Style.CssText + "; " + style;
                                }
                            }
                        }
                    }
                }
            }

            if (!string.IsNullOrEmpty(style))
            {
                string[] splitStyle = style.Split(';');

                for (int i = 0; i < splitStyle.Length; i++)
                {
                    string[] styleCouple = splitStyle[i].Split(':');

                    if (styleCouple.Length == 2)
                    {
                        string styleName = styleCouple[0].Trim();
                        string styleValue = styleCouple[1].Trim();

                        ((XmlElement)obj).SetAttribute(styleName, styleValue);
                    }
                    else if (!string.IsNullOrWhiteSpace(splitStyle[i]))
                    {
                        throw new InvalidOperationException("The style specification is not valid: " + splitStyle[i]);
                    }
                }
            }
        }

        internal static PresentationAttributes InterpretPresentationAttributes(XmlNode obj, PresentationAttributes parentPresentationAttributes, double width, double height, double diagonal, Graphics gpr, IEnumerable<Stylesheet> styleSheets, Dictionary<string, Brush> gradients)
```

```
        {
            SetStyleAttributes(obj, styleSheets);

            PresentationAttributes tbr = parentPresentationAttributes.Clone();

            string stroke = obj.Attributes?["stroke"]?.Value;
            string strokeOpacity = obj.Attributes?["stroke-opacity"]?.Value;
            string fill = obj.Attributes?["fill"]?.Value;
            string fillOpacity = obj.Attributes?["fill-opacity"]?.Value;
            string currentColour = obj.Attributes?["colour"]?.Value;
            string strokeThickness = obj.Attributes?["stroke-width"]?.Value;
            string lineCap = obj.Attributes?["stroke-linecap"]?.Value;
            string lineJoin = obj.Attributes?["stroke-linejoin"]?.Value;
            string opacity = obj.Attributes?["opacity"]?.Value;
            string strokeDashArray = obj.Attributes?["stroke-dasharray"]?.Value;
            string strokeDashOffset = obj.Attributes?["stroke-dashoffset"]?.Value;
            string paintOrder = obj.Attributes?["paint-order"]?.Value;

            string xA = obj.Attributes?["x"]?.Value;
            string yA = obj.Attributes?["y"]?.Value;
            string wA = obj.Attributes?["width"]?.Value;
            string hA = obj.Attributes?["height"]?.Value;

            string transform = obj.Attributes?["transform"]?.Value;

            if (xA != null)
            {
                tbr.X = ParseLengthOrPercentage(xA, width);
            }

            if (yA != null)
            {
                tbr.Y = ParseLengthOrPercentage(yA, height);
            }

            if (wA != null)
            {
                tbr.Width = ParseLengthOrPercentage(wA, width);
            }

            if (hA != null)
            {
                tbr.Height = ParseLengthOrPercentage(hA, height);
            }

            if (stroke != null)
            {
                if (stroke.Trim().StartsWith("url("))
                {
                    string url = stroke.Trim().Substring(4);
                    if (url.EndsWith(")"))
                    {
                        url = url.Substring(0, url.Length - 1);
                    }

                    url = url.Trim();

                    if (url.StartsWith("#"))
                    {
                        url = url.Substring(1);
                        if (gradients.TryGetValue(url, out Brush brush))
                        {
                            tbr.Stroke = brush;
                        }
                    }
                    else
                    {
                        tbr.Stroke = null;
                    }
                }
                else
                {
                    tbr.Stroke = Colour.FromCSSString(stroke);
                }
            }

            if (strokeOpacity != null)
            {
                tbr.StrokeOpacity = ParseLengthOrPercentage(strokeOpacity, 1);
            }

            if (fill != null)
            {
                if (fill.Trim().StartsWith("url("))
                {
                    string url = fill.Trim().Substring(4);
                    if (url.EndsWith(")"))
```

```
                    {
                        url = url.Substring(0, url.Length - 1);
                    }

                    url = url.Trim();

                    if (url.StartsWith("#"))
                    {
                        url = url.Substring(1);
                        if (gradients.TryGetValue(url, out Brush brush))
                        {
                            tbr.Fill = brush;
                        }
                    }
                    else
                    {
                        tbr.Fill = null;
                    }
                }
                else
                {
                    tbr.Fill = Colour.FromCSSString(fill);
                }
            }

            if (fillOpacity != null)
            {
                tbr.FillOpacity = ParseLengthOrPercentage(fillOpacity, 1);
            }

            if (currentColour != null)
            {
                tbr.CurrentColour = Colour.FromCSSString(currentColour);
            }

            if (strokeThickness != null)
            {
                tbr.StrokeThickness = ParseLengthOrPercentage(strokeThickness, diagonal);
            }

            if (lineCap != null)
            {
                if (lineCap.Equals("butt", StringComparison.OrdinalIgnoreCase))
                {
                    tbr.LineCap = LineCaps.Butt;
                }
                else if (lineCap.Equals("round", StringComparison.OrdinalIgnoreCase))
                {
                    tbr.LineCap = LineCaps.Round;
                }
                else if (lineCap.Equals("square", StringComparison.OrdinalIgnoreCase))
                {
                    tbr.LineCap = LineCaps.Square;
                }
            }

            if (lineJoin != null)
            {
                if (lineJoin.Equals("bevel", StringComparison.OrdinalIgnoreCase))
                {
                    tbr.LineJoin = LineJoins.Bevel;
                }
                else if (lineJoin.Equals("miter", StringComparison.OrdinalIgnoreCase) ||
lineJoin.Equals("miter-clip", StringComparison.OrdinalIgnoreCase))
                {
                    tbr.LineJoin = LineJoins.Miter;
                }
                else if (lineJoin.Equals("round", StringComparison.OrdinalIgnoreCase))
                {
                    tbr.LineJoin = LineJoins.Round;
                }
            }

            if (opacity != null)
            {
                tbr.Opacity = ParseLengthOrPercentage(opacity, 1);
            }

            if (strokeDashArray != null)
            {
                if (strokeDashArray != "none")
                {
                    double[] parsedArray = ParseListOfDoubles(strokeDashArray);

                    tbr.LineDash = new LineDash(parsedArray[0], parsedArray.Length > 1 ?
parsedArray[1] : parsedArray[0], tbr.LineDash.Phase);
                }
```

```
                else
                {
                    tbr.LineDash = LineDash.SolidLine;
                }
            }

            if (strokeDashOffset != null)
            {
                tbr.LineDash = new LineDash(tbr.LineDash.UnitsOn, tbr.LineDash.UnitsOff, ParseLengthOrPercentage(strokeDashOffset, diagonal));
            }

            if (paintOrder != null)
            {
                if (paintOrder.Equals("normal", StringComparison.OrdinalIgnoreCase))
                {
                    tbr.StrokeFirst = false;
                }
                else
                {
                    if (paintOrder.IndexOf("stroke", StringComparison.OrdinalIgnoreCase) >= 0 && (paintOrder.IndexOf("fill", StringComparison.OrdinalIgnoreCase) < 0 || paintOrder.IndexOf("fill", StringComparison.OrdinalIgnoreCase) > paintOrder.IndexOf("stroke", StringComparison.OrdinalIgnoreCase)))
                    {
                        tbr.StrokeFirst = true;
                    }
                    else
                    {
                        tbr.StrokeFirst = false;
                    }
                }
            }

            if (transform != null)
            {
                gpr.Save();
                tbr.NeedsRestore = true;

                string[] transforms = ParseListOfTransforms(transform);

                for (int i = 0; i < transforms.Length; i++)
                {
                    if (transforms[i].Equals("matrix", StringComparison.OrdinalIgnoreCase))
                    {
                        double a = ParseLengthOrPercentage(transforms[i + 1], 1);
                        double b = ParseLengthOrPercentage(transforms[i + 2], 1);
                        double c = ParseLengthOrPercentage(transforms[i + 3], 1);
                        double d = ParseLengthOrPercentage(transforms[i + 4], 1);
                        double e = ParseLengthOrPercentage(transforms[i + 5], 1);
                        double f = ParseLengthOrPercentage(transforms[i + 6], 1);

                        gpr.Transform(a, b, c, d, e, f);
                        i += 6;
                    }
                    else if (transforms[i].Equals("translate", StringComparison.OrdinalIgnoreCase))
                    {
                        double x = ParseLengthOrPercentage(transforms[i + 1], 1);

                        double y;

                        if (i < transforms.Length - 2 && !double.IsNaN(y = ParseLengthOrPercentage(transforms[i + 2], 1)))
                        {
                            gpr.Translate(x, y);
                            i += 2;
                        }
                        else
                        {
                            gpr.Translate(x, 0);
                            i++;
                        }
                    }
                    else if (transforms[i].Equals("scale", StringComparison.OrdinalIgnoreCase))
                    {
                        double x = ParseLengthOrPercentage(transforms[i + 1], 1);

                        double y;

                        if (i < transforms.Length - 2 && !double.IsNaN(y = ParseLengthOrPercentage(transforms[i + 2], 1)))
                        {
                            gpr.Scale(x, y);
                            i += 2;
                        }
                        else
                        {
```

```
                            gpr.Scale(x, x);
                            i++;
                        }
                    }
                    else if (transforms[i].Equals("rotate", StringComparison.OrdinalIgnoreCase))
                    {
                        double a = ParseLengthOrPercentage(transforms[i + 1], 1) * Math.PI / 180;

                        double x, y;

                        if (i < transforms.Length - 3 && !double.IsNaN(x = ParseLengthOrPercentage(transforms[i + 2], 1)) && !double.IsNaN(y = ParseLengthOrPercentage(transforms[i + 3], 1)))
                        {
                            gpr.RotateAt(a, new Point(x, y));
                            i += 2;
                        }
                        else
                        {
                            gpr.Rotate(a);
                            i++;
                        }
                    }
                    else if (transforms[i].Equals("skewX", StringComparison.OrdinalIgnoreCase))
                    {
                        double psi = ParseLengthOrPercentage(transforms[i + 1], 1) * Math.PI / 180;

                        gpr.Transform(1, 0, Math.Tan(psi), 1, 0, 0);

                        i++;
                    }
                    else if (transforms[i].Equals("skewY", StringComparison.OrdinalIgnoreCase))
                    {
                        double psi = ParseLengthOrPercentage(transforms[i + 1], 1) * Math.PI / 180;

                        gpr.Transform(1, Math.Tan(psi), 0, 1, 0, 0);

                        i++;
                    }
                }
            }

            return tbr;
        }

        private static double ParseLengthOrPercentage(string value, double total, double defaultValue = 0)
        {
            if (value != null)
            {
                if (value.Contains("%"))
                {
                    value = value.Replace("%", "");
                    return double.Parse(value, System.Globalization.CultureInfo.InvariantCulture) * total / 100;
                }
                else if (double.TryParse(value.Replace("px", "").Replace("pt", ""), System.Globalization.NumberStyles.Any, System.Globalization.CultureInfo.InvariantCulture, out double result))
                {
                    return result;
                }
                else
                {
                    string cleanedNumber = Regexes.NumberRegex.Match(value).Value;
                    return double.Parse(cleanedNumber, System.Globalization.CultureInfo.InvariantCulture);
                }
            }
            else
            {
                return defaultValue;
            }
        }

        internal class PresentationAttributes
        {
            public Dictionary<string, FontFamily> EmbeddedFonts;

            public Brush Stroke = null;
            public double StrokeOpacity = 1;
            public Brush Fill = Colour.FromRgb(0, 0, 0);
            public double FillOpacity = 1;
            public Brush CurrentColour = null;
            public double StrokeThickness = 1;
            public LineCaps LineCap = LineCaps.Butt;
            public LineJoins LineJoin = LineJoins.Miter;
```

```
            public double Opacity = 1;
            public LineDash LineDash = new LineDash(0, 0, 0);
            public bool NeedsRestore = false;
            public bool StrokeFirst = false;
            public double X;
            public double Y;
            public double Width;
            public double Height;

            public PresentationAttributes Clone()
            {
                return new PresentationAttributes()
                {
                    EmbeddedFonts = this.EmbeddedFonts,

                    Stroke = this.Stroke,
                    StrokeOpacity = this.StrokeOpacity,
                    Fill = this.Fill,
                    FillOpacity = this.FillOpacity,
                    CurrentColour = this.CurrentColour,
                    StrokeThickness = this.StrokeThickness,
                    LineCap = this.LineCap,
                    LineJoin = this.LineJoin,
                    Opacity = this.Opacity,
                    LineDash = this.LineDash,
                    StrokeFirst = this.StrokeFirst,
                    X = this.X,
                    Y = this.Y,
                    Width = this.Width,
                    Height = this.Height
                };
            }
        }

        private static class Regexes
        {
            public static Regex ListSeparator = new Regex("[ \\t\\n\\r\\f]*,[ \\t\\n\\r\\f]*|[ \\t\\n\\r\\f]+", RegexOptions.Compiled);
            public static Regex FontFamilySeparator = new Regex("(?:^|,)(\"(?:[^\"])*\"|[^,]*)", RegexOptions.Compiled);
            public static Regex NumberRegex = new Regex(@"^[-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?", RegexOptions.Compiled);
        }

        private static double[] ParseListOfDoubles(string value)
        {
            if (value == null)
            {
                return null;
            }

            string[] splitValue = Regexes.ListSeparator.Split(value);
            double[] tbr = new double[splitValue.Length];

            for (int i = 0; i < splitValue.Length; i++)
            {
                tbr[i] = double.Parse(splitValue[i], System.Globalization.CultureInfo.InvariantCulture);
            }

            return tbr;
        }

        private static string[] ParseListOfTransforms(string value)
        {
            if (value == null)
            {
                return null;
            }

            string[] splitValue = Regexes.ListSeparator.Split(value.Replace("(", " ").Replace(")", " ").Trim());

            return splitValue;
        }

        private static List<KeyValuePair<string, FontFamily» GetEmbeddedFonts(string styleBlock)
        {
            StringReader sr = new StringReader(styleBlock);

            List<KeyValuePair<string, FontFamily» tbr = new List<KeyValuePair<string, FontFamily»();

            while (sr.Peek() >= 0)
            {
                string token = ReadCSSToken(sr);

                if (token.Equals("@font-face", StringComparison.OrdinalIgnoreCase))
```

```
                {
                    List<string> tokens = new List<string>();

                    while (!token.Equals("}", StringComparison.OrdinalIgnoreCase))
                    {
                        token = ReadCSSToken(sr);
                        tokens.Add(token);
                    }

                    KeyValuePair<string, FontFamily>? fontFace = ParseFontFaceBlock(tokens);

                    if (fontFace != null)
                    {
                        tbr.Add(fontFace.Value);
                    }
                }
            }

            return tbr;
        }

        private static IEnumerable<KeyValuePair<string, IFilter» GetFilters(XmlNode definitionsNode, List<Stylesheet> styleSheets)
        {
            Dictionary<string, IFilter> tbr = new Dictionary<string, IFilter>();

            foreach (XmlNode definition in definitionsNode.ChildNodes)
            {
                if (definition.Name.Equals("filter", StringComparison.OrdinalIgnoreCase))
                {
                    XmlElement filter = (XmlElement)definition;

                    string id = filter.GetAttribute("id");

                    List<ILocationInvariantFilter> filterElements = new List<ILocationInvariantFilter>();

                    foreach (XmlNode filterDefinition in definition.ChildNodes)
                    {
                        if (filterDefinition.Name.Equals("feGaussianBlur", StringComparison.OrdinalIgnoreCase))
                        {
                            XmlElement actualFilter = (XmlElement)filterDefinition;

                            string stdDeviation = actualFilter.GetAttribute("stdDeviation");

                            filterElements.Add(new GaussianBlurFilter(double.Parse(stdDeviation, System.Globalization.CultureInfo.InvariantCulture)));
                        }
                        else if (filterDefinition.Name.Equals("feColorMatrix", StringComparison.OrdinalIgnoreCase))
                        {
                            XmlElement actualFilter = (XmlElement)filterDefinition;

                            string type = actualFilter.GetAttribute("type");

                            if (type.Equals("matrix", StringComparison.OrdinalIgnoreCase))
                            {
                                string values = actualFilter.GetAttribute("values");
                                double[] parsedValues = (from el in System.Text.RegularExpressions.Regex.Split(values, "\\s") select double.Parse(el.Trim())).ToArray();

                                if (parsedValues.Length == 20)
                                {
                                    double[,] matrix = new double[5, 5];
                                    matrix[4, 4] = 1;

                                    for (int i = 0; i < 20; i++)
                                    {
                                        int y = i / 5;
                                        int x = i % 5;

                                        matrix[y, x] = parsedValues[i];
                                    }

                                    filterElements.Add(new ColourMatrixFilter(new ColourMatrix(matrix)));
                                }
                            }
                        }
                    }

                    if (filterElements.Count > 0)
                    {
                        if (filterElements.Count == 1)
                        {
                            tbr.Add(id, filterElements[0]);
```

```
                        }
                        else
                        {
                            tbr.Add(id, new CompositeLocationInvariantFilter(filterElements));
                        }
                    }
                }
            }

            return tbr;
        }

        private static IEnumerable<KeyValuePair<string, XmlNode» GetMasks(XmlNode definitionsNode, List<Stylesheet> styleSheets)
        {
            Dictionary<string, XmlNode> tbr = new Dictionary<string, XmlNode>();

            foreach (XmlNode definition in definitionsNode.ChildNodes)
            {
                if (definition.Name.Equals("mask", StringComparison.OrdinalIgnoreCase))
                {
                    XmlElement mask = (XmlElement)definition;

                    string id = mask.GetAttribute("id");

                    tbr.Add(id, definition);
                }
            }

            return tbr;
        }

        private static IEnumerable<KeyValuePair<string, Brush» GetGradients(XmlNode definitionsNode, List<Stylesheet> styleSheets)
        {
            Dictionary<string, Brush> tbr = new Dictionary<string, Brush>();
            Dictionary<string, XmlNodeList> stopLists = new Dictionary<string, XmlNodeList>();

            foreach (XmlNode definition in definitionsNode.ChildNodes)
            {
                if (definition.Name.Equals("linearGradient", StringComparison.OrdinalIgnoreCase))
                {
                    XmlElement gradient = (XmlElement)definition;

                    string id = gradient.GetAttribute("id");

                    double x1;
                    double y1;
                    double x2;
                    double y2;

                    if (!(gradient.HasAttribute("x1") && double.TryParse(gradient.GetAttribute("x1"), out x1)))
                    {
                        x1 = 0;
                    }

                    if (!(gradient.HasAttribute("y1") && double.TryParse(gradient.GetAttribute("y1"), out y1)))
                    {
                        y1 = 0;
                    }

                    if (!(gradient.HasAttribute("x2") && double.TryParse(gradient.GetAttribute("x2"), out x2)))
                    {
                        x2 = 0;
                    }

                    if (!(gradient.HasAttribute("y2") && double.TryParse(gradient.GetAttribute("y2"), out y2)))
                    {
                        y2 = 0;
                    }

                    List<GradientStop> gradientStops = new List<GradientStop>();

                    XmlNodeList childNodes;

                    if (gradient.HasAttribute("xlink:href"))
                    {
                        string refId = gradient.GetAttribute("xlink:href").Trim();

                        if (refId.StartsWith("#"))
                        {
                            refId = refId.Substring(1);

```

```
                            if (!stopLists.TryGetValue(refId, out childNodes))
                            {
                                childNodes = gradient.ChildNodes;
                            }
                        }
                        else
                        {
                            childNodes = gradient.ChildNodes;
                        }
                    }
                    else
                    {
                        childNodes = gradient.ChildNodes;
                    }

                    stopLists[id] = childNodes;

                    foreach (XmlNode stopNode in childNodes)
                    {
                        if (stopNode.Name.Equals("stop", StringComparison.OrdinalIgnoreCase))
                        {
                            SetStyleAttributes(stopNode, styleSheets);

                            XmlElement stop = (XmlElement)stopNode;

                            double offset = 0;
                            double opacity = 1;

                            if (stop.HasAttribute("offset"))
                            {
                                offset = ParseLengthOrPercentage(stop.GetAttribute("offset"), 1);
                            }

                            if (stop.HasAttribute("stop-opacity"))
                            {
                                opacity = ParseLengthOrPercentage(stop.GetAttribute("stop-opacity"), 1);
                            }

                            Colour stopColour = Colour.FromRgba(0, 0, 0, 0);

                            if (stop.HasAttribute("stop-color"))
                            {
                                stopColour = (Colour.FromCSSString(stop.GetAttribute("stop-color")) ?? stopColour).WithAlpha(opacity);
                            }

                            gradientStops.Add(new GradientStop(stopColour, offset));
                        }
                    }

                    if (gradient.HasAttribute("gradientTransform"))
                    {
                        string transform = gradient.GetAttribute("gradientTransform");
                        string[] transforms = ParseListOfTransforms(transform);

                        double[,] transformMatrix = MatrixUtils.Identity;

                        for (int i = 0; i < transforms.Length; i++)
                        {
                            if (transforms[i].Equals("matrix", StringComparison.OrdinalIgnoreCase))
                            {
                                double a = ParseLengthOrPercentage(transforms[i + 1], 1);
                                double b = ParseLengthOrPercentage(transforms[i + 2], 1);
                                double c = ParseLengthOrPercentage(transforms[i + 3], 1);
                                double d = ParseLengthOrPercentage(transforms[i + 4], 1);
                                double e = ParseLengthOrPercentage(transforms[i + 5], 1);
                                double f = ParseLengthOrPercentage(transforms[i + 6], 1);

                                transformMatrix = MatrixUtils.Multiply(transformMatrix, new double[,] { { a, c, e }, { b, d, f }, { 0, 0, 1 } });

                                i += 6;
                            }
                            else if (transforms[i].Equals("translate", StringComparison.OrdinalIgnoreCase))
                            {
                                double x = ParseLengthOrPercentage(transforms[i + 1], 1);

                                double y;

                                if (i < transforms.Length - 2 && !double.IsNaN(y = ParseLengthOrPercentage(transforms[i + 2], 1)))
                                {
                                    transformMatrix = MatrixUtils.Translate(transformMatrix, x, y);
                                    i += 2;
                                }
```

```
                                else
                                {
                                    transformMatrix = MatrixUtils.Translate(transformMatrix, x, 0);
                                    i++;
                                }
                            }
                            else if (transforms[i].Equals("scale", StringComparison.OrdinalIgnoreCase))
                            {
                                double x = ParseLengthOrPercentage(transforms[i + 1], 1);

                                double y;

                                if (i < transforms.Length - 2 && !double.IsNaN(y = ParseLengthOrPercentage(transforms[i + 2], 1)))
                                {
                                    transformMatrix = MatrixUtils.Scale(transformMatrix, x, y);
                                    i += 2;
                                }
                                else
                                {
                                    transformMatrix = MatrixUtils.Scale(transformMatrix, x, x);
                                    i++;
                                }
                            }
                            else if (transforms[i].Equals("rotate", StringComparison.OrdinalIgnoreCase))
                            {
                                double a = ParseLengthOrPercentage(transforms[i + 1], 1) * Math.PI / 180;

                                double x, y;

                                if (i < transforms.Length - 3 && !double.IsNaN(x = ParseLengthOrPercentage(transforms[i + 2], 1)) && !double.IsNaN(y = ParseLengthOrPercentage(transforms[i + 3], 1)))
                                {
                                    transformMatrix = MatrixUtils.Translate(transformMatrix, x, y);
                                    transformMatrix = MatrixUtils.Rotate(transformMatrix, a);
                                    transformMatrix = MatrixUtils.Translate(transformMatrix, -x, -y);
                                    i += 2;
                                }
                                else
                                {
                                    transformMatrix = MatrixUtils.Rotate(transformMatrix, a);
                                    i++;
                                }
                            }
                            else if (transforms[i].Equals("skewX", StringComparison.OrdinalIgnoreCase))
                            {
                                double psi = ParseLengthOrPercentage(transforms[i + 1], 1) * Math.PI / 180;

                                transformMatrix = MatrixUtils.Multiply(transformMatrix, new double[,] { { 1, Math.Tan(psi), 0 }, { 0, 1, 0 }, { 0, 0, 1 } });
                                i++;
                            }
                            else if (transforms[i].Equals("skewY", StringComparison.OrdinalIgnoreCase))
                            {
                                double psi = ParseLengthOrPercentage(transforms[i + 1], 1) * Math.PI / 180;

                                transformMatrix = MatrixUtils.Multiply(transformMatrix, new double[,] { { 1, 0, 0 }, { Math.Tan(psi), 1, 0 }, { 0, 0, 1 } });
                                i++;
                            }
                        }

                        double[] start = MatrixUtils.Multiply(transformMatrix, new double[] { x1, y1 });
                        double[] end = MatrixUtils.Multiply(transformMatrix, new double[] { x2, y2 });

                        x1 = start[0];
                        y1 = start[1];
                        x2 = end[0];
                        y2 = end[1];
                    }

                    tbr.Add(id, new LinearGradientBrush(new Point(x1, y1), new Point(x2, y2), gradientStops));
                }
                else if (definition.Name.Equals("radialGradient", StringComparison.OrdinalIgnoreCase))
                {
                    XmlElement gradient = (XmlElement)definition;

```

```
                    string id = gradient.GetAttribute("id");

                    double cx;
                    double cy;
                    double r;

                    if (!(gradient.HasAttribute("cx") && double.TryParse(gradient.GetAttribute("cx"), out cx)))
                    {
                        cx = 0;
                    }

                    if (!(gradient.HasAttribute("cy") && double.TryParse(gradient.GetAttribute("cy"), out cy)))
                    {
                        cy = 0;
                    }

                    if (!(gradient.HasAttribute("r") && double.TryParse(gradient.GetAttribute("r"), out r)))
                    {
                        r = 0;
                    }

                    double fx;
                    double fy;

                    if (!(gradient.HasAttribute("fx") && double.TryParse(gradient.GetAttribute("fx"), out fx)))
                    {
                        fx = cx;
                    }

                    if (!(gradient.HasAttribute("fy") && double.TryParse(gradient.GetAttribute("fy"), out fy)))
                    {
                        fy = cy;
                    }

                    List<GradientStop> gradientStops = new List<GradientStop>();

                    XmlNodeList childNodes;

                    if (gradient.HasAttribute("xlink:href"))
                    {
                        string refId = gradient.GetAttribute("xlink:href").Trim();

                        if (refId.StartsWith("#"))
                        {
                            refId = refId.Substring(1);

                            if (!stopLists.TryGetValue(refId, out childNodes))
                            {
                                childNodes = gradient.ChildNodes;
                            }
                        }
                        else
                        {
                            childNodes = gradient.ChildNodes;
                        }
                    }
                    else
                    {
                        childNodes = gradient.ChildNodes;
                    }

                    stopLists[id] = childNodes;

                    foreach (XmlNode stopNode in childNodes)
                    {
                        if (stopNode.Name.Equals("stop", StringComparison.OrdinalIgnoreCase))
                        {
                            SetStyleAttributes(stopNode, styleSheets);

                            XmlElement stop = (XmlElement)stopNode;

                            double offset = 0;
                            double opacity = 1;

                            if (stop.HasAttribute("offset"))
                            {
                                offset = ParseLengthOrPercentage(stop.GetAttribute("offset"), 1);
                            }

                            if (stop.HasAttribute("stop-opacity"))
                            {
                                opacity = ParseLengthOrPercentage(stop.GetAttribute("stop-opacity"),
```

```
1);
                            }

                            Colour stopColour = Colour.FromRgba(0, 0, 0, 0);

                            if (stop.HasAttribute("stop-color"))
                            {
                                stopColour = (Colour.FromCSSString(stop.GetAttribute("stop-color")) ??
stopColour).WithAlpha(opacity);
                            }

                            gradientStops.Add(new GradientStop(stopColour, offset));
                        }
                    }

                    if (gradient.HasAttribute("gradientTransform"))
                    {
                        string transform = gradient.GetAttribute("gradientTransform");
                        string[] transforms = ParseListOfTransforms(transform);

                        double[,] transformMatrix = MatrixUtils.Identity;

                        for (int i = 0; i < transforms.Length; i++)
                        {
                            if (transforms[i].Equals("matrix", StringComparison.OrdinalIgnoreCase))
                            {
                                double a = ParseLengthOrPercentage(transforms[i + 1], 1);
                                double b = ParseLengthOrPercentage(transforms[i + 2], 1);
                                double c = ParseLengthOrPercentage(transforms[i + 3], 1);
                                double d = ParseLengthOrPercentage(transforms[i + 4], 1);
                                double e = ParseLengthOrPercentage(transforms[i + 5], 1);
                                double f = ParseLengthOrPercentage(transforms[i + 6], 1);

                                transformMatrix = MatrixUtils.Multiply(transformMatrix, new double[,]
{ { a, c, e }, { b, d, f }, { 0, 0, 1 } });

                                i += 6;
                            }
                            else if (transforms[i].Equals("translate",
StringComparison.OrdinalIgnoreCase))
                            {
                                double x = ParseLengthOrPercentage(transforms[i + 1], 1);

                                double y;

                                if (i < transforms.Length - 2 && !double.IsNaN(y =
ParseLengthOrPercentage(transforms[i + 2], 1)))
                                {
                                    transformMatrix = MatrixUtils.Translate(transformMatrix, x, y);
                                    i += 2;
                                }
                                else
                                {
                                    transformMatrix = MatrixUtils.Translate(transformMatrix, x, 0);
                                    i++;
                                }
                            }
                            else if (transforms[i].Equals("scale",
StringComparison.OrdinalIgnoreCase))
                            {
                                double x = ParseLengthOrPercentage(transforms[i + 1], 1);

                                double y;

                                if (i < transforms.Length - 2 && !double.IsNaN(y =
ParseLengthOrPercentage(transforms[i + 2], 1)))
                                {
                                    transformMatrix = MatrixUtils.Scale(transformMatrix, x, y);
                                    i += 2;
                                }
                                else
                                {
                                    transformMatrix = MatrixUtils.Scale(transformMatrix, x, x);
                                    i++;
                                }
                            }
                            else if (transforms[i].Equals("rotate",
StringComparison.OrdinalIgnoreCase))
                            {
                                double a = ParseLengthOrPercentage(transforms[i + 1], 1) * Math.PI /
180;

                                double x, y;

                                if (i < transforms.Length - 3 && !double.IsNaN(x =
ParseLengthOrPercentage(transforms[i + 2], 1)) && !double.IsNaN(y =
ParseLengthOrPercentage(transforms[i + 3], 1)))
```

```
                                {
                                    transformMatrix = MatrixUtils.Translate(transformMatrix, x, y);
                                    transformMatrix = MatrixUtils.Rotate(transformMatrix, a);
                                    transformMatrix = MatrixUtils.Translate(transformMatrix, -x, -y);
                                    i += 2;
                                }
                                else
                                {
                                    transformMatrix = MatrixUtils.Rotate(transformMatrix, a);
                                    i++;
                                }
                            }
                            else if (transforms[i].Equals("skewX", StringComparison.OrdinalIgnoreCase))
                            {
                                double psi = ParseLengthOrPercentage(transforms[i + 1], 1) * Math.PI / 180;

                                transformMatrix = MatrixUtils.Multiply(transformMatrix, new double[,] { { 1, Math.Tan(psi), 0 }, { 0, 1, 0 }, { 0, 0, 1 } });
                                i++;
                            }
                            else if (transforms[i].Equals("skewY", StringComparison.OrdinalIgnoreCase))
                            {
                                double psi = ParseLengthOrPercentage(transforms[i + 1], 1) * Math.PI / 180;

                                transformMatrix = MatrixUtils.Multiply(transformMatrix, new double[,] { { 1, 0, 0 }, { Math.Tan(psi), 1, 0 }, { 0, 0, 1 } });
                                i++;
                            }
                        }

                        double determinant = transformMatrix[0, 0] * (transformMatrix[1, 1] * transformMatrix[2, 2] - transformMatrix[1, 2] * transformMatrix[2, 1]) -
                            transformMatrix[0, 1] * (transformMatrix[1, 0] * transformMatrix[2, 2] - transformMatrix[1, 2] * transformMatrix[2, 0]) +
                            transformMatrix[0, 2] * (transformMatrix[1, 0] * transformMatrix[2, 1] - transformMatrix[1, 1] * transformMatrix[2, 0]);

                        double[] focus = MatrixUtils.Multiply(transformMatrix, new double[] { fx, fy });
                        double[] centre = MatrixUtils.Multiply(transformMatrix, new double[] { cx, cy });

                        fx = focus[0];
                        fy = focus[1];
                        cx = centre[0];
                        cy = centre[1];
                        r = r * Math.Sqrt(determinant);
                    }

                    tbr.Add(id, new RadialGradientBrush(new Point(fx, fy), new Point(cx, cy), r, gradientStops));
                }
            }


            return tbr;
        }

        private static KeyValuePair<string, FontFamily>? ParseFontFaceBlock(List<string> tokens)
        {
            int fontFamilyInd = tokens.IndexOf("font-family");
            string fontFamilyName = tokens[fontFamilyInd + 2].Trim().Trim('"').Trim();

            int srcInd = tokens.IndexOf("src");
            string src = tokens[srcInd + 2];

            string mimeType = src.Substring(src.IndexOf("data:") + 5);
            mimeType = mimeType.Substring(0, mimeType.IndexOf(";"));

            if (mimeType.Equals("font/ttf", StringComparison.OrdinalIgnoreCase) || mimeType.Equals("font/truetype", StringComparison.OrdinalIgnoreCase) || mimeType.Equals("application/x-font-ttf", StringComparison.OrdinalIgnoreCase))
            {
                src = src.Substring(src.IndexOf("base64,") + 7);
                src = src.TrimEnd(')').TrimEnd('\"').TrimEnd(')');
                byte[] fontBytes = Convert.FromBase64String(src);

                string tempFile = Path.GetTempFileName();

                File.WriteAllBytes(tempFile, fontBytes);

                FontFamily family = FontFamily.ResolveFontFamily(tempFile);
                return new KeyValuePair<string, FontFamily>(fontFamilyName, family);
```

```
            }

            return null;
        }

        private const string CSSDelimiters = ":;,{}";

        private static string ReadCSSToken(StringReader reader)
        {
            StringBuilder tbr = new StringBuilder();

            bool openQuotes = false;
            int openParentheses = 0;

            int c = reader.Read();
            if (c >= 0)
            {
                tbr.Append((char)c);

                if ((char)c == '"')
                {
                    openQuotes = !openQuotes;
                }

                if ((char)c == '(')
                {
                    openParentheses++;
                }
                if ((char)c == ')')
                {
                    openParentheses--;
                }


                while (c >= 0 && (!CSSDelimiters.Contains((char)c) || openQuotes || openParentheses > 0))
                {
                    c = reader.Read();
                    tbr.Append((char)c);
                    if ((char)c == '"')
                    {
                        openQuotes = !openQuotes;
                    }
                    if ((char)c == '(')
                    {
                        openParentheses++;
                    }
                    if ((char)c == ')')
                    {
                        openParentheses--;
                    }
                    c = reader.Peek();
                }
            }

            string val = tbr.ToString().Trim();

            return (string.IsNullOrEmpty(val) && c >= 0) ? ReadCSSToken(reader) : val;
        }
    }

    internal class PathTransformerGraphicsContext : IGraphicsContext
    {
        public GraphicsPath CurrentPath = new GraphicsPath();
        public double[,] TransformMatrix = MatrixUtils.Identity;

        private Stack<double[,]> transformMatrices;
        public PathTransformerGraphicsContext()
        {
            transformMatrices = new Stack<double[,]>();
            transformMatrices.Push((double[,])TransformMatrix.Clone());
        }

        public void CubicBezierTo(double p1X, double p1Y, double p2X, double p2Y, double p3X, double p3Y)
        {
            double[] p1 = MatrixUtils.Multiply(TransformMatrix, new double[] { p1X, p1Y });
            double[] p2 = MatrixUtils.Multiply(TransformMatrix, new double[] { p2X, p2Y });
            double[] p3 = MatrixUtils.Multiply(TransformMatrix, new double[] { p3X, p3Y });

            CurrentPath.CubicBezierTo(p1[0], p1[1], p2[0], p2[1], p3[0], p3[1]);
        }

        public void LineTo(double x, double y)
        {
            double[] p = MatrixUtils.Multiply(TransformMatrix, new double[] { x, y });
            CurrentPath.LineTo(p[0], p[1]);
```

```
        }

        public void MoveTo(double x, double y)
        {
            double[] p = MatrixUtils.Multiply(TransformMatrix, new double[] { x, y });
            CurrentPath.MoveTo(p[0], p[1]);
        }

        public void Close()
        {
            CurrentPath.Close();
        }

        public void Restore()
        {
            TransformMatrix = transformMatrices.Pop();
        }

        public void Rotate(double angle)
        {
            TransformMatrix = MatrixUtils.Rotate(TransformMatrix, angle);
        }

        public void Save()
        {
            transformMatrices.Push((double[,])TransformMatrix.Clone());
        }

        public void Scale(double scaleX, double scaleY)
        {
            TransformMatrix = MatrixUtils.Scale(TransformMatrix, scaleX, scaleY);
        }

        public void Transform(double a, double b, double c, double d, double e, double f)
        {
            double[,] transfMatrix = new double[3, 3] { { a, c, e }, { b, d, f }, { 0, 0, 1 } };
            TransformMatrix = MatrixUtils.Multiply(TransformMatrix, transfMatrix);
        }

        public void Translate(double x, double y)
        {
            TransformMatrix = MatrixUtils.Translate(TransformMatrix, x, y);
        }



        public double Width => throw new NotImplementedException();

        public double Height => throw new NotImplementedException();

        public Font Font { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }
        public TextBaselines TextBaseline { get => throw new NotImplementedException(); set => throw new NotImplementedException(); }

        public Brush FillStyle => Colours.Black;

        public Brush StrokeStyle => Colours.Black;

        public double LineWidth { get => 1; set { } }
        public LineCaps LineCap { set { } }
        public LineJoins LineJoin { set { } }
        public string Tag { get => null; set { } }

        public void DrawRasterImage(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image)
        {
            throw new NotImplementedException();
        }

        public void Fill()
        {

        }

        public void FillText(string text, double x, double y)
        {
            throw new NotImplementedException();
        }

        public void Rectangle(double x0, double y0, double width, double height)
        {
            MoveTo(x0, y0);
            LineTo(x0 + width, y0);
            LineTo(x0 + width, y0 + height);
            LineTo(x0, y0 + height);
```

```
            Close();
        }

        public void SetClippingPath()
        {
            throw new NotImplementedException();
        }

        public void SetFillStyle((int r, int g, int b, double a) style)
        {

        }

        public void SetFillStyle(Brush style)
        {

        }

        public void SetLineDash(LineDash dash)
        {

        }

        public void SetStrokeStyle((int r, int g, int b, double a) style)
        {

        }

        public void SetStrokeStyle(Brush style)
        {

        }

        public void Stroke()
        {

        }

        public void StrokeText(string text, double x, double y)
        {
            throw new NotImplementedException();
        }

        public void DrawFilteredGraphics(Graphics graphics, IFilter filter)
        {
            graphics.CopyToIGraphicsContext(this);
        }
    }
}
```

8.25 Lights.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Linq;

namespace VectSharp.ThreeD
{
    /// <summary>
    /// Represents the intensity of a light source at a particular point.
    /// </summary>
    public struct LightIntensity
    {
        /// <summary>
        /// The intensity of the light.
        /// </summary>
```

```
        public double Intensity;

        /// <summary>
        /// The direction towards from which the light comes.
        /// </summary>
        public NormalizedVector3D Direction;

        /// <summary>
        /// Creates a new <see cref="LightIntensity"/>.
        /// </summary>
        /// <param name="intensity">The intensity of the light.</param>
        /// <param name="direction">The direction from which the light comes.</param>
        public LightIntensity(double intensity, NormalizedVector3D direction)
        {
            this.Intensity = intensity;
            this.Direction = direction;
        }

        /// <summary>
        /// Deconstructs the struct.
        /// </summary>
        /// <param name="intensity">This parameter will hold the <see cref="Intensity"/> of the light.</param>
        /// <param name="direction">This parameter will hold the <see cref="Direction"/> of the light.</param>
        public void Deconstruct(out double intensity, out NormalizedVector3D direction)
        {
            intensity = this.Intensity;
            direction = this.Direction;
        }
    }

    /// <summary>
    /// Represents a light source.
    /// </summary>
    public interface ILightSource
    {
        /// <summary>
        /// Computes the light intensity at the specified point, without taking into account any obstructions.
        /// </summary>
        /// <param name="point">The <see cref="Point3DElement"/> at which the light intensity should be computed.</param>
        /// <returns></returns>
        LightIntensity GetLightAt(Point3D point);

        /// <summary>
        /// Determines whether the light casts a shadow or not.
        /// </summary>
        bool CastsShadow { get; }

        /// <summary>
        /// Determines the amount of obstruction of the light that results at <paramref name="point"/> due to the specified <paramref name="shadowingTriangles"/>.
        /// </summary>
        /// <param name="point">The <see cref="Point3D"/> at which the obstruction should be computed.</param>
        /// <param name="shadowingTriangles">A collection of <see cref="Triangle3DElement"/> casting shadows.</param>
        /// <returns>1 if the light is completely obstructed, 0 if the light is completely visible, a value between these if the light is partially obstructed.</returns>
        double GetObstruction(Point3D point, IEnumerable<Triangle3DElement> shadowingTriangles);
    }

    /// <summary>
    /// Represents a uniform ambien light source.
    /// </summary>
    public class AmbientLightSource : ILightSource
    {
        /// <summary>
        /// The intensity of the light.
        /// </summary>
        public double Intensity { get; set; }

        /// <inheritdoc/>
        public bool CastsShadow => false;

        /// <summary>
        /// Creates a new <see cref="AmbientLightSource"/> instance.
        /// </summary>
        /// <param name="intensity">The intensity of the light.</param>
        public AmbientLightSource(double intensity)
        {
            this.Intensity = intensity;
        }

        /// <inheritdoc/>
```

```
        public LightIntensity GetLightAt(Point3D point)
        {
            return new LightIntensity(Intensity, new NormalizedVector3D(double.NaN, double.NaN, double.NaN));
        }

        /// <inheritdoc/>
        public double GetObstruction(Point3D point, IEnumerable<Triangle3DElement> shadowingTriangles)
        {
            return 0;
        }
    }

    /// <summary>
    /// Represents a parallel light source.
    /// </summary>
    public class ParallelLightSource : ILightSource
    {
        /// <summary>
        /// The intensity of the light.
        /// </summary>
        public double Intensity { get; set; }

        /// <summary>
        /// The direction along which the light travels.
        /// </summary>
        public NormalizedVector3D Direction { get; }

        /// <summary>
        /// The reverse of <see cref="Direction"/>.
        /// </summary>
        public NormalizedVector3D ReverseDirection { get; }

        /// <inheritdoc/>
        public bool CastsShadow { get; set; } = true;

        /// <summary>
        /// Creates a new <see cref="ParallelLightSource"/> instance.
        /// </summary>
        /// <param name="intensity">The intensity of the light.</param>
        /// <param name="direction">The direction along which the light travels.</param>
        public ParallelLightSource(double intensity, NormalizedVector3D direction)
        {
            this.Intensity = intensity;
            this.Direction = direction;
            this.ReverseDirection = direction.Reverse();
        }

        /// <inheritdoc/>
        public LightIntensity GetLightAt(Point3D point)
        {
            return new LightIntensity(Intensity, Direction);
        }

        /// <inheritdoc/>
        public double GetObstruction(Point3D point, IEnumerable<Triangle3DElement> shadowingTriangles)
        {
            foreach (Triangle3DElement triangle in shadowingTriangles)
            {
                Point3D? projected = triangle.ProjectOnThisPlane(point, ReverseDirection, true, double.PositiveInfinity);

                if (projected != null && Intersections3D.PointInTriangle(projected.Value, triangle.Point1, triangle.Point2, triangle.Point3))
                {
                    return 1;
                }
            }

            return 0;
        }
    }

    /// <summary>
    /// Represents a point light source.
    /// </summary>
    public class PointLightSource : ILightSource
    {
        /// <inheritdoc/>
        public bool CastsShadow { get; set; } = true;

        /// <summary>
        /// The position of the light source.
        /// </summary>
        public Point3D Position { get; set; }

        /// <summary>
```

```
        /// The base intensity of the light.
        /// </summary>
        public double Intensity { get; set; }

        /// <summary>
        /// An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.
        /// </summary>
        public double DistanceAttenuationExponent { get; set; } = 2;

        /// <summary>
        /// Creates a new <see cref="PointLightSource"/> instance.
        /// </summary>
        /// <param name="intensity">The intensity of the light.</param>
        /// <param name="position">The position of the light source.</param>
        public PointLightSource(double intensity, Point3D position)
        {
            this.Position = position;
            this.Intensity = intensity;
        }

        /// <inheritdoc/>
        public LightIntensity GetLightAt(Point3D point)
        {
            if (DistanceAttenuationExponent == 2)
            {
                return new LightIntensity(Intensity / ((point.X - Position.X) * (point.X - Position.X) + (point.Y - Position.Y) * (point.Y - Position.Y) + (point.Z - Position.Z) * (point.Z - Position.Z)), (point - Position).Normalize());
            }
            else if (DistanceAttenuationExponent == 0)
            {
                return new LightIntensity(Intensity, (point - Position).Normalize());
            }
            else
            {
                return new LightIntensity(Intensity / Math.Pow((point.X - Position.X) * (point.X - Position.X) + (point.Y - Position.Y) * (point.Y - Position.Y) + (point.Z - Position.Z) * (point.Z - Position.Z), DistanceAttenuationExponent * 0.5), (point - Position).Normalize());
            }
        }

        /// <inheritdoc/>
        public double GetObstruction(Point3D point, IEnumerable<Triangle3DElement> shadowingTriangles)
        {
            Vector3D reverseDir = this.Position - point;
            double maxD = reverseDir.Modulus;
            NormalizedVector3D reverseDirNorm = new NormalizedVector3D(reverseDir.X / maxD, reverseDir.Y / maxD, reverseDir.Z / maxD, false);

            foreach (Triangle3DElement triangle in shadowingTriangles)
            {
                Point3D? projected = triangle.ProjectOnThisPlane(point, reverseDirNorm, true, maxD);

                if (projected != null && Intersections3D.PointInTriangle(projected.Value, triangle.Point1, triangle.Point2, triangle.Point3))
                {
                    return 1;
                }
            }

            return 0;
        }
    }

    /// <summary>
    /// Represents a conic spotlight.
    /// </summary>
    public class SpotlightLightSource : ILightSource
    {
        /// <inheritdoc/>
        public bool CastsShadow { get; set; } = true;

        /// <summary>
        /// The position of the light source.
        /// </summary>
        public Point3D Position { get; set; }

        /// <summary>
        /// The direction of the cone axis.
        /// </summary>
        public NormalizedVector3D Direction { get; set; }

        /// <summary>
        /// The base intensity of the light.
        /// </summary>
        public double Intensity { get; set; }
```

```
        /// <summary>
        /// The angular size of the light cone, in radians.
        /// </summary>
        public double BeamWidthAngle { get; set; }

        /// <summary>
        /// The angular size of the cutoff cone, in radians.
        /// </summary>
        public double CutoffAngle { get; set; }

        /// <summary>
        /// An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.
        /// </summary>
        public double DistanceAttenuationExponent { get; set; } = 2;

        /// <summary>
        /// An exponent determining how fast the light attenuates between the main light cone and the cutoff cone.
        /// </summary>
        public double AngleAttenuationExponent { get; set; } = 1;

        /// <summary>
        /// Creates a new <see cref="SpotlightLightSource"/> instance.
        /// </summary>
        /// <param name="intensity">The intensity of the light.</param>
        /// <param name="position">The position of the light source.</param>
        /// <param name="direction">The direction of the cone's axis.</param>
        /// <param name="beamWidthAngle">The angular size of the light cone, in radians.</param>
        /// <param name="cutoffAngle">The angular size of the cutoff cone, in radians.</param>
        public SpotlightLightSource(double intensity, Point3D position, NormalizedVector3D direction, double beamWidthAngle, double cutoffAngle)
        {
            this.Position = position;
            this.Direction = direction;
            this.Intensity = intensity;
            this.BeamWidthAngle = beamWidthAngle;
            this.CutoffAngle = cutoffAngle;
        }

        /// <inheritdoc/>
        public LightIntensity GetLightAt(Point3D point)
        {
            double angle = Math.Atan2(((point - Position) ^ Direction).Modulus, (point - Position) * Direction);
            if (angle > Math.PI)
            {
                angle -= 2 * Math.PI;
            }
            angle = Math.Abs(angle);
            double intensity;

            if (DistanceAttenuationExponent == 0)
            {
                intensity = Intensity;
            }
            else if (DistanceAttenuationExponent == 2)
            {
                intensity = Intensity / ((point.X - Position.X) * (point.X - Position.X) + (point.Y - Position.Y) * (point.Y - Position.Y) + (point.Z - Position.Z) * (point.Z - Position.Z));
            }
            else
            {
                intensity = Intensity / Math.Pow((point.X - Position.X) * (point.X - Position.X) + (point.Y - Position.Y) * (point.Y - Position.Y) + (point.Z - Position.Z) * (point.Z - Position.Z), 0.5 * DistanceAttenuationExponent);
            }

            if (angle <= BeamWidthAngle)
            {
                // * 1
            }
            else if (angle >= CutoffAngle)
            {
                intensity = 0;
            }
            else if (AngleAttenuationExponent == 0)
            {
                // * 1
            }
            else if (AngleAttenuationExponent == 1)
            {
                intensity *= (CutoffAngle - angle) / (CutoffAngle - BeamWidthAngle);
            }
            else
            {
```

```
                intensity *= Math.Pow((CutoffAngle - angle) / (CutoffAngle - BeamWidthAngle), AngleAttenuationExponent);
            }

            return new LightIntensity(intensity, (point - Position).Normalize());
        }

        /// <inheritdoc/>
        public double GetObstruction(Point3D point, IEnumerable<Triangle3DElement> shadowingTriangles)
        {
            Vector3D reverseDir = this.Position - point;
            double maxD = reverseDir.Modulus;
            NormalizedVector3D reverseDirNorm = new NormalizedVector3D(reverseDir.X / maxD, reverseDir.Y / maxD, reverseDir.Z / maxD, false);

            foreach (Triangle3DElement triangle in shadowingTriangles)
            {
                Point3D? projected = triangle.ProjectOnThisPlane(point, reverseDirNorm, true, maxD);

                if (projected != null && Intersections3D.PointInTriangle(projected.Value, triangle.Point1, triangle.Point2, triangle.Point3))
                {
                    return 1;
                }
            }

            return 0;
        }
    }

    /// <summary>
    /// Represents a point light source with a stencil in front of it.
    /// </summary>
    public class MaskedLightSource : ILightSource
    {
        /// <inheritdoc/>
        public bool CastsShadow { get; set; } = true;

        /// <summary>
        /// The position of the light source.
        /// </summary>
        public Point3D Position { get; }

        /// <summary>
        /// The projection of the <see cref="Position"/> on the mask plane along the light's <see cref="Direction"/>.
        /// </summary>
        public Point3D Origin { get; }

        /// <summary>
        /// The direction of the light.
        /// </summary>
        public NormalizedVector3D Direction { get; }

        /// <summary>
        /// The distance between the light source and the mask plane.
        /// </summary>
        public double Distance { get; }

        private List<Point3D[]> TriangulatedMask { get; }

        /// <summary>
        /// The base intensity of the light.
        /// </summary>
        public double Intensity { get; set; }

        /// <summary>
        /// An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.
        /// </summary>
        public double DistanceAttenuationExponent { get; set; } = 2;

        /// <summary>
        /// An exponent determining how fast the light attenuates away from the light's axis. Set to 0 to disable angular attenuation.
        /// </summary>
        public double AngleAttenuationExponent { get; set; } = 1;

        /// <summary>
        /// Creates a new <see cref="MaskedLightSource"/> by triangulating the specified <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="intensity">The base intensity of the light.</param>
        /// <param name="position">The position of the light source.</param>
        /// <param name="direction">The direction of the light.</param>
        /// <param name="distance">The distance between the light source and the mask plane.</param>
        /// <param name="mask">A <see cref="GraphicsPath"/> representing the transparent part of the
```

```
mask.</param>
        /// <param name="maskOrientation">An angle in radians determining the orientation of the 2D mask in the mask plane.</param>
        /// <param name="triangulationResolution">The resolution to use to triangulate the <paramref name="mask"/>.</param>
        public MaskedLightSource(double intensity, Point3D position, NormalizedVector3D direction, double distance, GraphicsPath mask, double maskOrientation, double triangulationResolution) : this(intensity, position, direction, distance, mask.Triangulate(triangulationResolution, true), maskOrientation)
        {

        }

        /// <summary>
        /// Creates a new <see cref="MaskedLightSource"/> using the specified <paramref name="triangulatedMask"/>.
        /// </summary>
        /// <param name="intensity">The base intensity of the light.</param>
        /// <param name="position">The position of the light source.</param>
        /// <param name="direction">The direction of the light.</param>
        /// <param name="distance">The distance between the light source and the mask plane.</param>
        /// <param name="triangulatedMask">A collection of <see cref="GraphicsPath"/>s representing the transparent part of the mask. Each <see cref="GraphicsPath"/> should represent a single triangle.</param>
        /// <param name="maskOrientation">An angle in radians determining the orientation of the 2D mask in the mask plane.</param>
        public MaskedLightSource(double intensity, Point3D position, NormalizedVector3D direction, double distance, IEnumerable<GraphicsPath> triangulatedMask, double maskOrientation)
        {
            this.Intensity = intensity;
            this.Position = position;
            this.Direction = direction;
            this.Distance = distance;
            this.Origin = this.Position + distance * this.Direction;

            double[,] rotation = Matrix3D.RotationToAlignWithZ(direction).Inverse();

            double[,] rotation2 = Matrix3D.RotationAroundAxis(direction, maskOrientation);

            List<Point3D[]> maskList = new List<Point3D[]>();

            foreach (GraphicsPath trianglePath in triangulatedMask)
            {
                Point3D[] triangle = new Point3D[3];

                List<Point> points = trianglePath.GetPoints().First();

                for (int i = 0; i < 3; i++)
                {
                    triangle[i] = (Point3D)((Vector3D)(rotation2 * (rotation * new Point3D(points[i].X, points[i].Y, 0))) + Origin);
                }

                maskList.Add(triangle);
            }

            this.TriangulatedMask = maskList;
        }

        /// <inheritdoc/>
        public LightIntensity GetLightAt(Point3D point)
        {
            double d = Distance / ((point - this.Position) * this.Direction);
            Point3D pt = this.Position + (point - this.Position) * d;

            bool contained = false;

            foreach (Point3D[] triangle in TriangulatedMask)
            {
                if (Intersections3D.PointInTriangle(pt, triangle[0], triangle[1], triangle[2]))
                {
                    contained = true;
                    break;
                }
            }

            if (contained)
            {
                double intensity;

                if (DistanceAttenuationExponent == 0)
                {
                    intensity = Intensity;
                }
                else if (DistanceAttenuationExponent == 2)
                {
                    intensity = Intensity / ((point.X - Position.X) * (point.X - Position.X) +
```

```
(point.Y - Position.Y) * (point.Y - Position.Y) + (point.Z - Position.Z) * (point.Z - Position.Z));
                }
                else
                {
                    intensity = Intensity / Math.Pow((point.X - Position.X) * (point.X - Position.X) + (point.Y - Position.Y) * (point.Y - Position.Y) + (point.Z - Position.Z) * (point.Z - Position.Z), 0.5 * DistanceAttenuationExponent);
                }

                if (AngleAttenuationExponent == 0)
                {
                    // * 1
                }
                else
                {
                    double angle = Math.Atan2(((point - Position) ^ Direction).Modulus, (point - Position) * Direction);
                    if (angle > Math.PI)
                    {
                        angle -= 2 * Math.PI;
                    }

                    if (Math.Abs(angle) < Math.PI / 2)
                    {
                        angle = Math.Abs(angle) / Math.PI * 2;

                        if (AngleAttenuationExponent == 1)
                        {
                            intensity *= 1 - angle;
                        }
                        else
                        {
                            intensity *= Math.Pow(1 - angle, AngleAttenuationExponent);
                        }
                    }
                    else
                    {
                        intensity = 0;
                    }
                }

                return new LightIntensity(intensity, (point - Position).Normalize());
            }
            else
            {
                return new LightIntensity(0, (point - Position).Normalize());
            }
        }

        /// <inheritdoc/>
        public double GetObstruction(Point3D point, IEnumerable<Triangle3DElement> shadowingTriangles)
        {
            Vector3D reverseDir = this.Position - point;
            double maxD = reverseDir.Modulus;
            NormalizedVector3D reverseDirNorm = new NormalizedVector3D(reverseDir.X / maxD, reverseDir.Y / maxD, reverseDir.Z / maxD, false);

            foreach (Triangle3DElement triangle in shadowingTriangles)
            {
                Point3D? projected = triangle.ProjectOnThisPlane(point, reverseDirNorm, true, maxD);

                if (projected != null && Intersections3D.PointInTriangle(projected.Value, triangle.Point1, triangle.Point2, triangle.Point3))
                {
                    return 1;
                }
            }

            return 0;
        }
    }

    /// <summary>
    /// Represents a light source emitting light from a circular area.
    /// </summary>
    public class AreaLightSource : ILightSource
    {
        /// <inheritdoc/>
        public bool CastsShadow { get; set; } = true;

        /// <summary>
        /// The centre of the light-emitting area.
        /// </summary>
        public Point3D Center { get; }

        private Point3D VirtualSource { get; }

```

```
        /// <summary>
        /// The direction of the light's main axis, i.e. the normal to the plane containing the light-emitting area.
        /// </summary>
        public NormalizedVector3D Direction { get; }

        /// <summary>
        /// The radius of the light emitting area.
        /// </summary>
        public double Radius { get; }

        /// <summary>
        /// The radius of the penumbra area.
        /// </summary>
        public double PenumbraRadius { get; }

        /// <summary>
        /// The base intensity of the light.
        /// </summary>
        public double Intensity { get; set; }

        /// <summary>
        /// The distance between the focal point of the light and the light's <see cref="Center"/>.
        /// </summary>
        public double SourceDistance { get; }

        /// <summary>
        /// An exponent determining how fast the light attenuates with increasing distance. Set to 0 to disable distance attenuation.
        /// </summary>
        public double DistanceAttenuationExponent { get; set; } = 2;

        /// <summary>
        /// An exponent determining how fast the light attenuates between the light-emitting area radius and the penumbra radius.
        /// </summary>
        public double PenumbraAttenuationExponent { get; set; } = 1;

        /// <summary>
        /// The number of points to use when determining the amount of light that is obstructed at a certain point.
        /// </summary>
        public int ShadowSamplingPointCount { get; }

        private Point3D[] ShadowSamplingPoints { get; }

        /// <summary>
        /// Creates a new <see cref="AreaLightSource"/> instance.
        /// </summary>
        /// <param name="intensity">The base intensity of the light.</param>
        /// <param name="center">The centre of the light-emitting area.</param>
        /// <param name="radius">The radius of the light-emitting area.</param>
        /// <param name="penumbraRadius">The radius of the penumbra area.</param>
        /// <param name="direction">The direction of the light.</param>
        /// <param name="sourceDistance">The distance between the focal point of the light and the light's center.</param>
        /// <param name="shadowSamplingPointCount">The number of points to use when determining the amount of light that is obstructed at a certain point.</param>
        public AreaLightSource(double intensity, Point3D center, double radius, double penumbraRadius, NormalizedVector3D direction, double sourceDistance, int shadowSamplingPointCount)
        {
            if (shadowSamplingPointCount < 1)
            {
                throw new ArgumentOutOfRangeException(nameof(shadowSamplingPointCount), shadowSamplingPointCount, "The number of shadow sampling points must be greater than or equal to 1!");
            }

            this.Center = center;
            this.Direction = direction;
            this.Intensity = intensity;
            this.Radius = radius;
            this.PenumbraRadius = penumbraRadius;
            this.SourceDistance = sourceDistance;

            this.VirtualSource = this.Center - this.Direction * this.SourceDistance;

            this.ShadowSamplingPointCount = shadowSamplingPointCount;

            NormalizedVector3D yAxis;

            if (Math.Abs(new Vector3D(0, 1, 0) * this.Direction) < 1)
            {
                yAxis = (new Vector3D(0, 1, 0) - (new Vector3D(0, 1, 0) * this.Direction) * this.Direction).Normalize();
            }
            else
```

```
            {
                yAxis = (new Vector3D(0, 0, 1) - (new Vector3D(0, 0, 1) * this.Direction) * this.Direction).Normalize();
            }

            NormalizedVector3D xAxis = (this.Direction ^ yAxis).Normalize();

            this.ShadowSamplingPoints = new Point3D[shadowSamplingPointCount - 1];

            for (int i = 0; i < shadowSamplingPointCount - 1; i++)
            {
                double r = ((double)i / (shadowSamplingPointCount - 2)) * this.Radius;
                double theta = (double)i / (shadowSamplingPointCount - 2) * 2 * Math.PI * ((shadowSamplingPointCount - 1) / 3.7);

                double x = r * Math.Cos(theta);
                double y = r * Math.Sin(theta);

                Point3D pt = this.Center + x * xAxis + y * yAxis;
                this.ShadowSamplingPoints[i] = pt;
            }
        }

        /// <inheritdoc/>
        public LightIntensity GetLightAt(Point3D point)
        {
            double denom = ((point - this.VirtualSource) * this.Direction);

            if (denom <= 0)
            {
                return new LightIntensity(0, (point - this.VirtualSource).Normalize());
            }
            else
            {
                double d = this.SourceDistance / denom;
                Point3D pt = this.VirtualSource + d * (point - this.VirtualSource);
                double distSq = (pt.X - this.Center.X) * (pt.X - this.Center.X) + (pt.Y - this.Center.Y) * (pt.Y - this.Center.Y) + (pt.Z - this.Center.Z) * (pt.Z - this.Center.Z);

                if (distSq < this.Radius * this.Radius)
                {
                    double intensity;
                    if (DistanceAttenuationExponent == 0)
                    {
                        intensity = Intensity;
                    }
                    else if (DistanceAttenuationExponent == 2)
                    {
                        intensity = Intensity / ((point.X - this.VirtualSource.X) * (point.X - this.VirtualSource.X) + (point.Y - this.VirtualSource.Y) * (point.Y - this.VirtualSource.Y) + (point.Z - this.VirtualSource.Z) * (point.Z - this.VirtualSource.Z));
                    }
                    else
                    {
                        intensity = Intensity / Math.Pow((point.X - this.VirtualSource.X) * (point.X - this.VirtualSource.X) + (point.Y - this.VirtualSource.Y) * (point.Y - this.VirtualSource.Y) + (point.Z - this.VirtualSource.Z) * (point.Z - this.VirtualSource.Z), 0.5 * DistanceAttenuationExponent);
                    }

                    return new LightIntensity(intensity, (point - this.VirtualSource).Normalize());
                }
                else if (distSq < this.PenumbraRadius * this.PenumbraRadius)
                {
                    double intensity;
                    if (DistanceAttenuationExponent == 0)
                    {
                        intensity = Intensity;
                    }
                    else if (DistanceAttenuationExponent == 2)
                    {
                        intensity = Intensity / ((point.X - this.VirtualSource.X) * (point.X - this.VirtualSource.X) + (point.Y - this.VirtualSource.Y) * (point.Y - this.VirtualSource.Y) + (point.Z - VirtualSource.Z) * (point.Z - this.VirtualSource.Z));
                    }
                    else
                    {
                        intensity = Intensity / Math.Pow((point.X - this.VirtualSource.X) * (point.X - this.VirtualSource.X) + (point.Y - this.VirtualSource.Y) * (point.Y - this.VirtualSource.Y) + (point.Z - this.VirtualSource.Z) * (point.Z - this.VirtualSource.Z), 0.5 * DistanceAttenuationExponent);
                    }

                    if (PenumbraAttenuationExponent == 0)
                    {
                        //intensity *= 1;
                    }
```

```
                    else if (PenumbraAttenuationExponent == 1)
                    {
                        double factor = (Math.Sqrt(distSq) - this.Radius) / (this.PenumbraRadius - this.Radius);
                        intensity *= 1 - factor;
                    }
                    else
                    {
                        double factor = (Math.Sqrt(distSq) - this.Radius) / (this.PenumbraRadius - this.Radius);
                        intensity *= Math.Pow(1 - factor, PenumbraAttenuationExponent);
                    }

                    return new LightIntensity(intensity, (point - this.VirtualSource).Normalize());
                }
                else
                {
                    return new LightIntensity(0, (point - this.VirtualSource).Normalize());
                }
            }
        }

        /// <inheritdoc/>
        public double GetObstruction(Point3D point, IEnumerable<Triangle3DElement> shadowingTriangles)
        {
            double totalObstruction = 0;
            int sampleCount = 0;

            {
                Vector3D reverseDir = this.VirtualSource - point;

                double denom = (reverseDir * this.Direction);

                if (denom == 0)
                {
                    //totalObstruction += 0;
                    sampleCount++;
                }
                else
                {
                    double d = this.SourceDistance / denom;
                    Point3D pt = this.VirtualSource + d * reverseDir;

                    double maxD = (point - pt).Modulus;
                    NormalizedVector3D reverseDirNorm = new NormalizedVector3D(reverseDir.X / maxD, reverseDir.Y / maxD, reverseDir.Z / maxD, false);

                    bool found = false;

                    foreach (Triangle3DElement triangle in shadowingTriangles)
                    {
                        Point3D? projected = triangle.ProjectOnThisPlane(point, reverseDirNorm, true, maxD);

                        if (projected != null && Intersections3D.PointInTriangle(projected.Value, triangle.Point1, triangle.Point2, triangle.Point3))
                        {
                            totalObstruction += 1;
                            sampleCount++;
                            found = true;
                            break;
                        }
                    }

                    if (!found)
                    {
                        //totalObstruction += 0;
                        sampleCount++;
                    }
                }
            }


            for (int i = 0; i < ShadowSamplingPoints.Length; i++)
            {
                Vector3D reverseDir = ShadowSamplingPoints[i] - point;

                double maxD = reverseDir.Modulus;
                NormalizedVector3D reverseDirNorm = new NormalizedVector3D(reverseDir.X / maxD, reverseDir.Y / maxD, reverseDir.Z / maxD, false);

                bool found = false;

                foreach (Triangle3DElement triangle in shadowingTriangles)
                {
                    Point3D? projected = triangle.ProjectOnThisPlane(point, reverseDirNorm, true, maxD);
```

```

                    if (projected != null && Intersections3D.PointInTriangle(projected.Value, triangle.Point1, triangle.Point2, triangle.Point3))
                    {
                        totalObstruction += 1;
                        sampleCount++;
                        found = true;
                        break;
                    }
                }

                if (!found)
                {
                    //totalObstruction += 0;
                    sampleCount++;
                }
            }

            return totalObstruction / sampleCount;
        }
    }
}
```

8.26 Materials.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Runtime.InteropServices.ComTypes;
using System.Security.Cryptography.X509Certificates;
using System.Text;

namespace VectSharp.ThreeD
{
    /// <summary>
    /// Represents a material used to the determine the appearance of <see cref="Triangle3DElement"/>.
    /// </summary>
    public interface IMaterial
    {
        /// <summary>
        /// Obtains the <see cref="Colour"/> at the specified point.
        /// </summary>
        /// <param name="point">The point whose colour should be determined.</param>
        /// <param name="surfaceNormal">The normal to the surface at the specified <paramref name="point"/>.</param>
        /// <param name="camera">The camera being used to render the scene.</param>
        /// <param name="lights">A list of light sources that are present in the scene.</param>
        /// <param name="obstructions">A list of values indicating how obstructed each light source is.</param>
        /// <returns>The <see cref="Colour"/> of the specified point.</returns>
        Colour GetColour(Point3D point, NormalizedVector3D surfaceNormal, Camera camera, IList<ILightSource> lights, IList<double> obstructions);
    }

    /// <summary>
    /// Represents a material that always has the same colour, regardless of light.
    /// </summary>
    public class ColourMaterial : IMaterial
    {
        /// <summary>
        /// The colour of the material.
        /// </summary>
        public Colour Colour { get; }

        /// <summary>
```

```
        /// Creates a new <see cref="ColourMaterial"/> instance.
        /// </summary>
        /// <param name="colour">The colour of the material.</param>
        public ColourMaterial(Colour colour)
        {
            this.Colour = colour;
        }

        /// <inheritdoc/>
        public Colour GetColour(Point3D point, NormalizedVector3D surfaceNormal, Camera camera, IList<ILightSource> lights, IList<double> obstructions)
        {
            return Colour;
        }
    }

    /// <summary>
    /// Represents a material that uses a Phong reflection model to determine the colour of the material based on the light sources that hit it.
    /// </summary>
    public class PhongMaterial : IMaterial
    {
        /// <summary>
        /// The base colour of the material.
        /// </summary>
        public Colour Colour { get; }

        private (double L, double a, double b) LabColour;
        private (double H, double S, double L) LabBlackHSL;
        private (double H, double S, double L) LabWhiteHSL;
        private double IntensityExponent;
        private double TotalLength;

        /// <summary>
        /// A coefficient determining how much ambient light is reflected by the material.
        /// </summary>
        public double AmbientReflectionCoefficient { get; set; } = 1;

        /// <summary>
        /// A coefficient determining how much directional light is reflected by the material.
        /// </summary>
        public double DiffuseReflectionCoefficient { get; set; } = 1;

        /// <summary>
        /// A coefficient determining the intensity of specular highlights.
        /// </summary>
        public double SpecularReflectionCoefficient { get; set; } = 1;

        /// <summary>
        /// A coefficient determining the extent of specular highlights.
        /// </summary>
        public double SpecularShininess { get; set; } = 1;

        /// <summary>
        /// Creates a new <see cref="PhongMaterial"/> instance.
        /// </summary>
        /// <param name="colour">The base colour of the material.</param>
        public PhongMaterial(Colour colour)
        {
            this.Colour = colour;
            this.LabColour = colour.ToLab();
            this.LabBlackHSL = Colour.FromLab(0, LabColour.a, LabColour.b).ToHSL();
            this.LabWhiteHSL = Colour.FromLab(1, LabColour.a, LabColour.b).ToHSL();
            this.TotalLength = 1 + LabBlackHSL.L + (1 - LabWhiteHSL.L);
            this.IntensityExponent = Math.Log((LabBlackHSL.L + LabColour.L) / TotalLength) / Math.Log(0.5);
        }

        private Colour GetScaledColour(double intensity)
        {
            intensity = Math.Max(Math.Min(intensity, 1), 0);

            double pos = Math.Pow(intensity, IntensityExponent) * TotalLength;

            if (pos <= LabBlackHSL.L)
            {
                return Colour.FromHSL(LabBlackHSL.H, LabBlackHSL.S, pos).WithAlpha(Colour.A);
            }
            else if (pos >= 1 + LabBlackHSL.L)
            {
                return Colour.FromHSL(LabWhiteHSL.H, LabWhiteHSL.S, LabWhiteHSL.L + pos - 1 - LabBlackHSL.L).WithAlpha(Colour.A);
            }
            else
            {
                return Colour.FromLab(pos - LabBlackHSL.L, LabColour.a, LabColour.b).WithAlpha(Colour.A);
```

```
            }
        }

        /// <inheritdoc/>
        public Colour GetColour(Point3D point, NormalizedVector3D surfaceNormal, Camera camera, IList<ILightSource> lights, IList<double> obstructions)
        {
            double intensity = 0;

            for (int i = 0; i < lights.Count; i++)
            {
                (double lightIntensity, NormalizedVector3D lightDirection) = lights[i].GetLightAt(point);
                lightIntensity *= 1 - obstructions[i];

                if (double.IsNaN(lightDirection.X))
                {
                    intensity += lightIntensity * AmbientReflectionCoefficient;
                }
                else
                {
                    double dotProd = lightDirection * surfaceNormal;

                    intensity += lightIntensity * Math.Max(0, dotProd) * DiffuseReflectionCoefficient;

                    if (dotProd > 0)
                    {
                        NormalizedVector3D mirroredDirection = (lightDirection - 2 * dotProd * surfaceNormal).Normalize();

                        NormalizedVector3D cameraDirection = (camera.ViewPoint - point).Normalize();

                        double dotProd2 = mirroredDirection * cameraDirection;

                        if (dotProd2 >= 0)
                        {
                            intensity += lightIntensity * Math.Pow(dotProd2, SpecularShininess) * SpecularReflectionCoefficient;
                        }
                    }
                }
            }

            return GetScaledColour(intensity);
        }
    }
}
```

8.27 ObjectFactory.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace VectSharp.ThreeD
{
    /// <summary>
    /// A static class containing methods to create complex 3D objects.
    /// </summary>
    public static class ObjectFactory
    {
        /// <summary>
        /// Creates a cube.
        /// </summary>
        /// <param name="center">The centre of the cube.</param>
```

```
        /// <param name="size">The length of each side of the cube.</param>
        /// <param name="fill">A collection of materials that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="tag">A tag that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="zIndex">A z-index that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <returns>A list of <see cref="Triangle3DElement"/>s that constitute the cube.</returns>
        public static List<Element3D> CreateCube(Point3D center, double size, IEnumerable<IMaterial> fill, string tag = null, int zIndex = 0)
        {
            return CreateCuboid(center, size, size, size, fill, tag, zIndex);
        }

        /// <summary>
        /// Creates a cuboid.
        /// </summary>
        /// <param name="center">The centre of the cube.</param>
        /// <param name="sizeX">The length of the sides of the cube parallel to the x axis.</param>
        /// <param name="sizeY">The length of the sides of the cube parallel to the y axis.</param>
        /// <param name="sizeZ">The length of the sides of the cube parallel to the z axis.</param>
        /// <param name="fill">A collection of materials that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="tag">A tag that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="zIndex">A z-index that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <returns>A list of <see cref="Triangle3DElement"/>s that constitute the cuboid.</returns>
        public static List<Element3D> CreateCuboid(Point3D center, double sizeX, double sizeY, double sizeZ, IEnumerable<IMaterial> fill, string tag = null, int zIndex = 0)
        {
            Point3D A = new Point3D(center.X - sizeX * 0.5, center.Y - sizeY * 0.5, center.Z - sizeZ * 0.5);
            Point3D B = new Point3D(center.X + sizeX * 0.5, center.Y - sizeY * 0.5, center.Z - sizeZ * 0.5);
            Point3D C = new Point3D(center.X + sizeX * 0.5, center.Y + sizeY * 0.5, center.Z - sizeZ * 0.5);
            Point3D D = new Point3D(center.X - sizeX * 0.5, center.Y + sizeY * 0.5, center.Z - sizeZ * 0.5);

            Point3D E = new Point3D(center.X - sizeX * 0.5, center.Y - sizeY * 0.5, center.Z + sizeZ * 0.5);
            Point3D F = new Point3D(center.X + sizeX * 0.5, center.Y - sizeY * 0.5, center.Z + sizeZ * 0.5);
            Point3D G = new Point3D(center.X + sizeX * 0.5, center.Y + sizeY * 0.5, center.Z + sizeZ * 0.5);
            Point3D H = new Point3D(center.X - sizeX * 0.5, center.Y + sizeY * 0.5, center.Z + sizeZ * 0.5);


            List<Element3D> tbr = new List<Element3D>();

            tbr.AddRange(CreateRectangle(A, B, C, D, fill, tag, zIndex));
            tbr.AddRange(CreateRectangle(H, G, F, E, fill, tag, zIndex));

            tbr.AddRange(CreateRectangle(E, F, B, A, fill, tag, zIndex));
            tbr.AddRange(CreateRectangle(G, H, D, C, fill, tag, zIndex));

            tbr.AddRange(CreateRectangle(A, D, H, E, fill, tag, zIndex));
            tbr.AddRange(CreateRectangle(F, G, C, B, fill, tag, zIndex));

            return tbr;
        }

        /// <summary>
        /// Creates a quadrilater. All the vertices need not be coplanar.
        /// </summary>
        /// <param name="point1">The first vertex of the quadrilater.</param>
        /// <param name="point2">The second vertex of the quadrilater.</param>
        /// <param name="point3">The third vertex of the quadrilater.</param>
        /// <param name="point4">The fourth vertex of the quadrilater.</param>
        /// <param name="fill">A collection of materials that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="tag">A tag that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="zIndex">A z-index that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <returns>A list containing two <see cref="Triangle3DElement"/>s representing the quadrilater.</returns>
        public static List<Element3D> CreateRectangle(Point3D point1, Point3D point2, Point3D point3, Point3D point4, IEnumerable<IMaterial> fill, string tag = null, int zIndex = 0)
        {
            Triangle3DElement triangle1 = new Triangle3DElement(point1, point2, point3);
            triangle1.Fill.AddRange(fill);
            triangle1.Tag = tag;
            triangle1.ZIndex = zIndex;

```

```
            Triangle3DElement triangle2 = new Triangle3DElement(point1, point3, point4);
            triangle2.Fill.AddRange(fill);
            triangle2.Tag = tag;
            triangle2.ZIndex = zIndex;

            return new List<Element3D> { triangle1, triangle2 };
        }

        /// <summary>
        /// Creates a quadrilater, specifying the vertex normals at the four vertices. All the vertices need not be coplanar.
        /// </summary>
        /// <param name="point1">The first vertex of the quadrilater.</param>
        /// <param name="point2">The second vertex of the quadrilater.</param>
        /// <param name="point3">The third vertex of the quadrilater.</param>
        /// <param name="point4">The fourth vertex of the quadrilater.</param>
        /// <param name="point1Normal">The vertex normal at the first vertex of the quadrilater.</param>
        /// <param name="point2Normal">The vertex normal at the second vertex of the quadrilater.</param>
        /// <param name="point3Normal">The vertex normal at the third vertex of the quadrilater.</param>
        /// <param name="point4Normal">The vertex normal at the fourth vertex of the quadrilater.</param>
        /// <param name="fill">A collection of materials that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="tag">A tag that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="zIndex">A z-index that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <returns>A list containing two <see cref="Triangle3DElement"/>s representing the quadrilater.</returns>
        public static List<Element3D> CreateRectangle(Point3D point1, Point3D point2, Point3D point3, Point3D point4, NormalizedVector3D point1Normal, NormalizedVector3D point2Normal, NormalizedVector3D point3Normal, NormalizedVector3D point4Normal, IEnumerable<IMaterial> fill, string tag = null, int zIndex = 0)
        {
            Triangle3DElement triangle1 = new Triangle3DElement(point1, point2, point3, point1Normal, point2Normal, point3Normal);
            triangle1.Fill.AddRange(fill);
            triangle1.Tag = tag;
            triangle1.ZIndex = zIndex;

            Triangle3DElement triangle2 = new Triangle3DElement(point1, point3, point4, point1Normal, point3Normal, point4Normal);
            triangle2.Fill.AddRange(fill);
            triangle2.Tag = tag;
            triangle2.ZIndex = zIndex;

            return new List<Element3D> { triangle1, triangle2 };
        }

        /// <summary>
        /// Creates a sphere.
        /// </summary>
        /// <param name="center">The centre of the sphere.</param>
        /// <param name="radius">The radius of the sphere.</param>
        /// <param name="steps">The number of meridians and parallels to use when generating the sphere.</param>
        /// <param name="fill">A collection of materials that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="tag">A tag that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="zIndex">A z-index that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <returns>A list of <see cref="Triangle3DElement"/>s that constitute the sphere.</returns>
        public static List<Element3D> CreateSphere(Point3D center, double radius, int steps, IEnumerable<IMaterial> fill, string tag = null, int zIndex = 0)
        {
            List<Point3D> points = new List<Point3D>();

            for (int t = 0; t <= steps; t++)
            {
                for (int p = 0; p < steps * 2; p++)
                {
                    double theta = Math.PI / steps * t;
                    double phi = Math.PI / steps * p;

                    double x = center.X + radius * Math.Sin(theta) * Math.Cos(phi);
                    double y = center.Y + radius * Math.Sin(theta) * Math.Sin(phi);
                    double z = center.Z + radius * Math.Cos(theta);

                    points.Add(new Point3D(x, y, z));

                    if (t == 0 || t == steps)
                    {
                        break;
```

```
                    }
                }
            }

            List<Element3D> tbr = new List<Element3D>(4 * steps + (points.Count - 2 - 2 * steps) * 2);

            for (int i = 0; i < points.Count - 1; i++)
            {
                if (i == 0)
                {
                    for (int j = 0; j < 2 * steps; j++)
                    {
                        Point3D p1 = points[i];
                        Point3D p3 = points[i + 1 + j];
                        Point3D p2 = points[i + 1 + (j + 1) % (2 * steps)];

                        Triangle3DElement tri = new Triangle3DElement(p1, p2, p3, (center - p1).Normalize(), (center - p2).Normalize(), (center - p3).Normalize());
                        tri.Fill.AddRange(fill);
                        tri.Tag = tag;
                        tri.ZIndex = zIndex;
                        tbr.Add(tri);
                    }
                }
                else if (i >= points.Count - 1 - 2 * steps)
                {
                    Point3D p1 = points[i];
                    Point3D p3 = points[points.Count - 1];
                    Point3D p2 = points[points.Count - 1 - 2 * steps + (i - (points.Count - 1 - 2 * steps) + 1) % (2 * steps)];

                    Triangle3DElement tri = new Triangle3DElement(p1, p2, p3, (center - p1).Normalize(), (center - p2).Normalize(), (center - p3).Normalize());
                    tri.Fill.AddRange(fill);
                    tri.Tag = tag;
                    tri.ZIndex = zIndex;
                    tbr.Add(tri);
                }
                else
                {
                    if ((i - 1) % (2 * steps) < 2 * steps - 1)
                    {
                        Point3D p4 = points[i + 2 * steps];
                        Point3D p3 = points[i + 2 * steps + 1];
                        Point3D p2 = points[i + 1];
                        Point3D p1 = points[i];

                        tbr.AddRange(CreateRectangle(p1, p2, p3, p4, (center - p1).Normalize(), (center - p2).Normalize(), (center - p3).Normalize(), (center - p4).Normalize(), fill, tag, zIndex));
                    }
                    else
                    {
                        Point3D p4 = points[i + 2 * steps];
                        Point3D p3 = points[(i / (2 * steps)) * 2 * steps + 1];
                        Point3D p2 = points[(i / (2 * steps) - 1) * 2 * steps + 1];
                        Point3D p1 = points[i];

                        tbr.AddRange(CreateRectangle(p1, p2, p3, p4, (center - p1).Normalize(), (center - p2).Normalize(), (center - p3).Normalize(), (center - p4).Normalize(), fill, tag, zIndex));
                    }
                }
            }

            return tbr;
        }

        /// <summary>
        /// Creates a tetrahedron inscribed in a sphere.
        /// </summary>
        /// <param name="center">The centre of the tetrahedron.</param>
        /// <param name="radius">The radius of the sphere in which the tetrahedron is inscribed.</param>
        /// <param name="fill">A collection of materials that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="tag">A tag that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="zIndex">A z-index that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <returns>A list of <see cref="Triangle3DElement"/>s that constitute the sphere.</returns>
        public static List<Element3D> CreateTetrahedron(Point3D center, double radius, IEnumerable<IMaterial> fill, string tag = null, int zIndex = 0)
        {
            Point3D tip = new Point3D(center.X, center.Y - radius, center.Z);
            Point3D base1 = new Point3D(Math.Sqrt(8.0 / 9) * radius + center.X, center.Y + radius / 3, center.Z);
            Point3D base2 = new Point3D(-Math.Sqrt(2.0 / 9) * radius + center.X, center.Y + radius / 3, center.Z + Math.Sqrt(2.0 / 3) * radius);
```

```
            Point3D base3 = new Point3D(-Math.Sqrt(2.0 / 9) * radius + center.X, center.Y + radius / 3, center.Z - Math.Sqrt(2.0 / 3) * radius);

            Triangle3DElement faceTriangle1 = new Triangle3DElement(tip, base2, base1) { Tag = tag, ZIndex = zIndex };
            faceTriangle1.Fill.AddRange(fill);

            Triangle3DElement faceTriangle2 = new Triangle3DElement(tip, base3, base2) { Tag = tag, ZIndex = zIndex };
            faceTriangle2.Fill.AddRange(fill);

            Triangle3DElement faceTriangle3 = new Triangle3DElement(tip, base1, base3) { Tag = tag, ZIndex = zIndex };
            faceTriangle3.Fill.AddRange(fill);

            Triangle3DElement baseTriangle = new Triangle3DElement(base1, base2, base3) { Tag = tag, ZIndex = zIndex };
            baseTriangle.Fill.AddRange(fill);

            return new List<Element3D>() { faceTriangle1, faceTriangle2, faceTriangle3, baseTriangle };
        }

        /// <summary>
        /// Creates a flat polygon.
        /// </summary>
        /// <param name="polygon2D">A 2D <see cref="GraphicsPath"/> representing the polygon.</param>
        /// <param name="triangulationResolution">The resolution that will be used to linearise curve segments in the <see cref="GraphicsPath"/>.</param>
        /// <param name="origin">A <see cref="Point3D"/> that will correspond to the origin of the 2D reference system.</param>
        /// <param name="xAxis">A <see cref="NormalizedVector3D"/> that will correspond to the x axis of the 2D reference system. This will be orthonormalised to the <paramref name="yAxis"/>.</param>
        /// <param name="yAxis">A <see cref="NormalizedVector3D"/> that will correspond to the y axis of the 2D reference system.</param>
        /// <param name="reverseTriangles">Indicates whether the order of the points (and thus the normals) of all the triangles returned by this method should be reversed.</param>
        /// <param name="fill">A collection of materials that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="tag">A tag that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="zIndex">A z-index that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <returns>A list of <see cref="Triangle3DElement"/>s that constitute the polygon.</returns>
        public static List<Element3D> CreatePolygon(GraphicsPath polygon2D, double triangulationResolution, Point3D origin, NormalizedVector3D xAxis, NormalizedVector3D yAxis, bool reverseTriangles, IEnumerable<IMaterial> fill, string tag = null, int zIndex = 0)
        {
            xAxis = (xAxis - yAxis * (xAxis * yAxis)).Normalize();

            List<GraphicsPath> triangles = polygon2D.Triangulate(triangulationResolution, true).ToList();

            List<Element3D> tbr = new List<Element3D>(triangles.Count);

            for (int i = 0; i < triangles.Count; i++)
            {
                Point p1 = triangles[i].Segments[0].Point;
                Point p2 = triangles[i].Segments[1].Point;
                Point p3 = triangles[i].Segments[2].Point;

                Point3D p13D = origin + xAxis * p1.X + yAxis * p1.Y;
                Point3D p23D = origin + xAxis * p2.X + yAxis * p2.Y;
                Point3D p33D = origin + xAxis * p3.X + yAxis * p3.Y;

                Triangle3DElement t = !reverseTriangles ? new Triangle3DElement(p13D, p23D, p33D) : new Triangle3DElement(p13D, p33D, p23D);
                t.Fill.AddRange(fill);
                t.Tag = tag;
                t.ZIndex = zIndex;
                tbr.Add(t);
            }

            return tbr;
        }

        /// <summary>
        /// Creates a prism with the specified base.
        /// </summary>
        /// <param name="polygonBase2D">A 2D <see cref="GraphicsPath"/> representing the base of the prism.</param>
        /// <param name="triangulationResolution">The resolution that will be used to linearise curve segments in the <see cref="GraphicsPath"/>.</param>
        /// <param name="bottomOrigin">A <see cref="Point3D"/> that will correspond to the origin of the 2D reference system of the bottom base.</param>
        /// <param name="topOrigin">A <see cref="Point3D"/> that will correspond to the origin of the 2D reference system of the top base.</param>
```

```
        /// <param name="baseXAxis">A <see cref="NormalizedVector3D"/> that will correspond to the x axis of the 2D reference system of the bases. This will be orthonormalised to the <paramref name="baseYAxis"/>.</param>
        /// <param name="baseYAxis">A <see cref="NormalizedVector3D"/> that will correspond to the y axis of the 2D reference system of the bases.</param>
        /// <param name="fill">A collection of materials that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="tag">A tag that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <param name="zIndex">A z-index that will be applied to the <see cref="Triangle3DElement"/>s returned by this method.</param>
        /// <returns>A list of <see cref="Triangle3DElement"/>s that constitute the prism.</returns>
        public static List<Element3D> CreatePrism(GraphicsPath polygonBase2D, double triangulationResolution, Point3D bottomOrigin, Point3D topOrigin, NormalizedVector3D baseXAxis, NormalizedVector3D baseYAxis, IEnumerable<IMaterial> fill, string tag = null, int zIndex = 0)
        {
            baseXAxis = (baseXAxis - baseYAxis * (baseXAxis * baseYAxis)).Normalize();

            List<Element3D> tbr = new List<Element3D>();

            bool orientation = (baseXAxis ^ baseYAxis) * (bottomOrigin - topOrigin) > 0;

            double[,] matrix1 = Matrix3D.RotationToAlignAWithB(new NormalizedVector3D(0, 1, 0), baseYAxis);
            double[,] matrix2 = Matrix3D.RotationToAlignAWithB(((Vector3D)(matrix1 * new Point3D(1, 0, 0))).Normalize(), baseXAxis);

            List<List<NormalizedVector3D» normals = (from el2 in polygonBase2D.GetLinearisationPointsNormals(triangulationResolution) select (from el in el2 select (el.X * baseXAxis + el.Y * baseYAxis).Normalize()).ToList()).ToList();

            polygonBase2D = polygonBase2D.Linearise(triangulationResolution);

            tbr.AddRange(CreatePolygon(polygonBase2D, triangulationResolution, bottomOrigin, baseXAxis, baseYAxis, orientation, fill, tag, zIndex));
            tbr.AddRange(CreatePolygon(polygonBase2D, triangulationResolution, topOrigin, baseXAxis, baseYAxis, !orientation, fill, tag, zIndex));

            List<List<Point3D» bottomPoints = (from el2 in polygonBase2D.GetPoints() select (from el in el2 select (Point3D)(el.X * baseXAxis + el.Y * baseYAxis + (Vector3D)bottomOrigin)).ToList()).ToList();
            List<List<Point3D» topPoints = (from el2 in polygonBase2D.GetPoints() select (from el in el2 select (Point3D)(el.X * baseXAxis + el.Y * baseYAxis + (Vector3D)topOrigin)).ToList()).ToList();

            if (orientation)
            {
                for (int i = 0; i < bottomPoints.Count; i++)
                {
                    for (int j = 0; j < bottomPoints[i].Count - 1; j++)
                    {
                        tbr.AddRange(CreateRectangle(bottomPoints[i][j], bottomPoints[i][j + 1], topPoints[i][j + 1], topPoints[i][j], normals[i][j], normals[i][j + 1], normals[i][j + 1], normals[i][j], fill, tag, zIndex));
                    }
                }
            }
            else
            {
                for (int i = 0; i < bottomPoints.Count; i++)
                {
                    for (int j = 0; j < bottomPoints[i].Count - 1; j++)
                    {
                        tbr.AddRange(CreateRectangle(bottomPoints[i][j], topPoints[i][j], topPoints[i][j + 1], bottomPoints[i][j + 1], normals[i][j], normals[i][j], normals[i][j + 1], normals[i][j + 1], fill, tag, zIndex));
                    }
                }
            }

            return tbr;
        }

        /// <summary>
        /// Creates a wireframe from a collection of <see cref="Element3D"/>s.
        /// </summary>
        /// <param name="object3D">The collection of <see cref="Element3D"/>s. <see cref="Line3DElement"/>s and <see cref="Point3DElement"/>s are ignored.</param>
        /// <param name="colour">The colour of the <see cref="Line3DElement"/>s returned by this method.</param>
        /// <param name="thickness">The thickness of the <see cref="Line3DElement"/>s returned by this method.</param>
        /// <param name="lineCap">The line cap of the <see cref="Line3DElement"/>s returned by this method.</param>
        /// <param name="lineDash">The line dash of the <see cref="Line3DElement"/>s returned by this method.</param>
        /// <param name="tag">A tag that will be applied to the <see cref="Line3DElement"/>s returned by this method.</param>
```

```
        /// <param name="zIndex">A z-index that will be applied to the <see cref="Line3DElement"/>s returned by this method.</param>
        /// <returns>A list of <see cref="Line3DElement"/>s that constitute the wireframe.</returns>
        public static List<Element3D> CreateWireframe(IEnumerable<Element3D> object3D, Colour colour, double thickness = 1, LineCaps lineCap = LineCaps.Butt, LineDash? lineDash = null, string tag = null, int zIndex = 0)
        {
            List<Element3D> tbr = new List<Element3D>();

            List<Point3D[]> addedLines = new List<Point3D[]>();

            void addLine(Point3D p1, Point3D p2)
            {
                for (int i = 0; i < addedLines.Count; i++)
                {
                    if ((addedLines[i][0].Equals(p1, 1e-4) && addedLines[i][1].Equals(p2, 1e-4)) || (addedLines[i][0].Equals(p2, 1e-4) && addedLines[i][1].Equals(p1, 1e-4)))
                    {
                        return;
                    }
                }

                tbr.Add(new Line3DElement(p1, p2) { Colour = colour, Thickness = thickness, LineCap = lineCap, LineDash = lineDash ?? LineDash.SolidLine, Tag = tag, ZIndex = zIndex });
                addedLines.Add(new Point3D[] { p1, p2 });
            }

            foreach (Element3D element in object3D)
            {
                if (element is Triangle3DElement triangle)
                {
                    addLine(triangle.Point1, triangle.Point2);
                    addLine(triangle.Point2, triangle.Point3);
                    addLine(triangle.Point3, triangle.Point1);
                }
            }

            return tbr;
        }

        /// <summary>
        /// Obtains a list of <see cref="Point3DElement"/> corresponding to the vertices of a list of <see cref="Element3D"/>s.
        /// </summary>
        /// <param name="object3D">The collection of <see cref="Element3D"/>s. <see cref="Point3DElement"/>s are ignored.</param>
        /// <param name="colour">The colour of the <see cref="Point3DElement"/>s returned by this method.</param>
        /// <param name="diameter">The diameter of the <see cref="Point3DElement"/>s returned by this method.</param>
        /// <param name="tag">A tag that will be applied to the <see cref="Point3DElement"/>s returned by this method.</param>
        /// <param name="zIndex">A z-index that will be applied to the <see cref="Point3DElement"/>s returned by this method.</param>
        /// <returns>A list of <see cref="Point3DElement"/>s corresponding to the vertices of the <see cref="Element3D"/>s.</returns>
        public static List<Element3D> CreatePoints(IEnumerable<Element3D> object3D, Colour colour, double diameter = 1, string tag = null, int zIndex = 0)
        {
            List<Element3D> tbr = new List<Element3D>();

            List<Point3D> addedPoints = new List<Point3D>();

            void addPoint(Point3D p)
            {
                for (int i = 0; i < addedPoints.Count; i++)
                {
                    if (addedPoints[i].Equals(p, 1e-4))
                    {
                        return;
                    }
                }

                tbr.Add(new Point3DElement(p) { Colour = colour, Diameter = diameter, Tag = tag, ZIndex = zIndex });
                addedPoints.Add(p);
            }

            foreach (Element3D element in object3D)
            {
                if (!(element is Point3DElement))
                {
                    foreach (Point3D p in element)
                    {
                        addPoint(p);
                    }
                }
```

```
            }

            return tbr;
        }
    }
}
```

8.28 Scene.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;

namespace VectSharp.ThreeD
{
    /// <summary>
    /// Represents a 3D scene.
    /// </summary>
    public interface IScene
    {
        /// <summary>
        /// The <see cref="Element3D"/>s constituting the scene.
        /// </summary>
        IEnumerable<Element3D> SceneElements { get; }

        /// <summary>
        /// Adds the specified <paramref name="element"/> to the scene.
        /// </summary>
        /// <param name="element">The <see cref="Element3D"/> to add.</param>
        void AddElement(Element3D element);

        /// <summary>
        /// Adds the specified <paramref name="elements"/> to the scene.
        /// </summary>
        /// <param name="elements">A collection of <see cref="Element3D"/>s to add.</param>
        void AddRange(IEnumerable<Element3D> elements);

        /// <summary>
        /// Replaces each element in the scene with the element returned by the <paramref name="replacementFunction"/>.
        /// </summary>
        /// <param name="replacementFunction">A function replacing each <see cref="Element3D"/> in the scene with another <see cref="Element3D"/>.</param>
        void Replace(Func<Element3D, Element3D> replacementFunction);

        /// <summary>
        /// Replaces each element in the scene with the element(s) returned by the <paramref name="replacementFunction"/>.
        /// </summary>
        /// <param name="replacementFunction">A function replacing each <see cref="Element3D"/> in the scene with 0 or more <see cref="Element3D"/>s.</param>
        void Replace(Func<Element3D, IEnumerable<Element3D» replacementFunction);

        /// <summary>
        /// An object used to synchronise multithreaded rendering of the same scene.
        /// </summary>
        object SceneLock { get; }
    }

    /// <summary>
    /// Represents a 3D scene.
    /// </summary>
    public class Scene : IScene
    {
        /// <inheritdoc/>
        public object SceneLock { get; }

```

```
        private List<Element3D> sceneElements;

        /// <inheritdoc/>
        public IEnumerable<Element3D> SceneElements => sceneElements;

        /// <summary>
        /// Creates a new <see cref="Scene"/>.
        /// </summary>
        public Scene()
        {
            sceneElements = new List<Element3D>();
            this.SceneLock = new object();
        }

        /// <inheritdoc/>
        public void AddElement(Element3D element)
        {
            this.sceneElements.Add(element);
        }

        /// <inheritdoc/>
        public void AddRange(IEnumerable<Element3D> elements)
        {
            this.sceneElements.AddRange(elements);
        }

        /// <inheritdoc/>
        public void Replace(Func<Element3D, Element3D> replacementFunction)
        {
            for (int i = 0; i < sceneElements.Count; i++)
            {
                sceneElements[i] = replacementFunction(sceneElements[i]);
            }
        }

        /// <inheritdoc/>
        public void Replace(Func<Element3D, IEnumerable<Element3D» replacementFunction)
        {
            List<Element3D> newElements = new List<Element3D>(sceneElements.Count);

            for (int i = 0; i < this.sceneElements.Count; i++)
            {
                newElements.AddRange(replacementFunction(sceneElements[i]));
            }

            newElements.TrimExcess();

            this.sceneElements = newElements;
        }
    }
}
```

8.29 Animation.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2022 Giorgio Bianchini, University of Bristol

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Diagnostics;
using System.Diagnostics.SymbolStore;
using System.IO;
using System.Linq;
using System.Net.Http.Headers;
using System.Text;
using System.Threading.Tasks;
using VectSharp.Filters;
```

```
namespace VectSharp
{
    internal interface IAnimation
    {
        IGraphicsAction StartValue { get; }
        IGraphicsAction EndValue { get; }
        IGraphicsAction Interpolate(double position, Dictionary<string, IEasing> easings);
    }

    internal abstract class IAnimation<T> : IAnimation where T : IGraphicsAction
    {
        public abstract T StartValue { get; }
        public abstract T EndValue { get; }

        IGraphicsAction IAnimation.StartValue => (IGraphicsAction)StartValue;

        IGraphicsAction IAnimation.EndValue => (IGraphicsAction)EndValue;

        public abstract T Interpolate(double position, Dictionary<string, IEasing> easings);

        IGraphicsAction IAnimation.Interpolate(double position, Dictionary<string, IEasing> easings)
        {
            return (IGraphicsAction)Interpolate(position, easings);
        }
    }

    internal class ConstantAnimation<T> : IAnimation<T> where T : IGraphicsAction
    {
        public override T StartValue { get; }

        public override T EndValue => StartValue;

        public ConstantAnimation(T action)
        {
            this.StartValue = action;
        }

        public override T Interpolate(double position, Dictionary<string, IEasing> easings)
        {
            return this.StartValue;
        }
    }

    internal class StateAnimation : IAnimation<StateAction>
    {
        public override StateAction StartValue { get; }

        public override StateAction EndValue { get; }

        public StateAnimation(StateAction startValue, StateAction endValue)
        {
            this.StartValue = startValue;
            this.EndValue = endValue;
        }

        public override StateAction Interpolate(double position, Dictionary<string, IEasing> easings)
        {
            if (easings != null && !string.IsNullOrEmpty(StartValue.Tag) &&
easings.TryGetValue(StartValue.Tag, out IEasing easing))
            {
                position = easing.Ease(position);
            }

            if (position < 0.5)
            {
                return StartValue;
            }
            else
            {
                return EndValue;
            }
        }
    }

    internal static class InterpolationUtils
    {
        public static Point InterpolatePoint(Point start, Point end, double position)
        {
            return new Point(start.X * (1 - position) + end.X * position, start.Y * (1 - position) +
end.Y * position);
        }

        public static Point InterpolatePoint(Point? start, Point? end, double position)
        {
            return InterpolatePoint(start ?? new Point(), end ?? new Point(), position);
        }
```

```
        public static Size InterpolateSize(Size start, Size end, double position)
        {
            return new Size(start.Width * (1 - position) + end.Width * position, start.Height * (1 - position) + end.Height * position);
        }

        public static Size InterpolateSize(Size? start, Size? end, double position)
        {
            return InterpolateSize(start ?? new Size(), end ?? new Size(), position);
        }

        public static double InterpolateDouble(double start, double end, double position)
        {
            return start * (1 - position) + end * position;
        }

        public static double InterpolateDouble(double? start, double? end, double position)
        {
            return start * (1 - position) + end * position ?? 0;
        }

        public static Colour InterpolateColour(Colour start, Colour end, double position)
        {
            return Colour.FromRgba(InterpolateDouble(start.R, end.R, position),
                    InterpolateDouble(start.G, end.G, position),
                    InterpolateDouble(start.B, end.B, position),
                    InterpolateDouble(start.A, end.A, position));
        }

        private static HashSet<double> GetAnchors(Brush start, Brush end)
        {
            HashSet<double> anchors = new HashSet<double>();

            if (start is GradientBrush gradientStart)
            {
                for (int i = 0; i < gradientStart.GradientStops.Count; i++)
                {
                    anchors.Add(gradientStart.GradientStops[i].Offset);
                }
            }

            if (end is GradientBrush gradientEnd)
            {
                for (int i = 0; i < gradientEnd.GradientStops.Count; i++)
                {
                    anchors.Add(gradientEnd.GradientStops[i].Offset);
                }
            }

            return anchors;
        }

        private static void AddGradientStop(List<GradientStop> stops)
        {
            int minIndex = -1;
            double maxInterval = double.MinValue;

            for (int i = 1; i < stops.Count; i++)
            {
                double interval = stops[i].Offset - stops[i - 1].Offset;

                if (interval > maxInterval)
                {
                    maxInterval = interval;
                    minIndex = i;
                }
            }

            if (minIndex > 0)
            {
                stops.Insert(minIndex, new GradientStop(Colour.FromRgba((stops[minIndex - 1].Colour.R + stops[minIndex].Colour.R) * 0.5, (stops[minIndex - 1].Colour.G + stops[minIndex].Colour.G) * 0.5, (stops[minIndex - 1].Colour.B + stops[minIndex].Colour.B) * 0.5, (stops[minIndex - 1].Colour.A + stops[minIndex].Colour.A) * 0.5), stops[minIndex - 1].Offset + maxInterval * 0.5));
            }
        }

        public static Brush InterpolateBrush(Brush start, Brush end, double position)
        {
            if (position <= 0)
            {
                return start;
            }
            else if (position >= 1)
            {
                return end;
```

```
            }
            else
            {
                HashSet<double> anchors = GetAnchors(start, end);

                if (start is SolidColourBrush solidStart && end is SolidColourBrush solidEnd)
                {
                    return new SolidColourBrush(InterpolateColour(solidStart.Colour, solidEnd.Colour, position));
                }
                else if (start is LinearGradientBrush linearStart && end is LinearGradientBrush linearEnd)
                {
                    /*List<GradientStop> gradientStops = new List<GradientStop>(anchors.Count);

                    foreach (double anchor in anchors)
                    {
                        Colour startColour = linearStart.GradientStops.GetColourAt(anchor);
                        Colour endColour = linearEnd.GradientStops.GetColourAt(anchor);

                        gradientStops.Add(new GradientStop(InterpolateColour(startColour, endColour, position), anchor));
                    }

                    return new LinearGradientBrush(InterpolatePoint(linearStart.StartPoint, linearEnd.StartPoint, position), InterpolatePoint(linearStart.EndPoint, linearEnd.EndPoint, position), gradientStops);*/

                    List<GradientStop> startStops = linearStart.GradientStops.ToList();
                    List<GradientStop> endStops = linearEnd.GradientStops.ToList();

                    while (startStops.Count < endStops.Count)
                    {
                        AddGradientStop(startStops);
                    }

                    while (startStops.Count > endStops.Count)
                    {
                        AddGradientStop(endStops);
                    }

                    List<GradientStop> interpolatedStops = new List<GradientStop>();

                    for (int i = 0; i < startStops.Count; i++)
                    {
                        interpolatedStops.Add(new GradientStop(InterpolateColour(startStops[i].Colour, endStops[i].Colour, position), InterpolateDouble(startStops[i].Offset, endStops[i].Offset, position)));
                    }

                    return new LinearGradientBrush(InterpolatePoint(linearStart.StartPoint, linearEnd.StartPoint, position), InterpolatePoint(linearStart.EndPoint, linearEnd.EndPoint, position), interpolatedStops);
                }
                else if (start is SolidColourBrush solidStart2 && end is LinearGradientBrush linearEnd2)
                {
                    List<GradientStop> gradientStops = new List<GradientStop>(anchors.Count);

                    foreach (double anchor in anchors)
                    {
                        Colour startColour = solidStart2.Colour;
                        Colour endColour = linearEnd2.GradientStops.GetColourAt(anchor);

                        gradientStops.Add(new GradientStop(InterpolateColour(startColour, endColour, position), anchor));
                    }

                    return new LinearGradientBrush(linearEnd2.StartPoint, linearEnd2.EndPoint, gradientStops);
                }
                else if (start is LinearGradientBrush linearStart2 && end is SolidColourBrush solidEnd2)
                {
                    List<GradientStop> gradientStops = new List<GradientStop>(anchors.Count);

                    foreach (double anchor in anchors)
                    {
                        Colour startColour = linearStart2.GradientStops.GetColourAt(anchor);
                        Colour endColour = solidEnd2.Colour;

                        gradientStops.Add(new GradientStop(InterpolateColour(startColour, endColour, position), anchor));
                    }

                    return new LinearGradientBrush(linearStart2.StartPoint, linearStart2.EndPoint, gradientStops);
```

```
                }
                else if (start is RadialGradientBrush radialStart && end is RadialGradientBrush radialEnd)
                {
                    /*List<GradientStop> gradientStops = new List<GradientStop>(anchors.Count);

                    foreach (double anchor in anchors)
                    {
                        Colour startColour = radialStart.GradientStops.GetColourAt(anchor);
                        Colour endColour = radialEnd.GradientStops.GetColourAt(anchor);

                        gradientStops.Add(new GradientStop(InterpolateColour(startColour, endColour, position), anchor));
                    }

                    return new RadialGradientBrush(InterpolatePoint(radialStart.FocalPoint, radialEnd.FocalPoint, position), InterpolatePoint(radialStart.Centre, radialEnd.Centre, position), InterpolateDouble(radialStart.Radius, radialEnd.Radius, position), gradientStops);*/

                    List<GradientStop> startStops = radialStart.GradientStops.ToList();
                    List<GradientStop> endStops = radialEnd.GradientStops.ToList();

                    while (startStops.Count < endStops.Count)
                    {
                        AddGradientStop(startStops);
                    }

                    while (startStops.Count > endStops.Count)
                    {
                        AddGradientStop(endStops);
                    }

                    List<GradientStop> interpolatedStops = new List<GradientStop>();

                    for (int i = 0; i < startStops.Count; i++)
                    {
                        interpolatedStops.Add(new GradientStop(InterpolateColour(startStops[i].Colour, endStops[i].Colour, position), InterpolateDouble(startStops[i].Offset, endStops[i].Offset, position)));
                    }

                    return new RadialGradientBrush(InterpolatePoint(radialStart.FocalPoint, radialEnd.FocalPoint, position), InterpolatePoint(radialStart.Centre, radialEnd.Centre, position), InterpolateDouble(radialStart.Radius, radialEnd.Radius, position), interpolatedStops);
                }
                else if (start is SolidColourBrush solidStart3 && end is RadialGradientBrush radialEnd2)
                {
                    List<GradientStop> gradientStops = new List<GradientStop>(anchors.Count);

                    foreach (double anchor in anchors)
                    {
                        Colour startColour = solidStart3.Colour;
                        Colour endColour = radialEnd2.GradientStops.GetColourAt(anchor);

                        gradientStops.Add(new GradientStop(InterpolateColour(startColour, endColour, position), anchor));
                    }

                    return new RadialGradientBrush(radialEnd2.FocalPoint, radialEnd2.Centre, radialEnd2.Radius, gradientStops);
                }
                else if (start is RadialGradientBrush radialStart2 && end is SolidColourBrush solidEnd3)
                {
                    List<GradientStop> gradientStops = new List<GradientStop>(anchors.Count);

                    foreach (double anchor in anchors)
                    {
                        Colour startColour = radialStart2.GradientStops.GetColourAt(anchor);
                        Colour endColour = solidEnd3.Colour;

                        gradientStops.Add(new GradientStop(InterpolateColour(startColour, endColour, position), anchor));
                    }

                    return new RadialGradientBrush(radialStart2.FocalPoint, radialStart2.Centre, radialStart2.Radius, gradientStops);
                }
                else if (start is LinearGradientBrush linearStart3 && end is RadialGradientBrush radialEnd3)
                {
                    Point linearVector = new Point(linearStart3.EndPoint.X - linearStart3.StartPoint.X, linearStart3.EndPoint.Y - linearStart3.StartPoint.Y);

                    double modulus = linearVector.Modulus();
                    linearVector = new Point(linearVector.X / modulus, linearVector.Y / modulus);
```

```
                    double radius = radialEnd3.Radius / position;// radialEnd3.Radius * (1 + 100 * (1 - position));

                    Point target = new Point(linearStart3.EndPoint.X - linearVector.X * radius, linearStart3.EndPoint.Y - linearVector.Y * radius);

                    List<GradientStop> gradientStops = new List<GradientStop>(anchors.Count);

                    double lOverR = (modulus + (radialEnd3.Radius - modulus) * position) / radialEnd3.Radius;

                    foreach (double anchor in anchors)
                    {
                        double offset = (1 - lOverR * position) + anchor * position * lOverR;

                        Colour startColour = linearStart3.GradientStops.GetColourAt(anchor);
                        Colour endColour = radialEnd3.GradientStops.GetColourAt(anchor);
                        gradientStops.Add(new GradientStop(InterpolateColour(startColour, endColour, position), offset));
                    }

                    return new RadialGradientBrush(InterpolatePoint(target, radialEnd3.FocalPoint, position), InterpolatePoint(target, radialEnd3.Centre, position), InterpolateDouble(radius, radialEnd3.Radius, position), gradientStops);
                }
                else if (start is RadialGradientBrush && end is LinearGradientBrush)
                {
                    return InterpolateBrush(end, start, 1 - position);
                }
                else
                {
                    if (position < 0.5)
                    {
                        return start;
                    }
                    else
                    {
                        return end;
                    }
                }
            }
        }

        public static int InterpolateInt(int start, int end, double position)
        {
            return (int)Math.Round(start * (1 - position) + end * position);
        }

        public static Font InterpolateFont(Font start, Font end, double position)
        {
            Font tbr = new Font(position < 0.5 ? start.FontFamily : end.FontFamily, InterpolateDouble(start.FontSize, end.FontSize, position), start.Underline != null || end.Underline != null);

            if (start.Underline != null && end.Underline == null)
            {
                tbr.Underline.FollowItalicAngle = start.Underline.FollowItalicAngle;
                tbr.Underline.LineCap = start.Underline.LineCap;
                tbr.Underline.Position = start.Underline.Position;
                tbr.Underline.SkipDescenders = start.Underline.SkipDescenders;
                tbr.Underline.Thickness = InterpolateDouble(start.Underline.Thickness, 0, position);
            }
            else if (start.Underline == null && end.Underline != null)
            {
                tbr.Underline.FollowItalicAngle = end.Underline.FollowItalicAngle;
                tbr.Underline.LineCap = end.Underline.LineCap;
                tbr.Underline.Position = end.Underline.Position;
                tbr.Underline.SkipDescenders = end.Underline.SkipDescenders;
                tbr.Underline.Thickness = InterpolateDouble(0, end.Underline.Thickness, position);
            }
            else if (start.Underline != null && end.Underline != null)
            {
                tbr.Underline.FollowItalicAngle = position < 0.5 ? start.Underline.FollowItalicAngle : end.Underline.FollowItalicAngle;
                tbr.Underline.LineCap = position < 0.5 ? start.Underline.LineCap : end.Underline.LineCap;
                tbr.Underline.Position = InterpolateDouble(start.Underline.Position, end.Underline.Position, position);
                tbr.Underline.SkipDescenders = position < 0.5 ? start.Underline.SkipDescenders : end.Underline.SkipDescenders;
                tbr.Underline.Thickness = InterpolateDouble(start.Underline.Thickness, 0, position);
            }

            return tbr;
        }
    }
```

```
    internal class TransformAnimation : IAnimation<TransformAction>
    {
        public override TransformAction StartValue { get; }

        public override TransformAction EndValue { get; }

        public TransformAnimation(TransformAction startValue, TransformAction endValue)
        {
            this.StartValue = startValue;
            this.EndValue = endValue;
        }
        public override TransformAction Interpolate(double position, Dictionary<string, IEasing> easings)
        {
            if (easings != null && !string.IsNullOrEmpty(StartValue.Tag) && easings.TryGetValue(StartValue.Tag, out IEasing easing))
            {
                position = easing.Ease(position);
            }

            if (position <= 0)
            {
                return this.StartValue;
            }
            else if (position >= 1)
            {
                return this.EndValue;
            }
            else
            {
                if (this.StartValue.Delta != null && this.EndValue.Delta != null)
                {
                    return new TransformAction(InterpolationUtils.InterpolatePoint(this.StartValue.Delta, this.EndValue.Delta, position), position < 0.5 ? this.StartValue.Tag : this.EndValue.Tag);
                }
                else if (this.StartValue.Scale != null && this.EndValue.Scale != null)
                {
                    return new TransformAction(InterpolationUtils.InterpolateSize(this.StartValue.Scale, this.EndValue.Scale, position), position < 0.5 ? this.StartValue.Tag : this.EndValue.Tag);
                }
                else if (this.StartValue.Angle != null && this.EndValue.Angle != null)
                {
                    return new TransformAction(InterpolationUtils.InterpolateDouble(this.StartValue.Angle, this.EndValue.Angle, position), position < 0.5 ? this.StartValue.Tag : this.EndValue.Tag);
                }
                else
                {
                    double[,] startMatrix = this.StartValue.GetMatrix();
                    double[,] endMatrix = this.EndValue.GetMatrix();

                    double[,] tbrMatrix = new double[startMatrix.GetLength(0), startMatrix.GetLength(1)];

                    for (int i = 0; i < startMatrix.GetLength(0); i++)
                    {
                        for (int j = 0; j < startMatrix.GetLength(1); j++)
                        {
                            tbrMatrix[i, j] = startMatrix[i, j] * (1 - position) + endMatrix[i, j] * position;
                        }
                    }

                    return new TransformAction(tbrMatrix, position < 0.5 ? this.StartValue.Tag : this.EndValue.Tag);
                }
            }
        }
    }

    internal abstract class PrintableActionAnimation<T> : IAnimation<T> where T : IPrintableAction, IGraphicsAction
    {
        public override T StartValue { get; }
        public override T EndValue { get; }

        protected abstract T InterpolateConcrete((double, LineJoins, LineCaps, LineDash, Brush, Brush) commonElements, double position, Dictionary<string, IEasing> easings);

        public PrintableActionAnimation(T startValue, T endValue)
        {
            this.StartValue = startValue;
            this.EndValue = endValue;
        }
```

```

        public override T Interpolate(double position, Dictionary<string, IEasing> easings)
        {
            string tag = (StartValue as IPrintableAction)?.Tag ?? (StartValue as IGraphicsAction)?.Tag;

            if (easings != null && !string.IsNullOrEmpty(tag) && easings.TryGetValue(tag, out IEasing easing))
            {
                position = easing.Ease(position);
            }

            if (position <= 0)
            {
                return StartValue;
            }
            else if (position >= 1)
            {
                return EndValue;
            }
            else
            {
                T tbr = InterpolateConcrete(InterpolateCommon(position), position, easings);
                ((IGraphicsAction)tbr).Tag = ((IGraphicsAction)this.StartValue).Tag;
                return tbr;
            }
        }

        protected (double, LineJoins, LineCaps, LineDash, Brush, Brush) InterpolateCommon(double position)
        {
            double lineWidth = InterpolationUtils.InterpolateDouble(this.StartValue.LineWidth, this.EndValue.LineWidth, position);
            LineJoins lineJoin = position < 0.5 ? this.StartValue.LineJoin : this.EndValue.LineJoin;
            LineCaps lineCap = position < 0.5 ? this.StartValue.LineCap : this.EndValue.LineCap;
            LineDash lineDash = new LineDash(InterpolationUtils.InterpolateDouble(this.StartValue.LineDash.UnitsOn, this.EndValue.LineDash.UnitsOn, position),
                InterpolationUtils.InterpolateDouble(this.StartValue.LineDash.UnitsOff, this.EndValue.LineDash.UnitsOff, position),
                InterpolationUtils.InterpolateDouble(this.StartValue.LineDash.Phase, this.EndValue.LineDash.Phase, position));

            Brush stroke = null;

            if (this.StartValue.Stroke != null && this.EndValue.Stroke != null)
            {
                stroke = InterpolationUtils.InterpolateBrush(this.StartValue.Stroke, this.EndValue.Stroke, position);
            }

            Brush fill = null;

            if (this.StartValue.Fill != null && this.EndValue.Fill != null)
            {
                fill = InterpolationUtils.InterpolateBrush(this.StartValue.Fill, this.EndValue.Fill, position);
            }

            return (lineWidth, lineJoin, lineCap, lineDash, stroke, fill);
        }
    }

    internal class RectangleActionAnimation : PrintableActionAnimation<RectangleAction>
    {
        public RectangleActionAnimation(RectangleAction startValue, RectangleAction endValue) : base(startValue, endValue) { }

        protected override RectangleAction InterpolateConcrete((double, LineJoins, LineCaps, LineDash, Brush, Brush) commonElements, double position, Dictionary<string, IEasing> easings)
        {
            (double lineWidth, LineJoins lineJoin, LineCaps lineCap, LineDash lineDash, Brush stroke, Brush fill) = commonElements;

            return new RectangleAction(InterpolationUtils.InterpolatePoint(this.StartValue.TopLeft, this.EndValue.TopLeft, position),
                                       InterpolationUtils.InterpolateSize(this.StartValue.Size, this.EndValue.Size, position),
                                       fill, stroke, lineWidth, lineCap, lineJoin, lineDash, position < 0.5 ? this.StartValue.Tag : this.EndValue.Tag);
        }
    }

    internal class RasterImageActionAnimation : PrintableActionAnimation<RasterImageAction>
    {
        public RasterImageActionAnimation(RasterImageAction startValue, RasterImageAction endValue) : base(startValue, endValue) { }
```

```

        protected override RasterImageAction InterpolateConcrete((double, LineJoins, LineCaps, LineDash, Brush, Brush) commonElements, double position, Dictionary<string, IEasing> easings)
        {
            return new RasterImageAction(InterpolationUtils.InterpolateInt(this.StartValue.SourceX, this.EndValue.SourceX, position),
                InterpolationUtils.InterpolateInt(this.StartValue.SourceY, this.EndValue.SourceY, position),
                InterpolationUtils.InterpolateInt(this.StartValue.SourceWidth, this.EndValue.SourceWidth, position),
                InterpolationUtils.InterpolateInt(this.StartValue.SourceHeight, this.EndValue.SourceHeight, position),
                InterpolationUtils.InterpolateDouble(this.StartValue.DestinationX, this.EndValue.DestinationX, position),
                InterpolationUtils.InterpolateDouble(this.StartValue.DestinationY, this.EndValue.DestinationY, position),
                InterpolationUtils.InterpolateDouble(this.StartValue.DestinationWidth, this.EndValue.DestinationWidth, position),
                InterpolationUtils.InterpolateDouble(this.StartValue.DestinationHeight, this.EndValue.DestinationHeight, position), position < 0.5 ? this.StartValue.Image : this.EndValue.Image
                , position < 0.5 ? this.StartValue.Tag : this.EndValue.Tag);
        }
    }

    internal class TextActionAnimation : PrintableActionAnimation<TextAction>
    {
        public TextActionAnimation(TextAction startValue, TextAction endValue) : base(startValue, endValue) { }

        protected override TextAction InterpolateConcrete((double, LineJoins, LineCaps, LineDash, Brush, Brush) commonElements, double position, Dictionary<string, IEasing> easings)
        {
            (double lineWidth, LineJoins lineJoin, LineCaps lineCap, LineDash lineDash, Brush stroke, Brush fill) = commonElements;

            return new TextAction(InterpolationUtils.InterpolatePoint(this.StartValue.Origin, this.EndValue.Origin, position),
                position < 0.5 ? this.StartValue.Text : this.EndValue.Text,
                InterpolationUtils.InterpolateFont(this.StartValue.Font, this.EndValue.Font, position),
                position < 0.5 ? this.StartValue.TextBaseline : this.EndValue.TextBaseline,
                fill, stroke, lineWidth, lineCap, lineJoin, lineDash, position < 0.5 ? this.StartValue.Tag : this.EndValue.Tag);
        }
    }

    internal class PathActionAnimation : PrintableActionAnimation<PathAction>
    {
        public override PathAction StartValue { get; }
        public override PathAction EndValue { get; }

        private bool AreTheyTheSame(GraphicsPath pth1, GraphicsPath pth2)
        {
            if (pth1.Segments.Count != pth2.Segments.Count)
            {
                return false;
            }

            for (int i = 0; i < pth1.Segments.Count; i++)
            {
                if (pth1.Segments[i].Type != pth2.Segments[i].Type)
                {
                    return false;
                }
            }

            return true;
        }

        private Point GetCenter(GraphicsPath path)
        {
            List<Point> points = path.GetPoints().ToList().Aggregate(new List<Point>(), (a, b) => { a.AddRange(b); return a; });

            double cX = 0;
            double cY = 0;

            for (int i = 0; i < points.Count; i++)
            {
                cX += points[i].X;
                cY += points[i].Y;
            }

            return new Point(cX / points.Count, cY / points.Count);
        }

```

```
        private double ComparePaths(GraphicsPath path1, GraphicsPath path2)
        {
            Point c1 = GetCenter(path1);
            Point c2 = GetCenter(path2);

            return (c1.X - c2.X) * (c1.X - c2.X) + (c1.Y - c2.Y) * (c1.Y - c2.Y);
        }

        private List<(int, int)> GetBestAssignment(List<GraphicsPath> figures1, List<GraphicsPath>
figures2)
        {
            List<(int, int)> bestAssignment = new List<(int, int)>();

            double[,] distMat = new double[figures1.Count, figures2.Count];

            for (int i = 0; i < figures1.Count; i++)
            {
                for (int j = 0; j < figures2.Count; j++)
                {
                    distMat[i, j] = ComparePaths(figures1[i], figures2[j]);
                }
            }

            List<int> missingIndices1 = Enumerable.Range(0, figures1.Count).ToList();
            List<int> missingIndices2 = Enumerable.Range(0, figures2.Count).ToList();

            if (figures1.Count > figures2.Count)
            {
                while (missingIndices2.Count > 0)
                {
                    double minValue = double.MaxValue;
                    (int, int) bestPair = (-1, -1);

                    for (int i = 0; i < missingIndices1.Count; i++)
                    {
                        for (int j = 0; j < missingIndices2.Count; j++)
                        {
                            if (distMat[missingIndices1[i], missingIndices2[j]] < minValue)
                            {
                                minValue = distMat[missingIndices1[i], missingIndices2[j]];
                                bestPair = (missingIndices1[i], missingIndices2[j]);
                            }
                        }
                    }

                    bestAssignment.Add(bestPair);
                    missingIndices1.Remove(bestPair.Item1);
                    missingIndices2.Remove(bestPair.Item2);
                }
            }
            else
            {
                while (missingIndices1.Count > 0)
                {
                    double minValue = double.MaxValue;
                    (int, int) bestPair = (-1, -1);

                    for (int i = 0; i < missingIndices1.Count; i++)
                    {
                        for (int j = 0; j < missingIndices2.Count; j++)
                        {
                            if (distMat[missingIndices1[i], missingIndices2[j]] < minValue)
                            {
                                minValue = distMat[missingIndices1[i], missingIndices2[j]];
                                bestPair = (missingIndices1[i], missingIndices2[j]);
                            }
                        }
                    }

                    bestAssignment.Add(bestPair);
                    missingIndices1.Remove(bestPair.Item1);
                    missingIndices2.Remove(bestPair.Item2);
                }
            }

            return bestAssignment;
        }

        private void IncreasePoints(List<Point> points, bool isClosed)
        {
            if (points.Count > 1)
            {
                double maxLength = double.MinValue;
                int maxIndex = -1;

                double target = isClosed ? points.Count : points.Count - 1;

```

```
                for (int i = 0; i < target; i++)
                {
                    double length = (points[i].X - points[(i + 1) % points.Count].X) * (points[i].X - points[(i + 1) % points.Count].X) + (points[i].Y - points[(i + 1) % points.Count].Y) * (points[i].Y - points[(i + 1) % points.Count].Y);

                    if (length > maxLength)
                    {
                        maxLength = length;
                        maxIndex = i;
                    }
                }

                points.Insert(maxIndex + 1, new Point((points[maxIndex].X + points[(maxIndex + 1) % points.Count].X) * 0.5, (points[maxIndex].Y + points[(maxIndex + 1) % points.Count].Y) * 0.5));
            }
            else
            {
                points.Add(points[0]);
            }

        }

        public PathActionAnimation(PathAction startValue, PathAction endValue, double linearisationResolution) : base(startValue, endValue)
        {
            GraphicsPath startPath = startValue.Path.ConvertArcsToBeziers();
            GraphicsPath endPath = endValue.Path.ConvertArcsToBeziers();

            if (AreTheyTheSame(startPath, endPath))
            {
                this.StartValue = new PathAction(startPath, startValue.Fill, startValue.Stroke, startValue.LineWidth, startValue.LineCap, startValue.LineJoin, startValue.LineDash, startValue.Tag, startValue.IsClipping);
                this.EndValue = new PathAction(endPath, endValue.Fill, endValue.Stroke, endValue.LineWidth, endValue.LineCap, endValue.LineJoin, endValue.LineDash, endValue.Tag, endValue.IsClipping);
            }
            else
            {
                List<GraphicsPath> startFigures = startPath.GetFigures().ToList();
                List<GraphicsPath> endFigures = endPath.GetFigures().ToList();

                List<(int, int)> correspondences = GetBestAssignment(startFigures, endFigures);

                HashSet<int> startFiguresIndices = new HashSet<int>(Enumerable.Range(0, startFigures.Count));
                HashSet<int> endFiguresIndices = new HashSet<int>(Enumerable.Range(0, endFigures.Count));

                for (int i = 0; i < correspondences.Count; i++)
                {
                    startFiguresIndices.Remove(correspondences[i].Item1);
                    endFiguresIndices.Remove(correspondences[i].Item2);
                }

                foreach (int i in startFiguresIndices)
                {
                    Point center = GetCenter(startFigures[i]);

                    GraphicsPath path = new GraphicsPath().MoveTo(center).LineTo(center);
                    if (startFigures[i].Segments[startFigures[i].Segments.Count - 1].Type == SegmentType.Close)
                    {
                        path.Close();
                    }

                    endFigures.Add(path);
                    correspondences.Add((i, endFigures.Count - 1));
                }


                foreach (int i in endFiguresIndices)
                {
                    Point center = GetCenter(endFigures[i]);

                    GraphicsPath path = new GraphicsPath().MoveTo(center).LineTo(center);
                    if (endFigures[i].Segments[endFigures[i].Segments.Count - 1].Type == SegmentType.Close)
                    {
                        path.Close();
                    }

                    startFigures.Add(path);
                    correspondences.Add((startFigures.Count - 1, i));
                }

```

```
                startPath = new GraphicsPath();
                endPath = new GraphicsPath();

                for (int i = 0; i < startFigures.Count; i++)
                {
                    double startLength = startFigures[correspondences[i].Item1].MeasureLength();
                    double endLength = endFigures[correspondences[i].Item2].MeasureLength();

                    double maxLength = Math.Max(startLength, endLength);
                    int numPoints = (int)Math.Ceiling(maxLength / linearisationResolution);

                    double startResolution = startLength / maxLength * linearisationResolution;
                    double endResolution = endLength / maxLength * linearisationResolution;

                    if (startResolution <= 0)
                    {
                        startResolution = 1;
                    }

                    if (endResolution <= 0)
                    {
                        endResolution = 1;
                    }

                    List<Point> startFigure = startFigures[correspondences[i].Item1].Linearise(startResolution).GetPoints().First();
                    bool startIsClosed = startFigures[correspondences[i].Item1].Segments[startFigures[correspondences[i].Item1].Segments.Count - 1].Type == SegmentType.Close;

                    List<Point> endFigure = endFigures[correspondences[i].Item2].Linearise(endResolution).GetPoints().First();
                    bool endIsClosed = endFigures[correspondences[i].Item2].Segments[endFigures[correspondences[i].Item2].Segments.Count - 1].Type == SegmentType.Close;

                    while (startFigure.Count < numPoints)
                    {
                        IncreasePoints(startFigure, startIsClosed);
                    }

                    while (endFigure.Count < numPoints)
                    {
                        IncreasePoints(endFigure, endIsClosed);
                    }


                    double minVariance = double.MaxValue;
                    int bestShift = -1;

                    for (int j = 0; j < numPoints; j++)
                    {
                        double average = 0;
                        //double averageSq = 0;

                        for (int k = 0; k < numPoints; k++)
                        {
                            double dist = (startFigure[k].X - endFigure[(k + j) % numPoints].X) * (startFigure[k].X - endFigure[(k + j) % numPoints].X) + (startFigure[k].Y - endFigure[(k + j) % numPoints].Y) * (startFigure[k].Y - endFigure[(k + j) % numPoints].Y);
                            average += dist;
                            //averageSq += dist * dist;
                        }

                        //double variance = averageSq / numPoints - (average / numPoints) * (average / numPoints);

                        if (average < minVariance)
                        {
                            minVariance = average;
                            bestShift = j;
                        }
                    }

                    startPath.MoveTo(startFigure[0]);
                    endPath.MoveTo(endFigure[bestShift % numPoints]);

                    for (int j = 0; j < numPoints; j++)
                    {
                        startPath.LineTo(startFigure[j]);
                        endPath.LineTo(endFigure[(j + bestShift) % numPoints]);
                    }

                    if (startIsClosed && endIsClosed)
                    {
                        startPath.Close();
```

```
                        endPath.Close();
                    }
                    else if (startIsClosed && !endIsClosed)
                    {
                        startPath.LineTo(startFigure[0]);
                        endPath.LineTo(endFigure[endFigure.Count - 1]);
                    }
                    else if (!startIsClosed && endIsClosed)
                    {
                        startPath.LineTo(startFigure[startFigure.Count - 1]);
                        endPath.LineTo(endFigure[0]);
                    }
                }

                this.StartValue = new PathAction(startPath, startValue.Fill, startValue.Stroke, startValue.LineWidth, startValue.LineCap, startValue.LineJoin, startValue.LineDash, startValue.Tag, startValue.IsClipping);
                this.EndValue = new PathAction(endPath, endValue.Fill, endValue.Stroke, endValue.LineWidth, endValue.LineCap, endValue.LineJoin, endValue.LineDash, endValue.Tag, endValue.IsClipping);

            }
        }

        private static Segment InterpolateSegment(Segment start, Segment end, double position)
        {
            if (start.Type == end.Type)
            {
                if (start.Type == SegmentType.Arc)
                {
                    ArcSegment startA = start as ArcSegment;
                    ArcSegment endA = end as ArcSegment;

                    return new ArcSegment(InterpolationUtils.InterpolatePoint(startA.Points[0], endA.Points[0], position), InterpolationUtils.InterpolateDouble(startA.Radius, endA.Radius, position), InterpolationUtils.InterpolateDouble(startA.StartAngle, endA.StartAngle, position), InterpolationUtils.InterpolateDouble(startA.EndAngle, endA.EndAngle, position));
                }
                else if (start.Type == SegmentType.Move)
                {
                    return new MoveSegment(InterpolationUtils.InterpolatePoint(start.Point, end.Point, position));
                }
                else if (start.Type == SegmentType.Line)
                {
                    return new LineSegment(InterpolationUtils.InterpolatePoint(start.Point, end.Point, position));
                }
                else if (start.Type == SegmentType.Close)
                {
                    return new CloseSegment();
                }
                else if (start.Type == SegmentType.CubicBezier)
                {
                    CubicBezierSegment startC = start as CubicBezierSegment;
                    CubicBezierSegment endC = end as CubicBezierSegment;

                    return new CubicBezierSegment(InterpolationUtils.InterpolatePoint(startC.Points[0], endC.Points[0], position),
                        InterpolationUtils.InterpolatePoint(startC.Points[1], endC.Points[1], position),
                        InterpolationUtils.InterpolatePoint(startC.Points[2], endC.Points[2], position));
                }
                else
                {
                    throw new NotImplementedException();
                }
            }
            else
            {
                return position < 0.5 ? start : end;
            }
        }

        protected override PathAction InterpolateConcrete((double, LineJoins, LineCaps, LineDash, Brush, Brush) commonElements, double position, Dictionary<string, IEasing> easings)
        {
            (double lineWidth, LineJoins lineJoin, LineCaps lineCap, LineDash lineDash, Brush stroke, Brush fill) = commonElements;

            GraphicsPath path = new GraphicsPath();

            for (int i = 0; i < this.StartValue.Path.Segments.Count; i++)
            {
                path.Segments.Add(InterpolateSegment(this.StartValue.Path.Segments[i], this.EndValue.Path.Segments[i], position));
```

```
            }

            return new PathAction(path, fill, stroke, lineWidth, lineCap, lineJoin, lineDash, position < 0.5 ? this.StartValue.Tag : this.EndValue.Tag, position < 0.5 ? this.StartValue.IsClipping : this.EndValue.IsClipping);
        }
    }

    internal class FilteredGraphicsAnimation : PrintableActionAnimation<FilteredGraphicsAction>
    {
        private IGraphicsAnimation GraphicsAnimation { get; }
        private IGraphicsAnimation MaskAnimation { get; }

        public FilteredGraphicsAnimation(FilteredGraphicsAction startValue, FilteredGraphicsAction endValue, double linearisationResolution) : base(startValue, endValue)
        {
            if (startValue.Content == endValue.Content)
            {
                this.GraphicsAnimation = new ConstantGraphicsAnimation(startValue.Content);
            }
            else
            {
                this.GraphicsAnimation = new GraphicsAnimation(startValue.Content, endValue.Content, linearisationResolution);
            }

            if (startValue.Filter is MaskFilter startM && endValue.Filter is MaskFilter endM)
            {
                if (startM.Mask == endM.Mask)
                {
                    this.MaskAnimation = new ConstantGraphicsAnimation(startM.Mask);
                }
                else
                {
                    this.MaskAnimation = new GraphicsAnimation(startM.Mask, endM.Mask, linearisationResolution);
                }
            }
        }

        protected override FilteredGraphicsAction InterpolateConcrete((double, LineJoins, LineCaps, LineDash, Brush, Brush) commonElements, double position, Dictionary<string, IEasing> easings)
        {
            Graphics gpr = this.GraphicsAnimation.Interpolate(position, easings);

            if (this.StartValue.Filter.GetType() == this.EndValue.Filter.GetType())
            {
                if (this.StartValue.Filter is BoxBlurFilter startBBF && this.EndValue.Filter is BoxBlurFilter endBBF)
                {
                    BoxBlurFilter filter = new BoxBlurFilter(InterpolationUtils.InterpolateDouble(startBBF.BoxRadius, endBBF.BoxRadius, position));

                    return new FilteredGraphicsAction(gpr, filter);
                }
                else if (this.StartValue.Filter is ColourMatrixFilter startCMF && this.EndValue.Filter is ColourMatrixFilter endCMF)
                {
                    double[,] colourMatrix = new double[5, 5];

                    for (int i = 0; i < 5; i++)
                    {
                        for (int j = 0; j < 5; j++)
                        {
                            colourMatrix[i, j] = InterpolationUtils.InterpolateDouble(startCMF.ColourMatrix[i, j], endCMF.ColourMatrix[i, j], position);
                        }
                    }

                    ColourMatrixFilter filter = new ColourMatrixFilter(new ColourMatrix(colourMatrix));

                    return new FilteredGraphicsAction(gpr, filter);
                }
                else if (this.StartValue.Filter is ConvolutionFilter startConvo && this.EndValue.Filter is ConvolutionFilter endConvo)
                {
                    if (startConvo.Kernel.GetLength(0) == endConvo.Kernel.GetLength(0) && startConvo.Kernel.GetLength(1) == endConvo.Kernel.GetLength(1))
                    {
                        double[,] kernel = new double[startConvo.Kernel.GetLength(0), startConvo.Kernel.GetLength(1)];

                        for (int i = 0; i < kernel.GetLength(0); i++)
                        {
                            for (int j = 0; j < kernel.GetLength(1); j++)
```

```
                            {
                                kernel[i, j] = InterpolationUtils.InterpolateDouble(startConvo.Kernel[i, j], endConvo.Kernel[i, j], position);
                            }
                        }

                        ConvolutionFilter filter = new ConvolutionFilter(kernel, InterpolationUtils.InterpolateDouble(startConvo.Scale, endConvo.Scale, position), position < 0.5 ? startConvo.PreserveAlpha : endConvo.PreserveAlpha, InterpolationUtils.InterpolateDouble(startConvo.Normalisation, endConvo.Normalisation, position), InterpolationUtils.InterpolateDouble(startConvo.Bias, endConvo.Bias, position));
                        return new FilteredGraphicsAction(gpr, filter);
                    }
                    else
                    {
                        return position < 0.5 ? this.StartValue : this.EndValue;
                    }
                }
                else if (this.StartValue.Filter is GaussianBlurFilter startGauss && this.EndValue.Filter is GaussianBlurFilter endGauss)
                {
                    GaussianBlurFilter filter = new GaussianBlurFilter(InterpolationUtils.InterpolateDouble(startGauss.StandardDeviation, endGauss.StandardDeviation, position));
                    return new FilteredGraphicsAction(gpr, filter);
                }
                else if (this.StartValue.Filter is MaskFilter startMask && this.EndValue.Filter is MaskFilter endMask)
                {
                    Graphics mask = MaskAnimation.Interpolate(position, easings);
                    MaskFilter filter = new MaskFilter(mask);
                    return new FilteredGraphicsAction(gpr, filter);
                }
                else
                {
                    return position < 0.5 ? this.StartValue : this.EndValue;
                }
            }
            else
            {
                return position < 0.5 ? this.StartValue : this.EndValue;
            }
        }
    }

    internal interface IGraphicsAnimation
    {
        Graphics StartValue { get; }
        Graphics EndValue { get; }
        Graphics Interpolate(double position, Dictionary<string, IEasing> easings);
    }

    internal class ConstantGraphicsAnimation : IGraphicsAnimation
    {
        public Graphics StartValue { get; }
        public Graphics EndValue => StartValue;
        public Graphics Interpolate(double position, Dictionary<string, IEasing> easings) => this.StartValue;

        public ConstantGraphicsAnimation(Graphics startValue)
        {
            this.StartValue = startValue;
        }
    }

    internal class GraphicsAnimation : IGraphicsAnimation
    {
        public Graphics StartValue { get; }
        public Graphics EndValue { get; }

        private List<IAnimation> StartAnimations { get; }
        private List<IAnimation> EndAnimations { get; }

        public GraphicsAnimation(Graphics startValue, Graphics endValue, double linearisationResolution)
        {
            this.StartValue = startValue;
            this.EndValue = endValue;

            Dictionary<string, IGraphicsAction> startTaggedActions = new Dictionary<string, IGraphicsAction>();

            foreach (IGraphicsAction action in startValue.Actions)
            {
                if (!string.IsNullOrEmpty(action.Tag))
                {
                    startTaggedActions[action.Tag] = action;
```

```
                }
            }

            Dictionary<string, IGraphicsAction> endTaggedActions = new Dictionary<string, IGraphicsAction>();

            foreach (IGraphicsAction action in endValue.Actions)
            {
                if (!string.IsNullOrEmpty(action.Tag))
                {
                    endTaggedActions[action.Tag] = action;
                }
            }

            List<IAnimation> startAnimations = new List<IAnimation>();

            Dictionary<string, IAnimation> computedAnimations = new Dictionary<string, IAnimation>();

            for (int i = 0; i < this.StartValue.Actions.Count; i++)
            {
                IGraphicsAction startAction = this.StartValue.Actions[i];

                if (!string.IsNullOrEmpty(this.StartValue.Actions[i].Tag) && endTaggedActions.TryGetValue(this.StartValue.Actions[i].Tag, out IGraphicsAction endAction) && startAction.GetType() == endAction.GetType())
                {
                    if (startAction is TransformAction startT && endAction is TransformAction endT)
                    {
                        IAnimation animation = new TransformAnimation(startT, endT);

                        computedAnimations[this.StartValue.Actions[i].Tag] = animation;

                        startAnimations.Add(animation);
                    }
                    else if (startAction is StateAction startS && endAction is StateAction endS)
                    {
                        IAnimation animation = new StateAnimation(startS, endS);
                        computedAnimations[this.StartValue.Actions[i].Tag] = animation;
                        startAnimations.Add(animation);
                    }
                    else if (startAction is TextAction startX && endAction is TextAction endX)
                    {
                        IAnimation animation = new TextActionAnimation(startX, endX);
                        computedAnimations[this.StartValue.Actions[i].Tag] = animation;
                        startAnimations.Add(animation);
                    }
                    else if (startAction is RectangleAction startR && endAction is RectangleAction endR)
                    {
                        IAnimation animation = new RectangleActionAnimation(startR, endR);
                        computedAnimations[this.StartValue.Actions[i].Tag] = animation;
                        startAnimations.Add(animation);
                    }
                    else if (startAction is PathAction startP && endAction is PathAction endP)
                    {
                        IAnimation animation = new PathActionAnimation(startP, endP, linearisationResolution);
                        computedAnimations[this.StartValue.Actions[i].Tag] = animation;
                        startAnimations.Add(animation);
                    }
                    else if (startAction is RasterImageAction startI && endAction is RasterImageAction endI)
                    {
                        IAnimation animation = new RasterImageActionAnimation(startI, endI);
                        computedAnimations[this.StartValue.Actions[i].Tag] = animation;
                        startAnimations.Add(animation);
                    }
                    else if (startAction is FilteredGraphicsAction startF && endAction is FilteredGraphicsAction endF)
                    {
                        IAnimation animation = new FilteredGraphicsAnimation(startF, endF, linearisationResolution);
                        computedAnimations[this.StartValue.Actions[i].Tag] = animation;
                        startAnimations.Add(animation);
                    }
                    else
                    {
                        startAnimations.Add(new ConstantAnimation<IGraphicsAction>(startAction));
                    }
                }
                else
                {
                    startAnimations.Add(new ConstantAnimation<IGraphicsAction>(startAction));
                }
            }


```

```
            List<IAnimation> endAnimations = new List<IAnimation>();

            for (int i = 0; i < this.EndValue.Actions.Count; i++)
            {
                IGraphicsAction endAction = this.EndValue.Actions[i];

                if (!string.IsNullOrEmpty(this.EndValue.Actions[i].Tag) && startTaggedActions.TryGetValue(this.EndValue.Actions[i].Tag, out IGraphicsAction startAction) && startAction.GetType() == endAction.GetType())
                {
                    endAnimations.Add(computedAnimations[endAction.Tag]);
                }
                else
                {
                    endAnimations.Add(new ConstantAnimation<IGraphicsAction>(endAction));
                }
            }

            this.StartAnimations = startAnimations;
            this.EndAnimations = endAnimations;
        }

        public Graphics Interpolate(double position, Dictionary<string, IEasing> easings)
        {
            if (position <= 0)
            {
                return this.StartValue;
            }
            else if (position >= 1)
            {
                return this.EndValue;
            }
            else
            {
                Graphics gpr = new Graphics();

                if (position < 0.5)
                {
                    for (int i = 0; i < StartAnimations.Count; i++)
                    {
                        gpr.Actions.Add(StartAnimations[i].Interpolate(position, easings));
                    }
                }
                else
                {
                    for (int i = 0; i < EndAnimations.Count; i++)
                    {
                        gpr.Actions.Add(EndAnimations[i].Interpolate(position, easings));
                    }
                }

                return gpr;
            }
        }

        /*public static (GraphicsPath, GraphicsPath) GetAnimatedPath(GraphicsPath start, GraphicsPath end)
        {
            PathAction startAction = new PathAction(start, null, null, 1, LineCaps.Butt, LineJoins.Miter, new LineDash(), null, false);
            PathAction endAction = new PathAction(end, null, null, 1, LineCaps.Butt, LineJoins.Miter, new LineDash(), null, false);

            PathActionAnimation anim = new PathActionAnimation(startAction, endAction, 1);

            return (anim.StartValue.Path, anim.EndValue.Path);
        }*/
    }

    /// <summary>
    /// A key frame for an animation.
    /// </summary>
    public class Frame
    {
        /// <summary>
        /// The duration of the frame, in milliseconds.
        /// </summary>
        public double Duration { get; }

        /// <summary>
        /// The contents of the frame.
        /// </summary>
        public Graphics Graphics { get; }

        /// <summary>
        /// Creates a new <see cref="Frame"/> with the specified contents and duration.
        /// </summary>
```

```
        /// <param name="graphics">The contents of the frame.</param>
        /// <param name="duration">The duration of the frame, in milliseconds.</param>
        public Frame(Graphics graphics, double duration)
        {
            this.Graphics = graphics;
            this.Duration = duration;
        }
    }

    /// <summary>
    /// Describes a function used to transform the transition speed.
    /// </summary>
    public interface IEasing
    {
        /// <summary>
        /// Applies the easing to the specified transition offset.
        /// </summary>
        /// <param name="value">The transition offset (ranging from 0 to 1).</param>
        /// <returns>The eased transition offset value.</returns>
        double Ease(double value);
    }

    /// <summary>
    /// Describes an easing defined by a Cubic Bezier curve.
    /// </summary>
    public class SplineEasing : IEasing
    {
        /// <summary>
        /// The first control point of the curve.
        /// </summary>
        public Point ControlPoint1 { get; }

        /// <summary>
        /// The second control point of the curve.
        /// </summary>
        public Point ControlPoint2 { get; }
        private double[] SampledEasings { get; }

        /// <summary>
        /// Creates a new <see cref="SplineEasing"/> with the specified control points. The start point is always (0, 0) and the end point is always (1, 1).
        /// </summary>
        /// <param name="controlPoint1">The first control point of the curve. Both X and Y must be between 0 and 1, inclusive.</param>
        /// <param name="controlPoint2">The second control point of the curve. Both X and Y must be between 0 and 1, inclusive.</param>
        /// <exception cref="ArgumentException">This exception is thrown if any coordinate of the control points is &lt; 0 or &gt; 1.</exception>
        public SplineEasing(Point controlPoint1, Point controlPoint2)
        {
            if (controlPoint1.X < 0 || controlPoint1.Y < 0 || controlPoint2.X > 1 || controlPoint2.Y > 1)
            {
                throw new ArgumentException("The control point coordinates are out of range! All coordinates must be within 0 and 1 (inclusive).");
            }

            this.ControlPoint1 = controlPoint1;
            this.ControlPoint2 = controlPoint2;

            Point[] pts = new Point[51];

            for (int i = 0; i <= 50; i++)
            {
                double t = i / 50.0;

                pts[i] = new Point(3 * (1 - t) * (1 - t) * t * controlPoint1.X + 3 * (1 - t) * t * t * controlPoint2.X + t * t * t, 3 * (1 - t) * (1 - t) * t * controlPoint1.Y + 3 * (1 - t) * t * t * controlPoint2.Y + t * t * t);
            }

            double[] sampledEasings = new double[51];


            double currX = 0;
            double currY = pts[0].Y;

            int nextSample = 1;
            sampledEasings[0] = pts[0].Y;

            for (int i = 1; i < sampledEasings.Length; i++)
            {
                double targetX = nextSample / 50.0;

                double newX = pts[i].X;

                while (currX <= targetX && newX >= targetX)
```

```
                {
                    sampledEasings[nextSample] = currY + (targetX - currX) / (newX - currX) * (pts[i].Y - currY);
                    nextSample++;
                    targetX = nextSample / 50.0;
                }

                currX = newX;
                currY = pts[i].Y;
            }

            this.SampledEasings = sampledEasings;
        }

        /// <inheritdoc/>
        public double Ease(double value)
        {
            if (value <= 0)
            {
                return 0;
            }
            else if (value >= 1)
            {
                return 1;
            }
            else
            {
                int lowIndex = (int)Math.Floor(value * 50);
                int highIndex = (int)Math.Ceiling(value * 50);

                return SampledEasings[lowIndex] + (SampledEasings[highIndex] - SampledEasings[lowIndex]) * (value * 50 - lowIndex);
            }
        }
    }

    /// <summary>
    /// Describes a linear easing (i.e., no easing).
    /// </summary>
    public class LinearEasing : IEasing
    {
        /// <summary>
        /// Creates a new <see cref="LinearEasing"/>.
        /// </summary>
        public LinearEasing()
        {

        }

        /// <inheritdoc/>
        public double Ease(double value) => value;
    }

    /// <summary>
    /// Describes the transition between two successive <see cref="Frame"/>s.
    /// </summary>
    public class Transition
    {
        /// <summary>
        /// The duration of the transition, in milliseconds.
        /// </summary>
        public double Duration { get; }

        /// <summary>
        /// The <see cref="IEasing"/> to apply to all elements for which another easing is not specified. Set to null to use the default linear easing.
        /// </summary>
        public IEasing OverallEasing { get; } = null;

        /// <summary>
        /// A dictionary associating graphic action tags to the corresponding <see cref="IEasing"/>.
        /// </summary>
        public Dictionary<string, IEasing> Easings { get; } = null;

        /// <summary>
        /// Creates a new <see cref="Transition"/> with the specified duration and easings.
        /// </summary>
        /// <param name="duration">The duration of the transition, in milliseconds.</param>
        /// <param name="easing">The <see cref="IEasing"/> to apply to all elements for which another easing is not specified. Set to null to use the default linear easing.</param>
        /// <param name="easings">A dictionary associating graphic action tags to the corresponding <see cref="IEasing"/>.</param>
        public Transition(double duration, IEasing easing = null, Dictionary<string, IEasing> easings = null)
        {
            if (easing == null)
            {
```

```
                easing = new LinearEasing();
            }

            this.Duration = duration;
            this.OverallEasing = easing;
            this.Easings = easings;
        }
    }

    /// <summary>
    /// Describes an animation constituted by a number of frames and transitions between them.
    /// </summary>
    public class Animation
    {
        /// <summary>
        /// The key frames of the animation.
        /// </summary>
        public ImmutableList<Frame> Frames { get; private set; } = ImmutableList<Frame>.Empty;

        /// <summary>
        /// The transitions between successive frames of the animation. This array always contains one fewer element than <see cref="Frames"/>.
        /// </summary>
        public ImmutableList<Transition> Transitions { get; private set; } = ImmutableList<Transition>.Empty;

        private ImmutableList<GraphicsAnimation> Animations { get; set; } = ImmutableList<GraphicsAnimation>.Empty;


        /// <summary>
        /// The width of the animation.
        /// </summary>
        public double Width { get; set; }

        /// <summary>
        /// The height of the animation.
        /// </summary>
        public double Height { get; set; }

        /// <summary>
        /// The background colour of the animation.
        /// </summary>
        public Colour Background { get; set; } = Colour.FromRgba(255, 255, 255, 0);

        /// <summary>
        /// The absolute length between successive samples to use when linearising <see cref="GraphicsPath"/>s.
        /// </summary>
        public double LinearisationResolution { get; }

        /// <summary>
        /// The total duration of the animation (not including the number of repeats).
        /// </summary>
        public double Duration { get; private set; } = 0;

        /// <summary>
        /// The number of times that the animation should repeat.
        /// </summary>
        public int RepeatCount { get; set; } = 0;

        /// <summary>
        /// Creates a new <see cref="Animation"/> with the specified width, height and linearisation resolution.
        /// </summary>
        /// <param name="width">The width of the animation.</param>
        /// <param name="height">The height of the animation.</param>
        /// <param name="linearisationResolution">The absolute length between successive samples to use when linearising <see cref="GraphicsPath"/>s.</param>
        public Animation(double width, double height, double linearisationResolution)
        {
            this.Width = width;
            this.Height = height;
            this.LinearisationResolution = linearisationResolution;
        }

        /// <summary>
        /// Obtains the (interpolated) frame that should be displayed after the specified time has passed since the start of the animation.
        /// </summary>
        /// <param name="time">The time since the start of the animation (in milliseconds).</param>
        /// <returns>A <see cref="Page"/> containing the interpolated frame that should be displayed after the specified time has passed since the start of the animation.</returns>
        public Page GetFrameAtAbsolute(double time)
        {
            if (time > Duration)
            {
```

```
                if (this.RepeatCount <= 0)
                {
                    time = time % this.Duration;
                }
                else
                {
                    time = Math.Max(time % this.Duration, time - this.Duration * RepeatCount);
                }
            }

            if (time <= 0)
            {
                Page pag = new Page(this.Width, this.Height) { Background = this.Background };
                pag.Graphics.DrawGraphics(0, 0, this.Frames[0].Graphics);
                return pag;
            }
            else if (time > this.Duration)
            {
                Page pag = new Page(this.Width, this.Height) { Background = this.Background };
                pag.Graphics.DrawGraphics(0, 0, this.Frames[this.Frames.Count - 1].Graphics);
                return pag;
            }
            else
            {
                double currTime = 0;

                for (int i = 0; i < Frames.Count; i++)
                {
                    if (i > 0)
                    {
                        double newTime = currTime + Transitions[i - 1].Duration;

                        if (currTime <= time && newTime >= time)
                        {
                            Frame previousFrame = Frames[i - 1];

                            Frame nextFrame = Frames[i];

                            if (previousFrame != null && nextFrame != null)
                            {
                                double position = Transitions[i - 1].OverallEasing.Ease((time - currTime) / (newTime - currTime));

                                Graphics gpr = Animations[i - 1].Interpolate(position, Transitions[i - 1].Easings);
                                Page pag = new Page(this.Width, this.Height) { Background = this.Background };
                                pag.Graphics.DrawGraphics(0, 0, gpr);
                                return pag;
                            }
                            else if (previousFrame == null && nextFrame != null)
                            {
                                Page pag = new Page(this.Width, this.Height) { Background = this.Background };
                                pag.Graphics.DrawGraphics(0, 0, nextFrame.Graphics);
                                return pag;
                            }
                            else if (previousFrame != null && nextFrame == null)
                            {
                                Page pag = new Page(this.Width, this.Height) { Background = this.Background };
                                pag.Graphics.DrawGraphics(0, 0, previousFrame.Graphics);
                                return pag;
                            }
                            else
                            {
                                Page pag = new Page(this.Width, this.Height) { Background = this.Background };
                                return pag;
                            }
                        }

                        currTime = newTime;
                    }


                    {
                        double newTime = currTime + Frames[i].Duration;

                        if (currTime <= time && newTime >= time)
                        {
                            Page pag = new Page(this.Width, this.Height) { Background = this.Background };
                            pag.Graphics.DrawGraphics(0, 0, Frames[i].Graphics);
                            return pag;
                        }

```

```
                        currTime = newTime;
                    }

                }


                {
                    Page pag = new Page(this.Width, this.Height) { Background = this.Background };
                    pag.Graphics.DrawGraphics(0, 0, this.Frames[this.Frames.Count - 1].Graphics);
                    return pag;
                }
            }
        }

        /// <summary>
        /// Obtains the (interpolated) frame that should be displayed after the specified relative time has passed since the start of the animation.
        /// </summary>
        /// <param name="relativeTime">The time since the start of the animation (ranging from 0 for the start of the animation, to 1 for the end of the animation).</param>
        /// <returns>A <see cref="Page"/> containing the interpolated frame that should be displayed after the specified time has passed since the start of the animation.</returns>
        public Page GetFrameAtRelative(double relativeTime)
        {
            return GetFrameAtAbsolute(this.Duration * relativeTime);
        }

        /// <summary>
        /// Adds a new frame to the animation, with the specified transition.
        /// </summary>
        /// <param name="frame">The new frame to add to the animation.</param>
        /// <param name="transition">The transition that should be applied between the previous frame and the new frame. This parameter is ignored for the first frame. If this is <see langword="null"/>, the animation will abruptly change from one frame to the next.</param>
        public void AddFrame(Frame frame, Transition transition = null)
        {
            this.Frames = this.Frames.Add(frame);
            this.Duration += frame.Duration;

            if (this.Frames.Count > 1)
            {
                if (transition == null)
                {
                    transition = new Transition(0, new LinearEasing());
                }

                this.Transitions = this.Transitions.Add(transition);
                this.Animations = this.Animations.Add(new GraphicsAnimation(this.Frames[this.Frames.Count - 2].Graphics, this.Frames[this.Frames.Count - 1].Graphics, this.LinearisationResolution));
                this.Duration += transition.Duration;
            }
        }

        /// <summary>
        /// Removes the last frame from the animation (and the corresponding transition).
        /// </summary>
        /// <exception cref="ArgumentOutOfRangeException">Thrown if there are no frames in the animation.</exception>
        public void RemoveLastFrame()
        {
            if (this.Frames.Count > 0)
            {
                if (this.Frames.Count > 1)
                {
                    double lostDuration = this.Frames[this.Frames.Count - 1].Duration + this.Transitions[this.Transitions.Count - 1].Duration;

                    this.Frames = this.Frames.RemoveAt(this.Frames.Count - 1);
                    this.Animations = this.Animations.RemoveAt(this.Animations.Count - 1);
                    this.Transitions = this.Transitions.RemoveAt(this.Transitions.Count - 1);

                    this.Duration -= lostDuration;
                }
                else
                {
                    this.Frames = ImmutableList<Frame>.Empty;
                    this.Duration = 0;
                }
            }
            else
            {
                throw new ArgumentOutOfRangeException("There are no frames in the animation!");
            }
        }
    }
}
```

8.30 Brush.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.Immutable;
using System.Linq;
using System.Text;

namespace VectSharp
{
    /// <summary>
    /// Represents a brush used to fill or stroke graphics elements. This could be a solid colour, or a more complicated gradient or pattern.
    /// </summary>
    public abstract class Brush
    {
        internal Brush() { }

        /// <summary>
        /// Returns a brush corresponding the current instance, with the specified <paramref name="opacity"/> multiplication applied.
        /// </summary>
        /// <param name="opacity">The value that will be used to multiply the opacity of the brush.</param>
        /// <returns>A brush corresponding the current instance, with the specified <paramref name="opacity"/> multiplication applied.</returns>
        public abstract Brush MultiplyOpacity(double opacity);

        /// <summary>
        /// Implicitly converts a <see cref="Colour"/> into a <see cref="SolidColourBrush"/>.
        /// </summary>
        /// <param name="colour">The <see cref="Colour"/> to use for the brush.</param>
        public static implicit operator Brush(Colour colour)
        {
            return new SolidColourBrush(colour);
        }
    }

    /// <summary>
    /// Represents a brush painting with a single solid colour.
    /// </summary>
    public class SolidColourBrush : Brush
    {
        /// <summary>
        /// The colour of the brush.
        /// </summary>
        public Colour Colour { get; }

        /// <summary>
        /// Red component of the colour. Range: [0, 1].
        /// </summary>
        public double R => Colour.R;

        /// <summary>
        /// Green component of the colour. Range: [0, 1].
        /// </summary>
        public double G => Colour.G;

        /// <summary>
        /// Blue component of the colour. Range: [0, 1].
        /// </summary>
        public double B => Colour.B;

        /// <summary>
        /// Alpha component of the colour. Range: [0, 1].
        /// </summary>
        public double A => Colour.A;

        /// <summary>
```

```
        /// Creates a new <see cref="SolidColourBrush"/> with the specified <paramref name="colour"/>.
        /// </summary>
        /// <param name="colour">The <see cref="Colour"/> to use for the brush.</param>
        public SolidColourBrush(Colour colour)
        {
            this.Colour = colour;
        }

        /// <inheritdoc/>
        public override Brush MultiplyOpacity(double opacity)
        {
            return new SolidColourBrush(this.Colour.WithAlpha(this.Colour.A * opacity));
        }

        /// <summary>
        /// Implicitly converts a <see cref="Colour"/> into a <see cref="SolidColourBrush"/>.
        /// </summary>
        /// <param name="colour">The <see cref="Colour"/> to use for the brush.</param>
        public static implicit operator SolidColourBrush(Colour colour)
        {
            return new SolidColourBrush(colour);
        }
    }

    /// <summary>
    /// Represents a colour stop in a gradient.
    /// </summary>
    public struct GradientStop
    {
        /// <summary>
        /// The <see cref="Colour"/> at the gradient stop.
        /// </summary>
        public Colour Colour { get; }

        /// <summary>
        /// The offset of the gradient stop. Range: [0, 1].
        /// </summary>
        public double Offset { get; }

        /// <summary>
        /// Creates a new <see cref="GradientStop"/> instance.
        /// </summary>
        /// <param name="colour">The <see cref="Colour"/> at the gradient stop.</param>
        /// <param name="offset">The offset of the gradient stop. Range: [0, 1].</param>
        public GradientStop(Colour colour, double offset)
        {
            this.Colour = colour;
            this.Offset = Math.Max(0, Math.Min(1, offset));
        }

        /// <summary>
        /// Returns a <see cref="GradientStop"/> corresponding to the current instance, whose colour's opacity has been multiplied by the specified value.
        /// </summary>
        /// <param name="opacity">The value that will be used to multiply the colour's opacity.</param>
        /// <returns>A <see cref="GradientStop"/> corresponding to the current instance, whose colour's opacity has been multiplied by the specified value.</returns>
        public GradientStop MultiplyOpacity(double opacity)
        {
            return new GradientStop(this.Colour.WithAlpha(this.Colour.A * opacity), this.Offset);
        }
    }

    /// <summary>
    /// Represents a read-only list of <see cref="GradientStop"/>s.
    /// </summary>
    public class GradientStops : IReadOnlyList<GradientStop>
    {
        /// <summary>
        /// The minimum distance that is enforced between consecutive gradient stops.
        /// </summary>
        public static readonly double StopTolerance = 1e-7;

        /// <inheritdoc/>
        public GradientStop this[int index] => gradientStops[index];

        /// <inheritdoc/>
        public int Count => gradientStops.Count;

        private ImmutableList<GradientStop> gradientStops { get; set; }

        /// <inheritdoc/>
        public IEnumerator<GradientStop> GetEnumerator()
        {
            return ((IEnumerable<GradientStop>)gradientStops).GetEnumerator();
        }
```

```
        IEnumerator IEnumerable.GetEnumerator()
        {
            return ((IEnumerable)gradientStops).GetEnumerator();
        }

        /// <summary>
        /// Creates a new <see cref="GradientStops"/> instance containing the specified gradient stops.
        /// </summary>
        /// <param name="gradientStops">The gradient stops that will be contained in the <see cref="GradientStops"/> object.</param>
        public GradientStops(IEnumerable<GradientStop> gradientStops)
        {
            List<GradientStop> stops = (from el in gradientStops orderby el.Offset ascending select el).ToList();

            if (stops.Count == 0)
            {
                stops.Add(new GradientStop(Colour.FromRgba(0, 0, 0, 0), 0));
            }

            if (stops[0].Offset > 0)
            {
                stops.Insert(0, new GradientStop(stops[0].Colour, 0));
            }

            if (stops[stops.Count - 1].Offset < 1)
            {
                stops.Add(new GradientStop(stops[stops.Count - 1].Colour, 1));
            }

            for (int i = 1; i < stops.Count - 1; i++)
            {
                bool closeToPrevious = (stops[i].Offset - stops[i - 1].Offset < StopTolerance);
                bool closeToNext = (stops[i + 1].Offset - stops[i].Offset < StopTolerance);

                if (closeToPrevious && !closeToNext)
                {
                    stops[i] = new GradientStop(stops[i].Colour, stops[i - 1].Offset + StopTolerance);
                }
                else if (!closeToPrevious && closeToNext)
                {
                    stops[i] = new GradientStop(stops[i].Colour, stops[i + 1].Offset - StopTolerance);
                }
                else if (closeToPrevious && closeToNext)
                {
                    stops.RemoveAt(i);
                    i--;
                }
            }

            this.gradientStops = ImmutableList.Create(stops.ToArray());
        }

        /// <summary>
        /// Creates a new <see cref="GradientStops"/> instance containing the specified gradient stops.
        /// </summary>
        /// <param name="gradientStops">The gradient stops that will be contained in the <see cref="GradientStops"/> object.</param>
        public GradientStops(params GradientStop[] gradientStops) : this((IEnumerable<GradientStop>)gradientStops)
        {

        }

        /// <summary>
        /// Gets the colour at a certain position on the gradient.
        /// </summary>
        /// <param name="position">The position in the gradient (ranging from 0 to 1).</param>
        /// <returns>The colour of the gradient at the specified position.</returns>
        public Colour GetColourAt(double position)
        {
            if (position <= 0)
            {
                return this.gradientStops[0].Colour;
            }
            else if (position >= 1)
            {
                return this.gradientStops[this.gradientStops.Count - 1].Colour;
            }
            else
            {
                for (int i = 1; i < this.gradientStops.Count; i++)
                {
                    if (this.gradientStops[i - 1].Offset < position && this.gradientStops[i].Offset >=
```

```
position)
                    {
                        double factor = (position - this.gradientStops[i - 1].Offset) / (this.gradientStops[i].Offset - this.gradientStops[i - 1].Offset);

                        return Colour.FromRgba(this.gradientStops[i - 1].Colour.R * (1 - factor) + this.gradientStops[i].Colour.R * factor,
                            this.gradientStops[i - 1].Colour.G * (1 - factor) + this.gradientStops[i].Colour.G * factor,
                            this.gradientStops[i - 1].Colour.B * (1 - factor) + this.gradientStops[i].Colour.B * factor,
                            this.gradientStops[i - 1].Colour.A * (1 - factor) + this.gradientStops[i].Colour.A * factor);
                    }
                }
                return Colour.FromRgba(0, 0, 0, 0);
            }
        }
    }

    /// <summary>
    /// Represents a brush painting with a gradient.
    /// </summary>
    public abstract class GradientBrush : Brush
    {
        /// <summary>
        /// The colour stops in the gradient.
        /// </summary>
        public GradientStops GradientStops { get; protected internal set; }

        internal GradientBrush() { }
    }

    /// <summary>
    /// Represents a brush painting with a linear gradient.
    /// </summary>
    public class LinearGradientBrush : GradientBrush
    {
        /// <summary>
        /// The starting point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
        /// </summary>
        public Point StartPoint { get; }

        /// <summary>
        /// The end point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.
        /// </summary>
        public Point EndPoint { get; }

        /// <summary>
        /// Creates a new <see cref="LinearGradientBrush"/> with the specified start point, end point and gradient stops.
        /// </summary>
        /// <param name="startPoint">The starting point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.</param>
        /// <param name="endPoint">The ending point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.</param>
        /// <param name="gradientStops">The colour stops in the gradient.</param>
        public LinearGradientBrush(Point startPoint, Point endPoint, IEnumerable<GradientStop> gradientStops)
        {
            this.StartPoint = startPoint;
            this.EndPoint = endPoint;

            this.GradientStops = new GradientStops(gradientStops);
        }

        /// <summary>
        /// Creates a new <see cref="LinearGradientBrush"/> with the specified start point, end point and gradient stops.
        /// </summary>
        /// <param name="startPoint">The starting point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.</param>
        /// <param name="endPoint">The ending point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.</param>
        /// <param name="gradientStops">The colour stops in the gradient.</param>
        public LinearGradientBrush(Point startPoint, Point endPoint, params GradientStop[] gradientStops)
        {
            this.StartPoint = startPoint;
            this.EndPoint = endPoint;
            List<GradientStop> stops = (from el in gradientStops orderby el.Offset ascending select el).ToList();

            if (stops.Count == 0)
            {
```

```
                stops.Add(new GradientStop(Colour.FromRgba(0, 0, 0, 0), 0));
            }

            if (stops[0].Offset > 0)
            {
                stops.Insert(0, new GradientStop(stops[0].Colour, 0));
            }

            if (stops[stops.Count - 1].Offset < 1)
            {
                stops.Add(new GradientStop(stops[stops.Count - 1].Colour, 1));
            }

            this.GradientStops = new GradientStops(gradientStops);
        }

        /// <summary>
        /// Returns a <see cref="LinearGradientBrush"/> with the same gradient stops as the current instance, whose start and end point correspond to the points of the current instance in the
        /// original reference frame of the <paramref name="referenceGraphics"/>. This involves computing the current transform matrix of the <paramref name="referenceGraphics" />, inverting it,
        /// and applying the inverse matrix to the <see cref="StartPoint"/> and <see cref="EndPoint"/> of the current instance.
        /// </summary>
        /// <param name="referenceGraphics">The <see cref="Graphics"/> whose original reference frame is to be used.</param>
        /// <returns>A <see cref="LinearGradientBrush"/> with the same gradient stops as the current instance, whose start and end point correspond to the points of the current instance in the
        /// original reference frame of the <paramref name="referenceGraphics"/>.</returns>
        public LinearGradientBrush RelativeTo(Graphics referenceGraphics)
        {
            Stack<double[,]> transformMatrix = new Stack<double[,]>();
            double[,] currMatrix = new double[3, 3] { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };

            for (int i = 0; i < referenceGraphics.Actions.Count; i++)
            {
                if (referenceGraphics.Actions[i] is TransformAction)
                {
                    TransformAction trf = referenceGraphics.Actions[i] as TransformAction;

                    if (trf.Delta != null)
                    {
                        currMatrix = Graphics.Multiply(currMatrix, Graphics.TranslationMatrix(trf.Delta.Value.X, trf.Delta.Value.Y));
                    }
                    else if (trf.Angle != null)
                    {
                        currMatrix = Graphics.Multiply(currMatrix, Graphics.RotationMatrix(trf.Angle.Value));
                    }
                    else if (trf.Scale != null)
                    {
                        currMatrix = Graphics.Multiply(currMatrix, Graphics.ScaleMatrix(trf.Scale.Value.Width, trf.Scale.Value.Height));
                    }
                    else if (trf.Matrix != null)
                    {
                        currMatrix = Graphics.Multiply(currMatrix, trf.Matrix);
                    }
                }
                else if (referenceGraphics.Actions[i] is StateAction)
                {
                    if (((StateAction)referenceGraphics.Actions[i]).StateActionType == StateAction.StateActionTypes.Save)
                    {
                        transformMatrix.Push(currMatrix);
                    }
                    else
                    {
                        currMatrix = transformMatrix.Pop();
                    }
                }
            }

            currMatrix = Graphics.Invert(currMatrix);

            Point p1 = Graphics.Multiply(currMatrix, this.StartPoint);
            Point p2 = Graphics.Multiply(currMatrix, this.EndPoint);

            return new LinearGradientBrush(p1, p2, this.GradientStops);
        }

        /// <inheritdoc/>
        public override Brush MultiplyOpacity(double opacity)
        {
            return new LinearGradientBrush(this.StartPoint, this.EndPoint, from el in this.GradientStops select el.MultiplyOpacity(opacity));
```

```
        }
    }

    /// <summary>
    /// Represents a brush painting with a radial gradient.
    /// </summary>
    public class RadialGradientBrush : GradientBrush
    {
        /// <summary>
        /// The focal point of the gradient (i.e. the point within the circle where the gradient starts).
        /// </summary>
        public Point FocalPoint { get; }

        /// <summary>
        /// Represents the centre of the gradient.
        /// </summary>
        public Point Centre { get; }

        /// <summary>
        /// The radius of the gradient.
        /// </summary>
        public double Radius { get; }

        /// <summary>
        /// Creates a new <see cref="RadialGradientBrush"/> with the specified focal point, centre, radius and gradient stops.
        /// </summary>
        /// <param name="focalPoint">The focal point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.</param>
        /// <param name="centre">The centre of the gradient. Note that this is relative to the current coordinate system when the gradient is used.</param>
        /// <param name="radius">The radius of the gradient. Note that this is relative to the current coordinate system when the gradient is used.</param>
        /// <param name="gradientStops">The colour stops in the gradient.</param>
        public RadialGradientBrush(Point focalPoint, Point centre, double radius, params GradientStop[] gradientStops)
        {
            if (new Point(focalPoint.X - centre.X, focalPoint.Y - centre.Y).Modulus() > radius)
            {
                Point norm = new Point(focalPoint.X - centre.X, focalPoint.Y - centre.Y).Normalize();
                focalPoint = new Point(centre.X + norm.X * radius, centre.Y + norm.Y * radius);
            }

            this.FocalPoint = focalPoint;
            this.Centre = centre;
            this.Radius = radius;

            List<GradientStop> stops = (from el in gradientStops orderby el.Offset ascending select el).ToList();

            if (stops.Count == 0)
            {
                stops.Add(new GradientStop(Colour.FromRgba(0, 0, 0, 0), 0));
            }

            if (stops[0].Offset > 0)
            {
                stops.Insert(0, new GradientStop(stops[0].Colour, 0));
            }

            if (stops[stops.Count - 1].Offset < 1)
            {
                stops.Add(new GradientStop(stops[stops.Count - 1].Colour, 1));
            }

            this.GradientStops = new GradientStops(gradientStops);
        }

        /// <summary>
        /// Creates a new <see cref="RadialGradientBrush"/> with the specified focal point, centre, radius and gradient stops.
        /// </summary>
        /// <param name="focalPoint">The focal point of the gradient. Note that this is relative to the current coordinate system when the gradient is used.</param>
        /// <param name="centre">The centre of the gradient. Note that this is relative to the current coordinate system when the gradient is used.</param>
        /// <param name="radius">The radius of the gradient. Note that this is relative to the current coordinate system when the gradient is used.</param>
        /// <param name="gradientStops">The colour stops in the gradient.</param>
        public RadialGradientBrush(Point focalPoint, Point centre, double radius, IEnumerable<GradientStop> gradientStops)
        {
            if (new Point(focalPoint.X - centre.X, focalPoint.Y - centre.Y).Modulus() > radius)
            {
                Point norm = new Point(focalPoint.X - centre.X, focalPoint.Y - centre.Y).Normalize();
                focalPoint = new Point(centre.X + norm.X * radius, centre.Y + norm.Y * radius);
```

```
            }

            this.FocalPoint = focalPoint;
            this.Centre = centre;
            this.Radius = radius;

            List<GradientStop> stops = (from el in gradientStops orderby el.Offset ascending select el).ToList();

            if (stops.Count == 0)
            {
                stops.Add(new GradientStop(Colour.FromRgba(0, 0, 0, 0), 0));
            }

            if (stops[0].Offset > 0)
            {
                stops.Insert(0, new GradientStop(stops[0].Colour, 0));
            }

            if (stops[stops.Count - 1].Offset < 1)
            {
                stops.Add(new GradientStop(stops[stops.Count - 1].Colour, 1));
            }

            this.GradientStops = new GradientStops(gradientStops);
        }

        /// <inheritdoc/>
        public override Brush MultiplyOpacity(double opacity)
        {
            return new RadialGradientBrush(this.FocalPoint, this.Centre, this.Radius, from el in this.GradientStops select el.MultiplyOpacity(opacity));
        }
    }
}
```

8.31 Colour.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;

namespace VectSharp
{
    /// <summary>
    /// Represents an RGB colour.
    /// </summary>
    public partial struct Colour : IEquatable<Colour>
    {
        /// <summary>
        /// Red component of the colour. Range: [0, 1].
        /// </summary>
        public double R;

        /// <summary>
        /// Green component of the colour. Range: [0, 1].
        /// </summary>
        public double G;

        /// <summary>
        /// Blue component of the colour. Range: [0, 1].
        /// </summary>
        public double B;

        /// <summary>
        /// Alpha component of the colour. Range: [0, 1].
        /// </summary>
        public double A;
```

```
        private Colour(double r, double g, double b, double a)
        {
            R = r;
            G = g;
            B = b;
            A = a;
        }

        /// <summary>
        /// Create a new colour from RGB (red, green and blue) values.
        /// </summary>
        /// <param name="r">The red component of the colour. Range: [0, 1].</param>
        /// <param name="g">The green component of the colour. Range: [0, 1].</param>
        /// <param name="b">The blue component of the colour. Range: [0, 1].</param>
        /// <returns>A <see cref="Colour"/> struct with the specified components and an alpha component of 1.</returns>
        public static Colour FromRgb(double r, double g, double b)
        {
            return new Colour(r, g, b, 1);
        }

        /// <summary>
        /// Create a new colour from RGB (red, green and blue) values.
        /// </summary>
        /// <param name="r">The red component of the colour. Range: [0, 255].</param>
        /// <param name="g">The green component of the colour. Range: [0, 255].</param>
        /// <param name="b">The blue component of the colour. Range: [0, 255].</param>
        /// <returns>A <see cref="Colour"/> struct with the specified components and an alpha component of 1.</returns>
        public static Colour FromRgb(byte r, byte g, byte b)
        {
            return new Colour(r / 255.0, g / 255.0, b / 255.0, 1);
        }

        /// <summary>
        /// Create a new colour from RGB (red, green and blue) values.
        /// </summary>
        /// <param name="r">The red component of the colour. Range: [0, 255].</param>
        /// <param name="g">The green component of the colour. Range: [0, 255].</param>
        /// <param name="b">The blue component of the colour. Range: [0, 255].</param>
        /// <returns>A <see cref="Colour"/> struct with the specified components and an alpha component of 1.</returns>
        public static Colour FromRgb(int r, int g, int b)
        {
            return new Colour(r / 255.0, g / 255.0, b / 255.0, 1);
        }

        /// <summary>
        /// Create a new colour from RGBA (red, green, blue and alpha) values.
        /// </summary>
        /// <param name="r">The red component of the colour. Range: [0, 1].</param>
        /// <param name="g">The green component of the colour. Range: [0, 1].</param>
        /// <param name="b">The blue component of the colour. Range: [0, 1].</param>
        /// <param name="a">The alpha component of the colour. Range: [0, 1].</param>
        /// <returns>A <see cref="Colour"/> struct with the specified components.</returns>
        public static Colour FromRgba(double r, double g, double b, double a)
        {
            return new Colour(r, g, b, a);
        }

        /// <summary>
        /// Create a new colour from RGBA (red, green, blue and alpha) values.
        /// </summary>
        /// <param name="r">The red component of the colour. Range: [0, 255].</param>
        /// <param name="g">The green component of the colour. Range: [0, 255].</param>
        /// <param name="b">The blue component of the colour. Range: [0, 255].</param>
        /// <param name="a">The alpha component of the colour. Range: [0, 255].</param>
        /// <returns>A <see cref="Colour"/><see cref="Colour"/> struct with the specified components.</returns>
        public static Colour FromRgba(byte r, byte g, byte b, byte a)
        {
            return new Colour(r / 255.0, g / 255.0, b / 255.0, a / 255.0);
        }

        /// <summary>
        /// Create a new colour from RGBA (red, green, blue and alpha) values.
        /// </summary>
        /// <param name="r">The red component of the colour. Range: [0, 255].</param>
        /// <param name="g">The green component of the colour. Range: [0, 255].</param>
        /// <param name="b">The blue component of the colour. Range: [0, 255].</param>
        /// <param name="a">The alpha component of the colour. Range: [0, 1].</param>
        /// <returns>A <see cref="Colour"/> struct with the specified components.</returns>
        public static Colour FromRgba(byte r, byte g, byte b, double a)
        {
            return new Colour(r / 255.0, g / 255.0, b / 255.0, a);
        }
```

```
        /// <summary>
        /// Create a new colour from RGBA (red, green, blue and alpha) values.
        /// </summary>
        /// <param name="r">The red component of the colour. Range: [0, 255].</param>
        /// <param name="g">The green component of the colour. Range: [0, 255].</param>
        /// <param name="b">The blue component of the colour. Range: [0, 255].</param>
        /// <param name="a">The alpha component of the colour. Range: [0, 255].</param>
        /// <returns>A <see cref="Colour"/> struct with the specified components.</returns>
        public static Colour FromRgba(int r, int g, int b, int a)
        {
            return new Colour(r / 255.0, g / 255.0, b / 255.0, a / 255.0);
        }

        /// <summary>
        /// Create a new colour from RGBA (red, green, blue and alpha) values.
        /// </summary>
        /// <param name="r">The red component of the colour. Range: [0, 255].</param>
        /// <param name="g">The green component of the colour. Range: [0, 255].</param>
        /// <param name="b">The blue component of the colour. Range: [0, 255].</param>
        /// <param name="a">The alpha component of the colour. Range: [0, 1].</param>
        /// <returns>A <see cref="Colour"/> struct with the specified components.</returns>
        public static Colour FromRgba(int r, int g, int b, double a)
        {
            return new Colour(r / 255.0, g / 255.0, b / 255.0, a);
        }

        /// <summary>
        /// Create a new colour from RGBA (red, green, blue and alpha) values.
        /// </summary>
        /// <param name="colour">A <see cref="ValueTuple{Int32, Int32, Int32, Double}"/> containing component information for the colour. For r, g, and b, range: [0, 255]; for a, range: [0, 1].</param>
        /// <returns>A <see cref="Colour"/> struct with the specified components.</returns>
        public static Colour FromRgba((int r, int g, int b, double a) colour)
        {
            return new Colour(colour.r / 255.0, colour.g / 255.0, colour.b / 255.0, colour.a);
        }

        /// <inheritdoc/>
        public override bool Equals(object obj)
        {
            if (!(obj is Colour))
            {
                return false;
            }
            else
            {
                return this.Equals((Colour)obj);
            }
        }

        /// <inheritdoc/>
        public bool Equals(Colour col)
        {
            return col.R == this.R && col.G == this.G && col.B == this.B && col.A == this.A;
        }

        /// <inheritdoc/>
        public static bool operator ==(Colour col1, Colour col2)
        {
            return col1.R == col2.R && col1.G == col2.G && col1.B == col2.B && col1.A == col2.A;
        }

        /// <inheritdoc/>
        public static bool operator !=(Colour col1, Colour col2)
        {
            return col1.R != col2.R || col1.G != col2.G || col1.B != col2.B || col1.A != col2.A;
        }

        /// <inheritdoc/>
        public override int GetHashCode()
        {
            return (int)(this.R * 255 + this.G * 255 * 255 + this.B * 255 * 255 * 255 + this.A * 255 * 255 * 255 * 255);
        }

        /// <summary>
        /// Convert the <see cref="Colour"/> object into a hex string that is constituted by a "#" followed by two-digit hexadecimal representations of the red, green and blue components of the colour (in the range 0x00 - 0xFF).
        /// Optionally also includes opacity (alpha channel) data.
        /// </summary>
        /// <param name="includeAlpha">Whether two additional hex digits representing the colour's opacity (alpha channel) should be included in the string.</param>
        /// <returns>A hex colour string.</returns>
        public string ToCSSString(bool includeAlpha)
        {
            if (includeAlpha)
```

```
            {
                return "#" + ((int)Math.Round(this.R * 255)).ToString("X2") + ((int)Math.Round(this.G * 255)).ToString("X2") + ((int)Math.Round(this.B * 255)).ToString("X2") + ((int)Math.Round(this.A * 255)).ToString("X2");
            }
            else
            {
                return "#" + ((int)Math.Round(this.R * 255)).ToString("X2") + ((int)Math.Round(this.G * 255)).ToString("X2") + ((int)Math.Round(this.B * 255)).ToString("X2");
            }
        }

        /// <summary>
        /// Convert a CSS colour string into a <see cref="Colour"/> object.
        /// </summary>
        /// <param name="cssString">The CSS colour string. In addition to 148 standard colour names (case-insensitive), #RGB, #RGBA, #RRGGBB and #RRGGBBAA hex strings and rgb(r, g, b) and rgba(r, g, b, a) functional colour notations are supported.</param>
        /// <returns></returns>
        public static Colour? FromCSSString(string cssString)
        {
            if (cssString.StartsWith("#"))
            {
                cssString = cssString.Substring(1);

                try
                {
                    if (cssString.Length == 3)
                    {
                        byte r = byte.Parse(cssString.Substring(0, 1) + cssString.Substring(0, 1), System.Globalization.NumberStyles.HexNumber);
                        byte g = byte.Parse(cssString.Substring(1, 1) + cssString.Substring(1, 1), System.Globalization.NumberStyles.HexNumber);
                        byte b = byte.Parse(cssString.Substring(2, 1) + cssString.Substring(2, 1), System.Globalization.NumberStyles.HexNumber);

                        return Colour.FromRgb(r, g, b);
                    }
                    else if (cssString.Length == 4)
                    {
                        byte r = byte.Parse(cssString.Substring(0, 1) + cssString.Substring(0, 1), System.Globalization.NumberStyles.HexNumber);
                        byte g = byte.Parse(cssString.Substring(1, 1) + cssString.Substring(1, 1), System.Globalization.NumberStyles.HexNumber);
                        byte b = byte.Parse(cssString.Substring(2, 1) + cssString.Substring(2, 1), System.Globalization.NumberStyles.HexNumber);
                        byte a = byte.Parse(cssString.Substring(3, 1) + cssString.Substring(3, 1), System.Globalization.NumberStyles.HexNumber);

                        return Colour.FromRgba(r, g, b, a);
                    }
                    else if (cssString.Length == 6)
                    {
                        byte r = byte.Parse(cssString.Substring(0, 2), System.Globalization.NumberStyles.HexNumber);
                        byte g = byte.Parse(cssString.Substring(2, 2), System.Globalization.NumberStyles.HexNumber);
                        byte b = byte.Parse(cssString.Substring(4, 2), System.Globalization.NumberStyles.HexNumber);

                        return Colour.FromRgb(r, g, b);
                    }
                    else if (cssString.Length == 8)
                    {
                        byte r = byte.Parse(cssString.Substring(0, 2), System.Globalization.NumberStyles.HexNumber);
                        byte g = byte.Parse(cssString.Substring(2, 2), System.Globalization.NumberStyles.HexNumber);
                        byte b = byte.Parse(cssString.Substring(4, 2), System.Globalization.NumberStyles.HexNumber);
                        byte a = byte.Parse(cssString.Substring(6, 2), System.Globalization.NumberStyles.HexNumber);

                        return Colour.FromRgba(r, g, b, a);
                    }
                    else
                    {
                        return null;
                    }
                }
                catch
                {
                    return null;
                }
            }
            else if (cssString.StartsWith("rgb(") || cssString.StartsWith("rgba("))
            {
```

```
                try
                {
                    cssString = cssString.Substring(cssString.IndexOf("(") + 1).Replace(")", "").Replace(" ", "");
                    string[] splitCssString = cssString.Split(',');

                    double R = ParseColourValueOrPercentage(splitCssString[0]);
                    double G = ParseColourValueOrPercentage(splitCssString[1]);
                    double B = ParseColourValueOrPercentage(splitCssString[2]);

                    double A = 1;

                    if (splitCssString.Length == 4)
                    {
                        A = double.Parse(splitCssString[3], System.Globalization.CultureInfo.InvariantCulture);
                    }

                    return Colour.FromRgba(R, G, B, A);
                }
                catch
                {
                    return null;
                }
            }
            else
            {
                if (StandardColours.TryGetValue(cssString, out Colour tbr))
                {
                    return tbr;
                }
                else
                {
                    return null;
                }
            }
        }

        private static double ParseColourValueOrPercentage(string value)
        {
            if (int.TryParse(value, out int tbr))
            {
                return tbr / 255.0;
            }
            else if (value.Contains("%"))
            {
                return double.Parse(value.Replace("%", ""), System.Globalization.CultureInfo.InvariantCulture) / 100.0;
            }
            else
            {
                return double.Parse(value, System.Globalization.CultureInfo.InvariantCulture);
            }
        }

        /// <summary>
        /// Create a new <see cref="Colour"/> with the same RGB components as the <paramref name="original"/> <see cref="Colour"/>, but with the specified <paramref name="alpha"/>.
        /// </summary>
        /// <param name="original">The original <see cref="Colour"/> from which the RGB components will be taken.</param>
        /// <param name="alpha">The alpha component of the new <see cref="Colour"/>.</param>
        /// <returns>A <see cref="Colour"/> struct with the same RGB components as the <paramref name="original"/> <see cref="Colour"/> and the specified <paramref name="alpha"/>.</returns>
        public static Colour WithAlpha(Colour original, double alpha)
        {
            return Colour.FromRgba(original.R, original.G, original.B, alpha);
        }

        /// <summary>
        /// Create a new <see cref="Colour"/> with the same RGB components as the <paramref name="original"/> <see cref="Colour"/>, but with the specified <paramref name="alpha"/>.
        /// </summary>
        /// <param name="original">The original <see cref="Colour"/> from which the RGB components will be taken.</param>
        /// <param name="alpha">The alpha component of the new <see cref="Colour"/>.</param>
        /// <returns>A <see cref="Colour"/> struct with the same RGB components as the <paramref name="original"/> <see cref="Colour"/> and the specified <paramref name="alpha"/>.</returns>
        public static Colour WithAlpha(Colour original, byte alpha)
        {
            return Colour.FromRgba(original.R, original.G, original.B, (double)alpha / 255.0);
        }

        /// <summary>
        /// Create a new <see cref="Colour"/> with the same RGB components as the current <see cref="Colour"/>, but with the specified <paramref name="alpha"/>.
        /// </summary>
```

```
        /// <param name="alpha">The alpha component of the new <see cref="Colour"/>.</param>
        /// <returns>A <see cref="Colour"/> struct with the same RGB components as the current <see cref="Colour"/> and the specified <paramref name="alpha"/>.</returns>
        public Colour WithAlpha(double alpha)
        {
            return Colour.FromRgba(this.R, this.G, this.B, alpha);
        }

        /// <summary>
        /// Create a new <see cref="Colour"/> with the same RGB components as the current <see cref="Colour"/>, but with the specified <paramref name="alpha"/>.
        /// </summary>
        /// <param name="alpha">The alpha component of the new <see cref="Colour"/>.</param>
        /// <returns>A <see cref="Colour"/> struct with the same RGB components as the current <see cref="Colour"/> and the specified <paramref name="alpha"/>.</returns>
        public Colour WithAlpha(byte alpha)
        {
            return Colour.FromRgba(this.R, this.G, this.B, (double)alpha / 255.0);
        }

        /// <summary>
        /// Converts a <see cref="Colour"/> to the CIE XYZ colour space.
        /// </summary>
        /// <returns>A <see cref="ValueTuple"/> containing the X, Y and Z components of the <see cref="Colour"/>.</returns>
        public (double X, double Y, double Z) ToXYZ()
        {
            double r, g, b;

            if (R <= 0.04045)
            {
                r = 25 * R / 323;
            }
            else
            {
                r = Math.Pow((200 * R + 11) / 211, 2.4);
            }

            if (G <= 0.04045)
            {
                g = 25 * G / 323;
            }
            else
            {
                g = Math.Pow((200 * G + 11) / 211, 2.4);
            }

            if (B <= 0.04045)
            {
                b = 25 * B / 323;
            }
            else
            {
                b = Math.Pow((200 * B + 11) / 211, 2.4);
            }

            double x = 0.41239080 * r + 0.35758434 * g + 0.18048079 * b;
            double y = 0.21263901 * r + 0.71516868 * g + 0.07219232 * b;
            double z = 0.01933082 * r + 0.11919478 * g + 0.95053215 * b;

            return (x, y, z);
        }

        /// <summary>
        /// Creates a <see cref="Colour"/> from CIE XYZ coordinates.
        /// </summary>
        /// <param name="x">The X coordinate.</param>
        /// <param name="y">The Y coordinate.</param>
        /// <param name="z">The Z coordinate.</param>
        /// <returns>An sRGB <see cref="Colour"/> created from the specified components.</returns>
        public static Colour FromXYZ(double x, double y, double z)
        {
            double r = +3.24096994 * x - 1.53738318 * y - 0.49861076 * z;
            double g = -0.96924364 * x + 1.8759675 * y + 0.04155506 * z;
            double b = 0.05563008 * x - 0.20397696 * y + 1.05697151 * z;

            if (r <= 0.0031308)
            {
                r = 323 * r / 25;
            }
            else
            {
                r = (211 * Math.Pow(r, 1 / 2.4) - 11) / 200;
            }

            if (g <= 0.0031308)
            {
```

```
                g = 323 * g / 25;
            }
            else
            {
                g = (211 * Math.Pow(g, 1 / 2.4) - 11) / 200;
            }

            if (b <= 0.0031308)
            {
                b = 323 * b / 25;
            }
            else
            {
                b = (211 * Math.Pow(b, 1 / 2.4) - 11) / 200;
            }

            r = Math.Min(Math.Max(0, r), 1);
            g = Math.Min(Math.Max(0, g), 1);
            b = Math.Min(Math.Max(0, b), 1);

            return Colour.FromRgb(r, g, b);
        }

        /// <summary>
        /// Converts a <see cref="Colour"/> to the CIE Lab colour space (under Illuminant D65).
        /// </summary>
        /// <returns>A <see cref="ValueType"/> containing the L*, a* and b* components of the <see cref="Colour"/>.</returns>
        public (double L, double a, double b) ToLab()
        {
            double f(double t)
            {
                const double d = 6.0 / 29;

                if (t > d * d * d)
                {
                    return Math.Pow(t, 1.0 / 3);
                }
                else
                {
                    return t / (3 * d * d) + 4.0 / 29;
                }
            }

            const double xN = 0.950489;
            const double yN = 1;
            const double zN = 1.088840;

            (double x, double y, double z) = this.ToXYZ();

            double fY = f(y / yN);

            double l = 1.16 * fY - 0.16;
            double a = 5 * (f(x / xN) - fY);
            double b = 2 * (fY - f(z / zN));

            return (l, a, b);
        }

        /// <summary>
        /// Creates a <see cref="Colour"/> from CIE Lab coordinates (under Illuminant D65).
        /// </summary>
        /// <param name="L">The L* component.</param>
        /// <param name="a">The a* component.</param>
        /// <param name="b">The b* component.</param>
        /// <returns>An sRGB <see cref="Colour"/> created from the specified components.</returns>
        public static Colour FromLab(double L, double a, double b)
        {
            double f(double t)
            {
                const double d = 6.0 / 29;

                if (t > d)
                {
                    return t * t * t;
                }
                else
                {
                    return 3 * d * d * (t - 4.0 / 29);
                }
            }

            const double xN = 0.950489;
            const double yN = 1;
            const double zN = 1.088840;

            double x = xN * f((L + 0.16) / 1.16 + a / 5);
```

```
            double y = yN * f((L + 0.16) / 1.16);
            double z = zN * f((L + 0.16) / 1.16 - b / 2);

            return Colour.FromXYZ(x, y, z);
        }

        /// <summary>
        /// Converts a <see cref="Colour"/> to the HSL colour space.
        /// </summary>
        /// <returns>A <see cref="ValueType"/> containing the H, S and L components of the <see
    cref="Colour"/>. Each component has range [0, 1].</returns>
        public (double H, double S, double L) ToHSL()
        {
            double xMax = Math.Max(Math.Max(R, G), B);
            double xMin = Math.Min(Math.Min(R, G), B);

            double l = (xMax + xMin) * 0.5;

            double h;

            if (xMax == xMin)
            {
                h = 0;
            }
            else if (xMax == R)
            {
                h = (G - B) / (xMax - xMin) / 6;
            }
            else if (xMax == G)
            {
                h = (2 + (B - R) / (xMax - xMin)) / 6;
            }
            else
            {
                h = (4 + (R - G) / (xMax - xMin)) / 6;
            }

            double s;

            if (l == 0 || l == 1)
            {
                s = 0;
            }
            else
            {
                s = (xMax - l) / Math.Min(l, 1 - l);
            }

            return (h, s, l);
        }

        /// <summary>
        /// Creates a <see cref="Colour"/> from HSL coordinates.
        /// </summary>
        /// <param name="h">The H component. Should be in range [0, 1].</param>
        /// <param name="s">The S component. Should be in range [0, 1].</param>
        /// <param name="l">The L component. Should be in range [0, 1].</param>
        /// <returns>A <see cref="Colour"/> created from the specified components.</returns>
        public static Colour FromHSL(double h, double s, double l)
        {
            double c = (1 - Math.Abs(2 * l - 1)) * s;

            double hp = h * 6;

            double x = c * (1 - Math.Abs((hp % 2) - 1));

            double r1, g1, b1;

            if (hp <= 1)
            {
                r1 = c;
                g1 = x;
                b1 = 0;
            }
            else if (hp <= 2)
            {
                r1 = x;
                g1 = c;
                b1 = 0;
            }
            else if (hp <= 3)
            {
                r1 = 0;
                g1 = c;
                b1 = x;
            }
            else if (hp <= 4)
```

```
            {
                r1 = 0;
                g1 = x;
                b1 = c;
            }
            else if (hp <= 5)
            {
                r1 = x;
                g1 = 0;
                b1 = c;
            }
            else if (hp <= 6)
            {
                r1 = c;
                g1 = 0;
                b1 = x;
            }
            else
            {
                r1 = 0;
                g1 = 0;
                b1 = 0;
            }

            double m = l - c / 2;

            return Colour.FromRgb(r1 + m, g1 + m, b1 + m);
        }

    }
}
```

8.32 Document.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Text;

namespace VectSharp
{
    /// <summary>
    /// Represents a collection of pages.
    /// </summary>
    public class Document
    {
        /// <summary>
        /// The pages in the document.
        /// </summary>
        public List<Page> Pages = new List<Page>();


        /// <summary>
        /// Create a new document.
        /// </summary>
        public Document()
        {

        }
    }

    /// <summary>
    /// Represents a <see cref="Graphics"/> object with a width and height.
    /// </summary>
    public class Page
    {
        /// <summary>
```

```
        /// Width of the page.
        /// </summary>
        public double Width { get; set; }

        /// <summary>
        /// Height of the page.
        /// </summary>
        public double Height { get; set; }

        /// <summary>
        /// Graphics surface of the page.
        /// </summary>
        public Graphics Graphics { get; set; }

        /// <summary>
        /// Background colour of the page.
        /// </summary>
        public Colour Background { get; set; } = Colour.FromRgba(255, 255, 255, 0);

        /// <summary>
        /// Create a new page.
        /// </summary>
        /// <param name="width">The width of the page.</param>
        /// <param name="height">The height of the page.</param>
        public Page(double width, double height)
        {
            this.Width = width;
            this.Height = height;

            this.Graphics = new Graphics();
            this.Graphics.Translate(0, 0);
        }

        /// <summary>
        /// Translate and resize the <see cref="Page"/> so that it displays the specified <paramref name="region"/>.
        /// </summary>
        /// <param name="region">The area to include in the page.</param>
        /// <param name="removeClippedGraphics">If this is <see langword="true"/>, graphics actions that fall outside of the specified region are completely removed from the plot, otherwise they are just hidden.</param>
        /// <param name="tag">A tag to identify the transform.</param>
        public void Crop(Rectangle region, bool removeClippedGraphics = false, string tag = null)
        {
            this.Crop(region.Location, region.Size, removeClippedGraphics, tag);
        }

        /// <summary>
        /// Translate and resize the <see cref="Page"/> so that it displays the rectangle defined by <paramref name="topLeft"/> and <paramref name="size"/>.
        /// </summary>
        /// <param name="topLeft">The top left corner of the area to include in the page.</param>
        /// <param name="size">The size of the area to include in the page.</param>
        /// <param name="removeClippedGraphics">If this is <see langword="true"/>, graphics actions that fall outside of the specified region are completely removed from the plot, otherwise they are just hidden.</param>
        /// <param name="tag">A tag to identify the transform.</param>
        public void Crop(Point topLeft, Size size, bool removeClippedGraphics = false, string tag = null)
        {
            if (removeClippedGraphics)
            {
                this.Graphics.Crop(new Rectangle(topLeft, size));
            }

            if (this.Graphics.Actions[0] is TransformAction transf)
            {
                double[,] currMatrix = transf.GetMatrix();

                double[,] newMatrix = Graphics.Multiply(Graphics.TranslationMatrix(-topLeft.X, -topLeft.Y), currMatrix);

                this.Graphics.Actions[0] = new TransformAction(newMatrix, tag);
            }
            else
            {
                this.Graphics.Actions.Insert(0, new TransformAction(new Point(-topLeft.X, -topLeft.Y), tag));
            }

            this.Width = size.Width;
            this.Height = size.Height;
        }

        /// <summary>
        /// Translate and resize the <see cref="Page"/> so that it displays the rectangle corresponding to the bounding box of its contents.
```

```
        /// </summary>
        /// <param name="tag">A tag to identify the transform.</param>
        public void Crop(string tag = null)
        {
            Rectangle bounds = this.Graphics.GetBounds();

            if (this.Graphics.Actions[0] is TransformAction transf)
            {
                double[,] currMatrix = transf.GetMatrix();

                double[,] newMatrix = Graphics.Multiply(Graphics.TranslationMatrix(-bounds.Location.X, -bounds.Location.Y), currMatrix);

                this.Graphics.Actions[0] = new TransformAction(newMatrix, tag);
            }
            else
            {
                this.Graphics.Actions.Insert(0, new TransformAction(new Point(-bounds.Location.X, -bounds.Location.Y), tag));
            }

            this.Width = bounds.Size.Width;
            this.Height = bounds.Size.Height;
        }
    }
}
```

8.33 Enums.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

namespace VectSharp
{
    /// <summary>
    /// Represent text baselines.
    /// </summary>
    public enum TextBaselines
    {
        /// <summary>
        /// The current vertical coordinate determines where the top of the text string will be placed.
        /// </summary>
        Top,

        /// <summary>
        /// The current vertical coordinate determines where the bottom of the text string will be placed.
        /// </summary>
        Bottom,

        /// <summary>
        /// The current vertical coordinate determines where the middle of the text string will be placed.
        /// </summary>
        Middle,

        /// <summary>
        /// The current vertical coordinate determines where the baseline of the text string will be placed.
        /// </summary>
        Baseline
    }

    /// <summary>
    /// Represents text anchors.
    /// </summary>
    public enum TextAnchors
    {
```

```
        /// <summary>
        /// The current coordinate will determine the position of the left side of the text string.
        /// </summary>
        Left,

        /// <summary>
        /// The current coordinate will determine the position of the center of the text string.
        /// </summary>
        Center,

        /// <summary>
        /// The current coordinate will determine the position of the right side of the text string.
        /// </summary>
        Right
    }

    /// <summary>
    /// Represents line caps.
    /// </summary>
    public enum LineCaps
    {
        /// <summary>
        /// The ends of the line are squared off at the endpoints.
        /// </summary>
        Butt = 0,

        /// <summary>
        /// The ends of the lines are rounded.
        /// </summary>
        Round = 1,

        /// <summary>
        /// The ends of the lines are squared off by adding an half square box at each end.
        /// </summary>
        Square = 2
    }

    /// <summary>
    /// Represents line joining options.
    /// </summary>
    public enum LineJoins
    {
        /// <summary>
        /// Consecutive segments are joined by straight corners.
        /// </summary>
        Bevel = 2,

        /// <summary>
        /// Consecutive segments are joined by extending their outside edges until they meet.
        /// </summary>
        Miter = 0,

        /// <summary>
        /// Consecutive segments are joined by arc segments.
        /// </summary>
        Round = 1
    }

    /// <summary>
    /// Represents instructions on how to paint a dashed line.
    /// </summary>
    public struct LineDash
    {
        /// <summary>
        /// A solid (not dashed) line
        /// </summary>
        public static LineDash SolidLine = new LineDash(0, 0, 0);

        /// <summary>
        /// Length of the "on" (painted) segment.
        /// </summary>
        public double UnitsOn;

        /// <summary>
        /// Length of the "off" (not painted) segment.
        /// </summary>
        public double UnitsOff;

        /// <summary>
        /// Position in the dash pattern at which the line starts.
        /// </summary>
        public double Phase;

        /// <summary>
        /// Define a new line dash pattern.
        /// </summary>
        /// <param name="unitsOn">The length of the "on" (painted) segment.</param>
```

```
        /// <param name="unitsOff">The length of the "off" (not painted) segment.</param>
        /// <param name="phase">The position in the dash pattern at which the line starts.</param>
        public LineDash(double unitsOn, double unitsOff, double phase)
        {
            UnitsOn = unitsOn;
            UnitsOff = unitsOff;
            Phase = phase;
        }
    }

    /// <summary>
    /// Types of <see cref="Segment"/>.
    /// </summary>
    public enum SegmentType
    {
        /// <summary>
        /// The segment represents a move from the current point to a new point.
        /// </summary>
        Move,

        /// <summary>
        /// The segment represents a straight line from the current point to a new point.
        /// </summary>
        Line,

        /// <summary>
        /// The segment represents a cubic bezier curve from the current point to a new point.
        /// </summary>
        CubicBezier,

        /// <summary>
        /// The segment represents a circular arc from the current point to a new point.
        /// </summary>
        Arc,

        /// <summary>
        /// The segment represents the closing segment of a figure.
        /// </summary>
        Close
    }

    /// <summary>
    /// Represents ways to deal with unbalanced graphics state stacks.
    /// </summary>
    public enum UnbalancedStackActions
    {
        /// <summary>
        /// If the graphics state stack is unbalanced, an exception will be thrown.
        /// </summary>
        Throw,

        /// <summary>
        /// The graphics state stack will be automatically balanced by adding or removing calls to <see cref="Graphics.Restore"/> as necessary.
        /// </summary>
        SilentlyFix,

        /// <summary>
        /// No attempt will be made at correcting an unbalanced graphics state stack. This may cause issues with some consumers.
        /// </summary>
        Ignore
    }
}
```

8.34 BoxBlurFilter.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/
```

```
using System;
using System.Threading.Tasks;

namespace VectSharp.Filters
{
    /// <summary>
    /// Represents a filter applying a box blur.
    /// </summary>
    public class BoxBlurFilter : ILocationInvariantFilter
    {
        /// <summary>
        /// The radius of the box blur (the actual size of the box is 2 * <see cref="BoxRadius"/> + 1).
        /// </summary>
        public double BoxRadius { get; }

        /// <inheritdoc/>
        public Point TopLeftMargin { get; }
        /// <inheritdoc/>
        public Point BottomRightMargin { get; }

        /// <summary>
        /// Creates a new <see cref="BoxBlurFilter"/> with the specified radius.
        /// </summary>
        /// <param name="boxRadius">The radius of the box blur (the actual size of the box is 2 * <paramref name="boxRadius"/> + 1).</param>
        public BoxBlurFilter(double boxRadius)
        {
            this.BoxRadius = boxRadius;
            this.TopLeftMargin = new Point(boxRadius, boxRadius);
            this.BottomRightMargin = new Point(boxRadius, boxRadius);
        }

        /// <inheritdoc/>
        public RasterImage Filter(RasterImage image, double scale)
        {
            return BoxBlurSRGB(image, (int)Math.Round(this.BoxRadius * scale));
        }

        private RasterImage BoxBlurSRGB(RasterImage image, int boxRadius)
        {
            IntPtr intermediateData = System.Runtime.InteropServices.Marshal.AllocHGlobal(image.Width * image.Height * (image.HasAlpha ? 4 : 3));
            IntPtr tbrData = System.Runtime.InteropServices.Marshal.AllocHGlobal(image.Width * image.Height * (image.HasAlpha ? 4 : 3));
            GC.AddMemoryPressure(2 * image.Width * image.Height * (image.HasAlpha ? 4 : 3));

            int width = image.Width;
            int height = image.Height;

            int pixelSize = image.HasAlpha ? 4 : 3;
            int stride = image.Width * pixelSize;

            double normFactor = 1.0 / (2 * boxRadius + 1);

            int threads;

            double size = Math.Sqrt((double)image.Width * image.Height);

            if (size <= 128)
            {
                threads = 1;
            }
            else if (size <= 512)
            {
                threads = Math.Min(4, Environment.ProcessorCount);
            }
            else
            {
                threads = Math.Min(8, Environment.ProcessorCount);
            }

            unsafe
            {
                byte* input = (byte*)image.ImageDataAddress;
                byte* intermediate = (byte*)intermediateData;
                byte* output = (byte*)tbrData;

                Action<int> yLoop;

                if (image.HasAlpha)
                {
                    yLoop = y =>
                    {
                        double rAcc = 0;
                        double gAcc = 0;
```

```
                        double bAcc = 0;
                        double aAcc = 0;

                        double prevR = 0;
                        double prevG = 0;
                        double prevB = 0;
                        double prevA = 0;

                        double r = 0;
                        double g = 0;
                        double b = 0;

                        double a = input[y * stride + 3] / 255.0;

                        prevR = input[y * stride] * a;
                        prevG = input[y * stride + 1] * a;
                        prevB = input[y * stride + 2] * a;
                        prevA = a;

                        rAcc = prevR * (boxRadius + 1);
                        gAcc = prevG * (boxRadius + 1);
                        bAcc = prevB * (boxRadius + 1);
                        aAcc = a * (boxRadius + 1);

                        int rX = 0;

                        for (int x = 0; x < boxRadius; x++)
                        {
                            rX = Math.Min(x, width - 1);

                            a = input[y * stride + rX * 4 + 3] / 255.0;

                            rAcc += input[y * stride + rX * 4] * a;
                            gAcc += input[y * stride + rX * 4 + 1] * a;
                            bAcc += input[y * stride + rX * 4 + 2] * a;
                            aAcc += a;
                        }

                        rX = Math.Min(boxRadius, width - 1);
                        int lX = 0;

                        for (int x = 0; x < width; x++)
                        {
                            rX = Math.Min(x + boxRadius, width - 1);
                            lX = Math.Max(x - boxRadius - 1, 0);

                            a = input[y * stride + rX * 4 + 3] / 255.0;
                            r = input[y * stride + rX * 4] * a;
                            g = input[y * stride + rX * 4 + 1] * a;
                            b = input[y * stride + rX * 4 + 2] * a;

                            prevA = input[y * stride + lX * 4 + 3] / 255.0;
                            prevR = input[y * stride + lX * 4] * prevA;
                            prevG = input[y * stride + lX * 4 + 1] * prevA;
                            prevB = input[y * stride + lX * 4 + 2] * prevA;

                            rAcc += r - prevR;
                            gAcc += g - prevG;
                            bAcc += b - prevB;
                            aAcc += a - prevA;

                            if (aAcc != 0)
                            {
                                intermediate[y * stride + x * 4] = (byte)Math.Min(255, Math.Max(0, rAcc / aAcc));
                                intermediate[y * stride + x * 4 + 1] = (byte)Math.Min(255, Math.Max(0, gAcc / aAcc));
                                intermediate[y * stride + x * 4 + 2] = (byte)Math.Min(255, Math.Max(0, bAcc / aAcc));
                                intermediate[y * stride + x * 4 + 3] = (byte)Math.Min(255, Math.Max(0, aAcc * 255 * normFactor));
                            }
                            else
                            {
                                intermediate[y * stride + x * 4] = 0;
                                intermediate[y * stride + x * 4 + 1] = 0;
                                intermediate[y * stride + x * 4 + 2] = 0;
                                intermediate[y * stride + x * 4 + 3] = 0;
                            }
                        }
                    };
                }
                else
                {
                    yLoop = y =>
                    {
                        double rAcc = 0;
```

```
                        double gAcc = 0;
                        double bAcc = 0;

                        double prevR = 0;
                        double prevG = 0;
                        double prevB = 0;

                        double r = 0;
                        double g = 0;
                        double b = 0;

                        prevR = input[y * stride];
                        prevG = input[y * stride + 1];
                        prevB = input[y * stride + 2];

                        rAcc = prevR * (boxRadius + 1);
                        gAcc = prevG * (boxRadius + 1);
                        bAcc = prevB * (boxRadius + 1);

                        int rX = 0;

                        for (int x = 0; x < boxRadius; x++)
                        {
                            rX = Math.Min(x, width - 1);

                            rAcc += input[y * stride + rX * 3];
                            gAcc += input[y * stride + rX * 3 + 1];
                            bAcc += input[y * stride + rX * 3 + 2];
                        }

                        rX = Math.Min(boxRadius, width - 1);
                        int lX = 0;

                        for (int x = 0; x < width; x++)
                        {
                            rX = Math.Min(x + boxRadius, width - 1);
                            lX = Math.Max(x - boxRadius - 1, 0);

                            r = input[y * stride + rX * 3];
                            g = input[y * stride + rX * 3 + 1];
                            b = input[y * stride + rX * 3 + 2];

                            prevR = input[y * stride + lX * 3];
                            prevG = input[y * stride + lX * 3 + 1];
                            prevB = input[y * stride + lX * 3 + 2];

                            rAcc += r - prevR;
                            gAcc += g - prevG;
                            bAcc += b - prevB;

                            intermediate[y * stride + x * 3] = (byte)Math.Min(255, Math.Max(0, rAcc * normFactor));
                            intermediate[y * stride + x * 3 + 1] = (byte)Math.Min(255, Math.Max(0, gAcc * normFactor));
                            intermediate[y * stride + x * 3 + 2] = (byte)Math.Min(255, Math.Max(0, bAcc * normFactor));
                        }
                    };
                }

                Action<int> xLoop;

                if (image.HasAlpha)
                {
                    xLoop = x =>
                    {
                        double rAcc = 0;
                        double gAcc = 0;
                        double bAcc = 0;
                        double aAcc = 0;

                        double prevR = 0;
                        double prevG = 0;
                        double prevB = 0;
                        double prevA = 0;

                        double r = 0;
                        double g = 0;
                        double b = 0;

                        double a = intermediate[x * 4 + 3] / 255.0;

                        prevR = intermediate[x * 4] * a;
                        prevG = intermediate[x * 4 + 1] * a;
                        prevB = intermediate[x * 4 + 2] * a;
                        prevA = a;

```

```
                        rAcc = prevR * (boxRadius + 1);
                        gAcc = prevG * (boxRadius + 1);
                        bAcc = prevB * (boxRadius + 1);
                        aAcc = a * (boxRadius + 1);

                        int rY = 0;

                        for (int y = 0; y < boxRadius; y++)
                        {
                            rY = Math.Min(y, height - 1);

                            a = intermediate[rY * stride + x * 4 + 3] / 255.0;

                            rAcc += intermediate[rY * stride + x * 4] * a;
                            gAcc += intermediate[rY * stride + x * 4 + 1] * a;
                            bAcc += intermediate[rY * stride + x * 4 + 2] * a;
                            aAcc += a;
                        }

                        rY = Math.Min(boxRadius, height - 1);
                        int lY = 0;

                        for (int y = 0; y < height; y++)
                        {
                            rY = Math.Min(y + boxRadius, height - 1);
                            lY = Math.Max(y - boxRadius - 1, 0);

                            a = intermediate[rY * stride + x * 4 + 3] / 255.0;
                            r = intermediate[rY * stride + x * 4] * a;
                            g = intermediate[rY * stride + x * 4 + 1] * a;
                            b = intermediate[rY * stride + x * 4 + 2] * a;

                            prevA = intermediate[lY * stride + x * 4 + 3] / 255.0;
                            prevR = intermediate[lY * stride + x * 4] * prevA;
                            prevG = intermediate[lY * stride + x * 4 + 1] * prevA;
                            prevB = intermediate[lY * stride + x * 4 + 2] * prevA;

                            rAcc += r - prevR;
                            gAcc += g - prevG;
                            bAcc += b - prevB;
                            aAcc += a - prevA;

                            if (aAcc != 0)
                            {
                                output[y * stride + x * 4] = (byte)Math.Min(255, Math.Max(0, rAcc / aAcc));
                                output[y * stride + x * 4 + 1] = (byte)Math.Min(255, Math.Max(0, gAcc / aAcc));
                                output[y * stride + x * 4 + 2] = (byte)Math.Min(255, Math.Max(0, bAcc / aAcc));
                                output[y * stride + x * 4 + 3] = (byte)Math.Min(255, Math.Max(0, aAcc * 255 * normFactor));
                            }
                            else
                            {
                                output[y * stride + x * 4] = 0;
                                output[y * stride + x * 4 + 1] = 0;
                                output[y * stride + x * 4 + 2] = 0;
                                output[y * stride + x * 4 + 3] = 0;
                            }
                        }
                    };
                }
                else
                {
                    xLoop = x =>
                    {
                        double rAcc = 0;
                        double gAcc = 0;
                        double bAcc = 0;

                        double prevR = 0;
                        double prevG = 0;
                        double prevB = 0;

                        double r = 0;
                        double g = 0;
                        double b = 0;

                        prevR = intermediate[x * 3];
                        prevG = intermediate[x * 3 + 1];
                        prevB = intermediate[x * 3 + 2];

                        rAcc = prevR * (boxRadius + 1);
                        gAcc = prevG * (boxRadius + 1);
                        bAcc = prevB * (boxRadius + 1);

```

```
                        int rY = 0;

                        for (int y = 0; y < boxRadius; y++)
                        {
                            rY = Math.Min(y, height - 1);

                            rAcc += intermediate[rY * stride + x * 3];
                            gAcc += intermediate[rY * stride + x * 3 + 1];
                            bAcc += intermediate[rY * stride + x * 3 + 2];
                        }

                        rY = Math.Min(boxRadius, height - 1);
                        int lY = 0;

                        for (int y = 0; y < height; y++)
                        {
                            rY = Math.Min(y + boxRadius, height - 1);
                            lY = Math.Max(y - boxRadius - 1, 0);

                            r = intermediate[rY * stride + x * 3];
                            g = intermediate[rY * stride + x * 3 + 1];
                            b = intermediate[rY * stride + x * 3 + 2];

                            prevR = intermediate[lY * stride + x * 3];
                            prevG = intermediate[lY * stride + x * 3 + 1];
                            prevB = intermediate[lY * stride + x * 3 + 2];

                            rAcc += r - prevR;
                            gAcc += g - prevG;
                            bAcc += b - prevB;

                            output[y * stride + x * 3] = (byte)Math.Min(255, Math.Max(0, rAcc * normFactor));
                            output[y * stride + x * 3 + 1] = (byte)Math.Min(255, Math.Max(0, gAcc * normFactor));
                            output[y * stride + x * 3 + 2] = (byte)Math.Min(255, Math.Max(0, bAcc * normFactor));
                        }
                    };
                }

                if (threads == 1)
                {
                    for (int y = 0; y < height; y++)
                    {
                        yLoop(y);
                    }

                    for (int x = 0; x < width; x++)
                    {
                        xLoop(x);
                    }
                }
                else
                {
                    ParallelOptions options = new ParallelOptions() { MaxDegreeOfParallelism = threads };

                    Parallel.For(0, height, options, yLoop);
                    Parallel.For(0, width, options, xLoop);
                }
            }

            System.Runtime.InteropServices.Marshal.FreeHGlobal(intermediateData);
            GC.RemoveMemoryPressure(image.Width * image.Height * (image.HasAlpha ? 4 : 3));

            DisposableIntPtr disp = new DisposableIntPtr(tbrData);

            return new RasterImage(ref disp, image.Width, image.Height, image.HasAlpha, image.Interpolate);
        }
    }
}
```

8.35 ColourMatrixFilter.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.
```

```

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Threading.Tasks;

namespace VectSharp.Filters
{
    /// <summary>
    /// Represents a colour transformation matrix.
    /// </summary>
    public class ColourMatrix
    {
        /// <summary>
        /// The coefficient relating the R component of the output colour to the R component of the input colour.
        /// </summary>
        public double R1 { get; set; }

        /// <summary>
        /// The coefficient relating the R component of the output colour to the G component of the input colour.
        /// </summary>
        public double R2 { get; set; }

        /// <summary>
        /// The coefficient relating the R component of the output colour to the B component of the input colour.
        /// </summary>
        public double R3 { get; set; }

        /// <summary>
        /// The coefficient relating the R component of the output colour to the A component of the input colour.
        /// </summary>
        public double R4 { get; set; }

        /// <summary>
        /// The bias (translation) applied to the R component of the output colour.
        /// </summary>
        public double R5 { get; set; }

        /// <summary>
        /// The coefficient relating the G component of the output colour to the R component of the input colour.
        /// </summary>
        public double G1 { get; set; }

        /// <summary>
        /// The coefficient relating the G component of the output colour to the G component of the input colour.
        /// </summary>
        public double G2 { get; set; }

        /// <summary>
        /// The coefficient relating the G component of the output colour to the B component of the input colour.
        /// </summary>
        public double G3 { get; set; }

        /// <summary>
        /// The coefficient relating the G component of the output colour to the A component of the input colour.
        /// </summary>
        public double G4 { get; set; }

        /// <summary>
        /// The bias (translation) applied to the R component of the output colour.
        /// </summary>
        public double G5 { get; set; }

        /// <summary>
        /// The coefficient relating the B component of the output colour to the R component of the input colour.
        /// </summary>
        public double B1 { get; set; }

        /// <summary>
        /// The coefficient relating the B component of the output colour to the G component of the input colour.
```

```
        /// </summary>
        public double B2 { get; set; }

        /// <summary>
        /// The coefficient relating the B component of the output colour to the B component of the input colour.
        /// </summary>
        public double B3 { get; set; }

        /// <summary>
        /// The coefficient relating the B component of the output colour to the A component of the input colour.
        /// </summary>
        public double B4 { get; set; }

        /// <summary>
        /// The bias (translation) applied to the B component of the output colour.
        /// </summary>
        public double B5 { get; set; }

        /// <summary>
        /// The coefficient relating the A component of the output colour to the R component of the input colour.
        /// </summary>
        public double A1 { get; set; }

        /// <summary>
        /// The coefficient relating the A component of the output colour to the G component of the input colour.
        /// </summary>
        public double A2 { get; set; }

        /// <summary>
        /// The coefficient relating the A component of the output colour to the B component of the input colour.
        /// </summary>
        public double A3 { get; set; }

        /// <summary>
        /// The coefficient relating the A component of the output colour to the A component of the input colour.
        /// </summary>
        public double A4 { get; set; }

        /// <summary>
        /// The bias (translation) applied to the A component of the output colour.
        /// </summary>
        public double A5 { get; set; }

        /// <summary>
        /// Gets or sets the requested element of the matrix. Elements of the last row of the matrix can be read, but not set.
        /// </summary>
        /// <param name="y">The row of the matrix.</param>
        /// <param name="x">The column of the matrix.</param>
        /// <returns>The requested element of the matrix.</returns>
        /// <exception cref="ArgumentOutOfRangeException">An attempt has been made to access an element out of the bounds of the matrix.</exception>
        public double this[int y, int x]
        {
            get
            {
                switch (y)
                {
                    case 0:
                        switch (x)
                        {
                            case 0:
                                return R1;
                            case 1:
                                return R2;
                            case 2:
                                return R3;
                            case 3:
                                return R4;
                            case 4:
                                return R5;
                            default:
                                throw new ArgumentOutOfRangeException(nameof(x), x, "Coordinate out of range!");
                        }

                    case 1:
                        switch (x)
                        {
                            case 0:
                                return G1;
```

```
                            case 1:
                                return G2;
                            case 2:
                                return G3;
                            case 3:
                                return G4;
                            case 4:
                                return G5;
                            default:
                                throw new ArgumentOutOfRangeException(nameof(x), x, "Coordinate out of range!");
                        }

                    case 2:
                        switch (x)
                        {
                            case 0:
                                return B1;
                            case 1:
                                return B2;
                            case 2:
                                return B3;
                            case 3:
                                return B4;
                            case 4:
                                return B5;
                            default:
                                throw new ArgumentOutOfRangeException(nameof(x), x, "Coordinate out of range!");
                        }

                    case 3:
                        switch (x)
                        {
                            case 0:
                                return A1;
                            case 1:
                                return A2;
                            case 2:
                                return A3;
                            case 3:
                                return A4;
                            case 4:
                                return A5;
                            default:
                                throw new ArgumentOutOfRangeException(nameof(x), x, "Coordinate out of range!");
                        }

                    case 4:
                        switch (x)
                        {
                            case 0:
                                return 0;
                            case 1:
                                return 0;
                            case 2:
                                return 0;
                            case 3:
                                return 0;
                            case 4:
                                return 1;
                            default:
                                throw new ArgumentOutOfRangeException(nameof(x), x, "Coordinate out of range!");
                        }

                    default:
                        throw new ArgumentOutOfRangeException(nameof(y), y, "Coordinate out of range!");
                }
            }

            private set
            {
                switch (y)
                {
                    case 0:
                        switch (x)
                        {
                            case 0:
                                R1 = value;
                                break;
                            case 1:
                                R2 = value;
                                break;
                            case 2:
```

```
                                R3 = value;
                                break;
                            case 3:
                                R4 = value;
                                break;
                            case 4:
                                R5 = value;
                                break;
                            default:
                                throw new ArgumentOutOfRangeException(nameof(x), x, "Coordinate out of range!");
                        }
                        break;

                    case 1:
                        switch (x)
                        {
                            case 0:
                                G1 = value;
                                break;
                            case 1:
                                G2 = value;
                                break;
                            case 2:
                                G3 = value;
                                break;
                            case 3:
                                G4 = value;
                                break;
                            case 4:
                                G5 = value;
                                break;
                            default:
                                throw new ArgumentOutOfRangeException(nameof(x), x, "Coordinate out of range!");
                        }
                        break;

                    case 2:
                        switch (x)
                        {
                            case 0:
                                B1 = value;
                                break;
                            case 1:
                                B2 = value;
                                break;
                            case 2:
                                B3 = value;
                                break;
                            case 3:
                                B4 = value;
                                break;
                            case 4:
                                B5 = value;
                                break;
                            default:
                                throw new ArgumentOutOfRangeException(nameof(x), x, "Coordinate out of range!");
                        }
                        break;

                    case 3:
                        switch (x)
                        {
                            case 0:
                                A1 = value;
                                break;
                            case 1:
                                A2 = value;
                                break;
                            case 2:
                                A3 = value;
                                break;
                            case 3:
                                A4 = value;
                                break;
                            case 4:
                                A5 = value;
                                break;
                            default:
                                throw new ArgumentOutOfRangeException(nameof(x), x, "Coordinate out of range!");
                        }
                        break;

                    default:
```

```
                        throw new ArgumentOutOfRangeException(nameof(y), y, "Coordinate out of range!");
                }
            }
        }

        /// <summary>
        /// A <see cref="ColourMatrix"/> that whose output colour is always the same as the input colour.
        /// </summary>
        public static ColourMatrix Identity = new ColourMatrix(new double[,] { { 1, 0, 0, 0, 0 }, { 0, 1, 0, 0, 0 }, { 0, 0, 1, 0, 0 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } });

        /// <summary>
        /// A <see cref="ColourMatrix"/> that transforms every colour in a shade of grey with approximately the same luminance.
        /// </summary>
        public static ColourMatrix GreyScale = new ColourMatrix(new double[,] { { 0.2126, 0.7152, 0.0722, 0, 0 }, { 0.2126, 0.7152, 0.0722, 0, 0 }, { 0.2126, 0.7152, 0.0722, 0, 0 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } });

        /// <summary>
        /// A <see cref="ColourMatrix"/> producing a "pastel" (desaturation) effect.
        /// </summary>
        public static ColourMatrix Pastel = new ColourMatrix(new double[,] { { 0.75, 0.25, 0.25, 0, 0 }, { 0.25, 0.75, 0.25, 0, 0 }, { 0.25, 0.25, 0.75, 0, 0 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } });

        /// <summary>
        /// A <see cref="ColourMatrix"/> that inverts every colour, leaving the alpha component intact.
        /// </summary>
        public static ColourMatrix Inversion = new ColourMatrix(new double[,] { { -1, 0, 0, 0, 1 }, { 0, -1, 0, 0, 1 }, { 0, 0, -1, 0, 1 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } });

        /// <summary>
        /// A <see cref="ColourMatrix"/> that inverts the alpha component, leaving the other components intact.
        /// </summary>
        public static ColourMatrix AlphaInversion = new ColourMatrix(new double[,] { { 1, 0, 0, 0, 0 }, { 0, 1, 0, 0, 0 }, { 0, 0, 1, 0, 0 }, { 0, 0, 0, -1, 1 }, { 0, 0, 0, 0, 1 } });

        /// <summary>
        /// A <see cref="ColourMatrix"/> that shifts every colour component by an amount corresponding to the inverted alpha value. The alpha component is left intact.
        /// </summary>
        public static ColourMatrix InvertedAlphaShift = new ColourMatrix(new double[,] { { 1, 0, 0, -1, 1 }, { 0, 1, 0, -1, 1 }, { 0, 0, 1, -1, 1 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } });

        /// <summary>
        /// Creates a <see cref="ColourMatrix"/> that turns every colour to which it is applied into the specified <paramref name="colour"/>.
        /// </summary>
        /// <param name="colour">The colour that will be produced by the <see cref="ColourMatrix"/>.</param>
        /// <param name="useAlpha">If this is <see langword="true"/>, all output pixels will have the same alpha value as the supplied <paramref name="colour"/>. If this is false, the alpha value of the input pixels is preserved.</param>
        /// <returns>A <see cref="ColourMatrix"/> that turns every colour to which it is applied into the specified <paramref name="colour"/>.</returns>
        public static ColourMatrix ToColour(Colour colour, bool useAlpha = false)
        {
            if (!useAlpha)
            {
                return new ColourMatrix(new double[,] { { 0, 0, 0, 0, colour.R }, { 0, 0, 0, 0, colour.G }, { 0, 0, 0, 0, colour.B }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } });
            }
            else
            {
                return new ColourMatrix(new double[,] { { 0, 0, 0, 0, colour.R }, { 0, 0, 0, 0, colour.G }, { 0, 0, 0, 0, colour.B }, { 0, 0, 0, 0, colour.A }, { 0, 0, 0, 0, 1 } });
            }
        }

        /// <summary>
        /// Creates a <see cref="ColourMatrix"/> that turns every colour to which it is applied into a shade of the specified <paramref name="colour"/>. The brightness of the output colour depends on the luminance of the input colour.
        /// </summary>
        /// <param name="colour">The colour whose shades will be produced by the <see cref="ColourMatrix"/>.</param>
        /// <param name="useAlpha">If this is <see langword="true"/>, the transformation will also be applied to the alpha channel. If this is false, the alpha value of the input pixels is preserved.</param>
        /// <returns>A <see cref="ColourMatrix"/> that turns every colour to which it is applied into a shade of the specified <paramref name="colour"/>.</returns>
        public static ColourMatrix LuminanceToColour(Colour colour, bool useAlpha = false)
        {
```

```
            if (!useAlpha)
            {
                return new ColourMatrix(new double[,] { { 0.2126 * colour.R, 0.7152 * colour.R, 0.0722 * colour.R, 0, 0 }, { 0.2126 * colour.G, 0.7152 * colour.G, 0.0722 * colour.G, 0, 0 }, { 0.2126 * colour.B, 0.7152 * colour.B, 0.0722 * colour.B, 0, 0 }, { 0, 0, 0, 1, 0 }, { 0, 0, 0, 0, 1 } });
            }
            else
            {
                return new ColourMatrix(new double[,] { { 0.2126 * colour.R, 0.7152 * colour.R, 0.0722 * colour.R, 0, 0 }, { 0.2126 * colour.G, 0.7152 * colour.G, 0.0722 * colour.G, 0, 0 }, { 0.2126 * colour.B, 0.7152 * colour.B, 0.0722 * colour.B, 0, 0 }, { 0.2126 * colour.A, 0.7152 * colour.A, 0.0722 * colour.A, 0, 0 }, { 0, 0, 0, 0, 1 } });
            }
        }

        /// <summary>
        /// Creates a <see cref="ColourMatrix"/> that transforms the alpha value of the colour it is applied to into a value depending on the luminance of the input colour.
        /// </summary>
        /// <param name="preserveColour">If this is <see langword="true"/>, the values of the red, green and blue components of the input colour are preserved in the output colour. If this is false, the output colour will always be black.</param>
        /// <returns>A <see cref="ColourMatrix"/> that transforms the alpha value of the colour it is applied to into a value depending on the luminance of the input colour.</returns>
        public static ColourMatrix LuminanceToAlpha(bool preserveColour = false)
        {
            if (!preserveColour)
            {
                return new ColourMatrix(new double[,] { { 0, 0, 0, 0, 0 }, { 0, 0, 0, 0, 0 }, { 0, 0, 0, 0, 0 }, { 0.2126, 0.7152, 0.0722, 0, 0 }, { 0, 0, 0, 0, 1 } });
            }
            else
            {
                return new ColourMatrix(new double[,] { { 1, 0, 0, 0, 0 }, { 0, 1, 0, 0, 0 }, { 0, 0, 1, 0, 0 }, { 0.2126, 0.7152, 0.0722, 0, 0 }, { 0, 0, 0, 0, 1 } });
            }
        }

        /// <summary>
        /// Creates a new <see cref="ColourMatrix"/> whose alpha coefficients are multiplied by the specified value.
        /// </summary>
        /// <param name="alpha">The value that will be used to multiply all the alpha coefficients of the <see cref="ColourMatrix"/>.</param>
        /// <returns>A new <see cref="ColourMatrix"/> whose alpha coefficients have been multiplied by the specified value.</returns>
        public ColourMatrix WithAlpha(double alpha)
        {
            return new ColourMatrix(new double[,] { { R1, R2, R3, R4, R5 }, { G1, G2, G3, G4, G5 }, { B1, B2, B3, B4, B5 }, { A1 * alpha, A2 * alpha, A3 * alpha, A4 * alpha, A5 * alpha }, { 0, 0, 0, 0, 1 } });
        }

        /// <summary>
        /// Concatenates two matrices. The resulting <see cref="ColourMatrix"/> is equivalent to first applying <paramref name="matrix2"/>, and then <paramref name="matrix1"/>.
        /// </summary>
        /// <param name="matrix1">The matrix that acts second.</param>
        /// <param name="matrix2">The matrix that acts first.</param>
        /// <returns>A <see cref="ColourMatrix"/> equivalent to first applying <paramref name="matrix2"/>, and then <paramref name="matrix1"/>.</returns>
        public static ColourMatrix operator *(ColourMatrix matrix1, ColourMatrix matrix2)
        {
            double[,] tbr = new double[5, 5];

            for (int i = 0; i < 5; i++)
            {
                for (int j = 0; j < 5; j++)
                {
                    for (int k = 0; k < 5; k++)
                    {
                        tbr[i, j] += matrix1[i, k] * matrix2[k, j];
                    }
                }
            }

            return new ColourMatrix(tbr);
        }

        /// <summary>
        /// Creates a new <see cref="ColourMatrix"/> with the specified coefficients.
        /// </summary>
        /// <param name="matrix">The coefficients of the <see cref="ColourMatrix"/>.</param>
        public ColourMatrix(double[,] matrix)
        {
            for (int i = 0; i < 4; i++)
            {
```

```
                for (int j = 0; j < 5; j++)
                {
                    this[i, j] = matrix[i, j];
                }
            }
        }

        /// <summary>
        /// Applies the <see cref="ColourMatrix"/> to the specified <see cref="Colour"/>.
        /// </summary>
        /// <param name="colour">The <see cref="Colour"/> to which the <see cref="ColourMatrix"/> should be applied.</param>
        /// <returns>The result of applying the <see cref="ColourMatrix"/> to the specified colour.</returns>
        public Colour Apply(Colour colour)
        {
            double[] col = new double[] { colour.R, colour.G, colour.B, colour.A, 1 };

            double[] tbr = new double[5];

            for (int i = 0; i < tbr.Length; i++)
            {
                for (int j = 0; j < tbr.Length; j++)
                {
                    tbr[i] += this[i, j] * col[j];
                }
            }

            return Colour.FromRgba(tbr[0], tbr[1], tbr[2], tbr[3]);
        }

        /// <summary>
        /// Applies the <see cref="ColourMatrix"/> to the specified colour, represented as four bytes, and stores the resulting colour in the
        /// same variables as the original RGBA values.
        /// </summary>
        /// <param name="R">The R component of the input colour. After this method returns, this will contain the R component of the output colour.</param>
        /// <param name="G">The G component of the input colour. After this method returns, this will contain the G component of the output colour.</param>
        /// <param name="B">The B component of the input colour. After this method returns, this will contain the B component of the output colour.</param>
        /// <param name="A">The A component of the input colour. After this method returns, this will contain the A component of the output colour.</param>
        public void Apply(ref byte R, ref byte G, ref byte B, ref byte A)
        {
            byte r = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.R1 + G * this.R2 + B * this.R3 + A * this.R4 + this.R5 * 255)));
            byte g = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.G1 + G * this.G2 + B * this.G3 + A * this.G4 + this.G5 * 255)));
            byte b = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.B1 + G * this.B2 + B * this.B3 + A * this.B4 + this.B5 * 255)));
            byte a = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.A1 + G * this.A2 + B * this.A3 + A * this.A4 + this.A5 * 255)));

            R = r;
            G = g;
            B = b;
            A = a;
        }

        /// <summary>
        /// Applies the <see cref="ColourMatrix"/> to the specified colour, represented as three bytes, and stores the resulting colour in the
        /// same variables as the original RGB values.
        /// </summary>
        /// <param name="R">The R component of the input colour. After this method returns, this will contain the R component of the output colour.</param>
        /// <param name="G">The G component of the input colour. After this method returns, this will contain the G component of the output colour.</param>
        /// <param name="B">The B component of the input colour. After this method returns, this will contain the B component of the output colour.</param>
        public void Apply(ref byte R, ref byte G, ref byte B)
        {
            byte r = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.R1 + G * this.R2 + B * this.R3 + 255 * this.R4 + this.R5 * 255)));
            byte g = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.G1 + G * this.G2 + B * this.G3 + 255 * this.G4 + this.G5 * 255)));
            byte b = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.B1 + G * this.B2 + B * this.B3 + 255 * this.B4 + this.B5 * 255)));

            R = r;
            G = g;
            B = b;
        }

        /// <summary>
```

```
        /// Applies the <see cref="ColourMatrix"/> to the specified colour, represented as four bytes, and stores the resulting colour in the
        /// specified output bytes.
        /// </summary>
        /// <param name="R">The R component of the input colour.</param>
        /// <param name="G">The G component of the input colour.</param>
        /// <param name="B">The B component of the input colour.</param>
        /// <param name="A">The A component of the input colour.</param>
        /// <param name="r">After this method returns, this will contain the R component of the output colour.</param>
        /// <param name="g">After this method returns, this will contain the G component of the output colour.</param>
        /// <param name="b">After this method returns, this will contain the B component of the output colour.</param>
        /// <param name="a">After this method returns, this will contain the A component of the output colour.</param>
        public void Apply(byte R, byte G, byte B, byte A, out byte r, out byte g, out byte b, out byte a)
        {
            r = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.R1 + G * this.R2 + B * this.R3 + A * this.R4 + this.R5 * 255)));
            g = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.G1 + G * this.G2 + B * this.G3 + A * this.G4 + this.G5 * 255)));
            b = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.B1 + G * this.B2 + B * this.B3 + A * this.B4 + this.B5 * 255)));
            a = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.A1 + G * this.A2 + B * this.A3 + A * this.A4 + this.A5 * 255)));
        }

        /// <summary>
        /// Applies the <see cref="ColourMatrix"/> to the specified colour, represented as three bytes, and stores the resulting colour in the
        /// specified output bytes.
        /// </summary>
        /// <param name="R">The R component of the input colour.</param>
        /// <param name="G">The G component of the input colour.</param>
        /// <param name="B">The B component of the input colour.</param>
        /// <param name="r">After this method returns, this will contain the R component of the output colour.</param>
        /// <param name="g">After this method returns, this will contain the G component of the output colour.</param>
        /// <param name="b">After this method returns, this will contain the B component of the output colour.</param>
        public void Apply(byte R, byte G, byte B, out byte r, out byte g, out byte b)
        {
            r = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.R1 + G * this.R2 + B * this.R3 + 255 * this.R4 + this.R5 * 255)));
            g = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.G1 + G * this.G2 + B * this.G3 + 255 * this.G4 + this.G5 * 255)));
            b = (byte)Math.Min(255, Math.Max(0, Math.Round(R * this.B1 + G * this.B2 + B * this.B3 + 255 * this.B4 + this.B5 * 255)));
        }
    }

    /// <summary>
    /// Represents a filter that applies a <see cref="Filters.ColourMatrix"/> to the colours of the image.
    /// </summary>
    public class ColourMatrixFilter : ILocationInvariantFilter
    {
        /// <summary>
        /// The <see cref="Filters.ColourMatrix"/> that is applied by this filter.
        /// </summary>
        public ColourMatrix ColourMatrix { get; }

        /// <inheritdoc/>
        public Point TopLeftMargin { get; } = new Point();

        /// <inheritdoc/>
        public Point BottomRightMargin { get; } = new Point();

        /// <summary>
        /// Creates a new <see cref="ColourMatrixFilter"/> with the specified <see cref="Filters.ColourMatrix"/>.
        /// </summary>
        /<param name="colorMatrix">The <see cref="Filters.ColourMatrix"/> that will be applied by the filter.</param>
        public ColourMatrixFilter(ColourMatrix colorMatrix)
        {
            this.ColourMatrix = colorMatrix;
        }

        /// <inheritdoc/>
        public RasterImage Filter(RasterImage image, double scale)
        {
            IntPtr tbrData = System.Runtime.InteropServices.Marshal.AllocHGlobal(image.Width * image.Height * (image.HasAlpha ? 4 : 3));
```

```
            GC.AddMemoryPressure(image.Width * image.Height * (image.HasAlpha ? 4 : 3));

            int width = image.Width;
            int height = image.Height;

            int pixelSize = image.HasAlpha ? 4 : 3;
            int stride = image.Width * pixelSize;

            int threads = Math.Min(8, Environment.ProcessorCount);

            unsafe
            {
                byte* input = (byte*)image.ImageDataAddress;
                byte* output = (byte*)tbrData;

                Action<int> yLoop;

                if (image.HasAlpha)
                {
                    yLoop = (y) =>
                    {
                        for (int x = 0; x < width; x++)
                        {
                            this.ColourMatrix.Apply(input[y * stride + x * 4], input[y * stride + x * 4 + 1], input[y * stride + x * 4 + 2], input[y * stride + x * 4 + 3], out output[y * stride + x * 4], out output[y * stride + x * 4 + 1], out output[y * stride + x * 4 + 2], out output[y * stride + x * 4 + 3]);
                        }
                    };
                }
                else
                {
                    yLoop = (y) =>
                    {
                        for (int x = 0; x < width; x++)
                        {
                            this.ColourMatrix.Apply(input[y * stride + x * 4], input[y * stride + x * 4 + 1], input[y * stride + x * 4 + 2], out output[y * stride + x * 4], out output[y * stride + x * 4 + 1], out output[y * stride + x * 4 + 2]);
                        }
                    };
                }

                if (threads == 1)
                {
                    for (int y = 0; y < height; y++)
                    {
                        yLoop(y);
                    }
                }
                else
                {
                    ParallelOptions options = new ParallelOptions() { MaxDegreeOfParallelism = threads };

                    Parallel.For(0, height, options, yLoop);
                }

            }

            DisposableIntPtr disp = new DisposableIntPtr(tbrData);

            return new RasterImage(ref disp, width, height, image.HasAlpha, image.Interpolate);
        }
    }
}
```

8.36 CompositeFilter.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
```

```
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System.Collections.Generic;
using System.Collections.Immutable;

namespace VectSharp.Filters
{
    /// <summary>
    /// Represents a filter that corresponds to applying multiple <see cref="ILocationInvariantFilter"/>s one after the other.
    /// </summary>
    public class CompositeLocationInvariantFilter : ILocationInvariantFilter
    {
        /// <inheritdoc/>
        public Point TopLeftMargin { get; }

        /// <inheritdoc/>
        public Point BottomRightMargin { get; }

        /// <summary>
        /// The filters that are applied by this filter.
        /// </summary>
        public ImmutableList<ILocationInvariantFilter> Filters { get; }

        /// <summary>
        /// Creates a new <see cref="CompositeLocationInvariantFilter"/> with the specified filters.
        /// </summary>
        /// <param name="filters">The filters that will be applied by the new filter.</param>
        public CompositeLocationInvariantFilter(IEnumerable<ILocationInvariantFilter> filters)
        {
            IEnumerable<ILocationInvariantFilter> flattenedFilters = FlattenFilters(filters);
            this.Filters = ImmutableList.CreateRange(flattenedFilters);

            bool initialised = false;

            foreach (IFilter filter in flattenedFilters)
            {
                if (!initialised)
                {
                    this.TopLeftMargin = filter.TopLeftMargin;
                    this.BottomRightMargin = filter.BottomRightMargin;
                    initialised = true;
                }
                else
                {
                    this.TopLeftMargin = Point.Max(this.TopLeftMargin, filter.TopLeftMargin);
                    this.BottomRightMargin = Point.Max(this.BottomRightMargin, filter.BottomRightMargin);
                }
            }
        }

        /// <summary>
        /// Creates a new <see cref="CompositeLocationInvariantFilter"/> with the specified filters.
        /// </summary>
        /// <param name="filters">The filters that will be applied by the new filter.</param>
        public CompositeLocationInvariantFilter(params ILocationInvariantFilter[] filters) : this((IEnumerable<ILocationInvariantFilter>)filters) {  }

        private IEnumerable<ILocationInvariantFilter> FlattenFilters(IEnumerable<ILocationInvariantFilter> filters)
        {
            foreach (ILocationInvariantFilter filter in filters)
            {
                if (filter is CompositeLocationInvariantFilter composite)
                {
                    foreach (ILocationInvariantFilter filter2 in FlattenFilters(composite.Filters))
                    {
                        yield return filter2;
                    }
                }
                else
                {
                    yield return filter;
                }
            }
        }

        /// <inheritdoc/>
        public RasterImage Filter(RasterImage image, double scale)
        {
            RasterImage currImage = image;

            foreach (ILocationInvariantFilter filter in this.Filters)
            {
                RasterImage prevImage = currImage;
```

```
                currImage = filter.Filter(prevImage, scale);

                if (prevImage != image)
                {
                    prevImage.Dispose();
                }
            }

            return currImage;
        }
    }
}
```

8.37 ConvolutionFilter.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Threading.Tasks;

namespace VectSharp.Filters
{
    /// <summary>
    /// Represents a filter that applies a matrix convolution to the image.
    /// </summary>
    public class ConvolutionFilter : ILocationInvariantFilter
    {
        /// <inheritdoc/>
        public Point TopLeftMargin { get; protected set; }
        /// <inheritdoc/>
        public Point BottomRightMargin { get; protected set; }

        /// <summary>
        /// The kernel of the <see cref="ConvolutionFilter"/>. The dimensions of this matrix should all be odd numbers. The larger the kernel, the worse the performance.
        /// </summary>
        public virtual double[,] Kernel { get; protected set; }

        /// <summary>
        /// The normalisation value that is applies to the kernel.
        /// </summary>
        public virtual double Normalisation { get; protected set; } = 1;

        /// <summary>
        /// The bias value that is added to every colour component when the filter is applied.
        /// </summary>
        public virtual double Bias { get; protected set; } = 0;

        /// <summary>
        /// The scale relating the size of the kernel to graphics units.
        /// </summary>
        public virtual double Scale { get; protected set; }

        /// <summary>
        /// If this is <see langword="true"/>, the alpha value of the input pixels is preserved. Otherwise, the alpha channel is subject to the same convolution process as the other colour components.
        /// </summary>
        public virtual bool PreserveAlpha { get; protected set; } = true;

        /// <summary>
        /// Creates a new <see cref="ConvolutionFilter"/> with the specified parameters.
        /// </summary>
        /// <param name="kernel">The kernel of the <see cref="ConvolutionFilter"/>. The dimensions of this matrix should all be odd numbers. The larger the kernel, the worse the performance.</param>
        /// <param name="scale">The scale relating the size of the kernel to graphics units.</param>
        /// <param name="preserveAlpha">If this is <see langword="true"/>, the alpha value of the input pixels is preserved. Otherwise, the alpha channel is subject to the same convolution process as the other colour components.</param>
```

```
        /// <param name="normalisation">The normalisation value that is applies to the kernel.</param>
        /// <param name="bias">The bias value that is added to every colour component when the filter is applied.</param>
        /// <exception cref="ArgumentException">This exception is thrown when the kernel dimensions are not odd numbers.</exception>
        public ConvolutionFilter(double[,] kernel, double scale, bool preserveAlpha = true, double normalisation = 1, double bias = 0)
        {
            if (kernel.GetLength(0) % 2 != 1 || kernel.GetLength(1) % 2 != 1)
            {
                throw new ArgumentException("The kernel must have an odd number of rows and columns!", nameof(kernel));
            }

            this.Kernel = kernel;

            int kernelWidth = (this.Kernel.GetLength(0) - 1) / 2;
            int kernelHeight = (this.Kernel.GetLength(1) - 1) / 2;

            this.TopLeftMargin = new Point(kernelWidth * scale, kernelHeight * scale);
            this.BottomRightMargin = new Point(kernelWidth * scale, kernelHeight * scale);
            this.Scale = scale;
            this.PreserveAlpha = preserveAlpha;
            this.Normalisation = normalisation;
            this.Bias = bias;
        }

        /// <inheritdoc/>
        public virtual RasterImage Filter(RasterImage image, double scale)
        {
            IntPtr tbrData = System.Runtime.InteropServices.Marshal.AllocHGlobal(image.Width * image.Height * (image.HasAlpha ? 4 : 3));

            int width = image.Width;
            int height = image.Height;

            int kernelWidth = (this.Kernel.GetLength(0) - 1) / 2;
            int kernelHeight = (this.Kernel.GetLength(1) - 1) / 2;

            int actualKernelWidth = (int)Math.Round(kernelWidth * scale * this.Scale);
            int actualKernelHeight = (int)Math.Round(kernelHeight * scale * this.Scale);

            actualKernelWidth = Math.Max(actualKernelWidth, 1);
            actualKernelHeight = Math.Max(actualKernelHeight, 1);

            int[] kernelX = new int[actualKernelWidth * 2 + 1];

            for (int x = 0; x < actualKernelWidth * 2 + 1; x++)
            {
                kernelX[x] = (int)Math.Round((double)(x - actualKernelWidth) / actualKernelWidth * kernelWidth + kernelWidth);
            }

            int[] kernelY = new int[actualKernelHeight * 2 + 1];

            for (int y = 0; y < actualKernelHeight * 2 + 1; y++)
            {
                kernelY[y] = (int)Math.Round((double)(y - actualKernelHeight) / actualKernelHeight * kernelHeight + kernelHeight);
            }

            int[] countsX = new int[2 * kernelWidth + 1];

            for (int i = 0; i < 2 * actualKernelWidth + 1; i++)
            {
                countsX[kernelX[i]]++;
            }

            int[] countsY = new int[2 * kernelHeight + 1];

            for (int i = 0; i < 2 * actualKernelHeight + 1; i++)
            {
                countsY[kernelY[i]]++;
            }


            double[] weightsX = new double[2 * actualKernelWidth + 1];

            for (int i = 0; i < 2 * actualKernelWidth + 1; i++)
            {
                weightsX[i] = 1.0 / countsX[kernelX[i]];
            }

            double[] weightsY = new double[2 * actualKernelHeight + 1];

            for (int i = 0; i < 2 * actualKernelHeight + 1; i++)
            {
```

```
                weightsY[i] = 1.0 / countsY[kernelY[i]];
            }

            kernelWidth = actualKernelWidth;
            kernelHeight = actualKernelHeight;

            double normalisation = this.Normalisation;

            if (double.IsNaN(normalisation) || normalisation == 0)
            {
                normalisation = 1;
            }

            double totalWeight = 0;
            for (int i = 0; i < kernelWidth * 2 + 1; i++)
            {
                for (int j = 0; j < kernelHeight * 2 + 1; j++)
                {
                    totalWeight += Kernel[kernelX[i], kernelY[j]] * weightsX[i] * weightsY[j];
                }
            }

            if (Math.Abs(totalWeight) < 1e-5)
            {
                totalWeight = 1;
            }


            double bias = this.Bias * 255;
            int pixelSize = image.HasAlpha ? 4 : 3;
            int stride = image.Width * pixelSize;

            int threads = Math.Min(8, Environment.ProcessorCount);

            unsafe
            {
                byte* input = (byte*)image.ImageDataAddress;
                byte* output = (byte*)tbrData;

                Action<int> yLoop;

                if (image.HasAlpha && !PreserveAlpha)
                {
                    yLoop = (y) =>
                    {
                        for (int x = 0; x < width; x++)
                        {
                            double R = 0;
                            double G = 0;
                            double B = 0;

                            double weight = 0;

                            for (int targetX = 0; targetX <= kernelWidth * 2; targetX++)
                            {
                                for (int targetY = 0; targetY <= kernelHeight * 2; targetY++)
                                {
                                    int tX = Math.Min(Math.Max(0, x + targetX - kernelWidth), width - 1);
                                    int tY = Math.Min(Math.Max(0, y + targetY - kernelHeight), height - 1);

                                    double a = input[tY * stride + tX * 4 + 3] / 255.0 * weightsX[targetX] * weightsY[targetY];

                                    int projectedX = kernelX[targetX];
                                    int projectedY = kernelY[targetY];

                                    weight += Kernel[projectedX, projectedY] * a;

                                    R += Kernel[projectedX, projectedY] * input[tY * stride + tX * 4] * a;
                                    G += Kernel[projectedX, projectedY] * input[tY * stride + tX * 4 + 1] * a;
                                    B += Kernel[projectedX, projectedY] * input[tY * stride + tX * 4 + 2] * a;
                                }
                            }

                            if (weight != 0)
                            {
                                output[y * stride + x * 4] = (byte)Math.Min(255, Math.Max(0, R / (normalisation * weight) + bias));
                                output[y * stride + x * 4 + 1] = (byte)Math.Min(255, Math.Max(0, G / (normalisation * weight) + bias));
                                output[y * stride + x * 4 + 2] = (byte)Math.Min(255, Math.Max(0, B / (normalisation * weight) + bias));
```

```
                                output[y * stride + x * 4 + 3] = (byte)Math.Min(255, Math.Max(0, (weight / (normalisation * totalWeight) * 255 + bias)));
                            }
                            else
                            {
                                output[y * stride + x * 4] = 0;
                                output[y * stride + x * 4 + 1] = 0;
                                output[y * stride + x * 4 + 2] = 0;
                                output[y * stride + x * 4 + 3] = 0;
                            }
                        }
                    };
                }
                else if (image.HasAlpha && PreserveAlpha)
                {
                    yLoop = (y) =>
                    {
                        for (int x = 0; x < width; x++)
                        {
                            double R = 0;
                            double G = 0;
                            double B = 0;

                            for (int targetX = 0; targetX <= kernelWidth * 2; targetX++)
                            {
                                for (int targetY = 0; targetY <= kernelHeight * 2; targetY++)
                                {
                                    int tX = Math.Min(Math.Max(0, x + targetX - kernelWidth), width - 1);
                                    int tY = Math.Min(Math.Max(0, y + targetY - kernelHeight), height - 1);

                                    int projectedX = kernelX[targetX];
                                    int projectedY = kernelY[targetY];

                                    R += Kernel[projectedX, projectedY] * input[tY * stride + tX * 4] * weightsX[targetX] * weightsY[targetY];
                                    G += Kernel[projectedX, projectedY] * input[tY * stride + tX * 4 + 1] * weightsX[targetX] * weightsY[targetY];
                                    B += Kernel[projectedX, projectedY] * input[tY * stride + tX * 4 + 2] * weightsX[targetX] * weightsY[targetY];
                                }
                            }

                            output[y * stride + x * 4] = (byte)Math.Min(255, Math.Max(0, R / (normalisation * totalWeight) + bias));
                            output[y * stride + x * 4 + 1] = (byte)Math.Min(255, Math.Max(0, G / (normalisation * totalWeight) + bias));
                            output[y * stride + x * 4 + 2] = (byte)Math.Min(255, Math.Max(0, B / (normalisation * totalWeight) + bias));
                            output[y * stride + x * 4 + 3] = input[y * stride + x * 4 + 3];
                        }
                    };
                }
                else
                {
                    yLoop = (y) =>
                    {
                        for (int x = 0; x < width; x++)
                        {
                            double R = 0;
                            double G = 0;
                            double B = 0;

                            for (int targetX = 0; targetX <= kernelWidth * 2; targetX++)
                            {
                                for (int targetY = 0; targetY <= kernelHeight * 2; targetY++)
                                {
                                    int tX = Math.Min(Math.Max(0, x + targetX - kernelWidth), width - 1);
                                    int tY = Math.Min(Math.Max(0, y + targetY - kernelHeight), height - 1);

                                    int projectedX = kernelX[targetX];
                                    int projectedY = kernelY[targetY];

                                    R += Kernel[projectedX, projectedY] * input[tY * stride + tX * 3] * weightsX[targetX] * weightsY[targetY];
                                    G += Kernel[projectedX, projectedY] * input[tY * stride + tX * 3 + 1] * weightsX[targetX] * weightsY[targetY];
                                    B += Kernel[projectedX, projectedY] * input[tY * stride + tX * 3 + 2] * weightsX[targetX] * weightsY[targetY];
                                }
                            }

                            output[y * stride + x * 3] = (byte)Math.Min(255, Math.Max(0, R / (normalisation * totalWeight) + bias));
```

```
                            output[y * stride + x * 3 + 1] = (byte)Math.Min(255, Math.Max(0, G / (normalisation * totalWeight) + bias));
                            output[y * stride + x * 3 + 2] = (byte)Math.Min(255, Math.Max(0, B / (normalisation * totalWeight) + bias));
                        }
                    };
                }

                if (threads == 1)
                {
                    for (int y = 0; y < height; y++)
                    {
                        yLoop(y);
                    }
                }
                else
                {
                    ParallelOptions options = new ParallelOptions() { MaxDegreeOfParallelism = threads };

                    Parallel.For(0, height, options, yLoop);
                }
            }

            return new RasterImage(tbrData, image.Width, image.Height, image.HasAlpha, image.Interpolate);
        }
    }
}
```

8.38 Filters.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;

namespace VectSharp.Filters
{
    /// <summary>
    /// Represents a filter. Do not implement this interface directly; instead, implement <see cref="ILocationInvariantFilter"/> or <see cref="IFilterWithLocation"/>.
    /// </summary>
    public interface IFilter
    {
        /// <summary>
        /// Determines how much the area of the filter's subject should be expanded on the top-left to accommodate the results of the filter.
        /// </summary>
        Point TopLeftMargin { get; }

        /// <summary>
        /// Determines how much the area of the filter's subject should be expanded on the bottom-right to accommodate the results of the filter.
        /// </summary>
        Point BottomRightMargin { get; }
    }

    /// <summary>
    /// Represents a filter that can be applied to an image regardless of its location on the graphics surface.
    /// </summary>
    public interface ILocationInvariantFilter : IFilter
    {
        /// <summary>
        /// Applies the filter to a <see cref="RasterImage"/>.
        /// </summary>
        /// <param name="image">The <see cref="RasterImage"/> to which the filter will be applied.</param>
```

```
        /// <param name="scale">The scale of the image with respect to the filter.</param>
        /// <returns>A new <see cref="RasterImage"/> containing the filtered image. The source <paramref name="image"/> is left unaltered.</returns>
        RasterImage Filter(RasterImage image, double scale);
    }

    /// <summary>
    /// Represents a filter whose results depend on the position of the subject image on the graphics surface.
    /// </summary>
    public interface IFilterWithLocation : IFilter
    {
        /// <summary>
        /// Applies the filter to a <see cref="RasterImage"/>.
        /// </summary>
        /// <param name="image">The <see cref="RasterImage"/> to which the filter will be applied.</param>
        /// <param name="bounds">The region on the graphics surface where the image will be drawn.</param>
        /// <param name="scale">The scale of the image with respect to the filter.</param>
        /// <returns>A new <see cref="RasterImage"/> containing the filtered image. The source <paramref name="image"/> is left unaltered.</returns>
        RasterImage Filter(RasterImage image, Rectangle bounds, double scale);
    }

    /// <summary>
    /// Represents a filter with a parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter.
    /// The <see cref="FilterWithRasterisableParameter"/> abstract class provides a default implementation of this interface.
    /// </summary>
    public interface IFilterWithRasterisableParameter
    {
        /// <summary>
        /// Rasterises the filter's parameter at the specified scale, using the specified rasterisation method.
        /// </summary>
        /// <param name="rasterisationMethod">The method used to rasterise the image. The first argument of this method is the <see cref="Graphics"/> to be rasterised,
        /// the second is a <see cref="Rectangle"/> representing the region to rasterise, the third is a <see cref="double"/> representing the scale, and the third is
        /// a boolean value indicating whether the resulting <see cref="RasterImage"/> should be interpolated.</param>
        /// <param name="scale">The scale factor at which the parameter is rasterised.</param>
        void RasteriseParameter(Func<Graphics, Rectangle, double, bool, RasterImage> rasterisationMethod, double scale);
    }

    /// <summary>
    /// Represents a filter with a parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter.
    /// </summary>
    public abstract class FilterWithRasterisableParameter : IFilterWithRasterisableParameter, IDisposable
    {
        /// <summary>
        /// The parameter that needs to be rasterised.
        /// </summary>
        protected virtual Graphics RasterisableParameter { get; }

        /// <summary>
        /// The result of the last rasterisation of the <see cref="RasterisableParameter"/>.
        /// </summary>
        protected RasterImage cachedRasterisation;
        private bool disposedValue;

        /// <summary>
        /// Get a rasterised version of the <see cref="RasterisableParameter"/> at the specified scale. If this has already been computed, the cached result is returned.
        /// Otherwise, the image is rasterised using the default rasterisation method.
        /// </summary>
        /<param name="scale">The scale at which the <see cref="RasterisableParameter"/> will be rasterised.</param>
        /// <returns>A rasterised version of the <see cref="RasterisableParameter"/> at the specified scale.</returns>
        protected virtual RasterImage GetCachedRasterisation(double scale)
        {
            if (this.cachedRasterisation == null || CachedResolution != scale)
            {
                this.RasteriseParameter(scale);
            }

            return this.cachedRasterisation;
        }

        /// <summary>
        /// The bounds of the last cached rendering of the <see cref="RasterisableParameter"/> on the
```

```
graphics surface.
        /// </summary>
        protected virtual Rectangle CachedBounds { get; set; }

        /// <summary>
        /// The resolution using which the cached rendering of the <see cref="RasterisableParameter"/> has been computed.
        /// </summary>
        protected virtual double CachedResolution { get; set; } = double.NaN;

        /// <summary>
        /// Create a new <see cref="FilterWithRasterisableParameter"/> with the specified parameter.
        /// </summary>
        /// <param name="rasterisableParameter">The parameter that needs to be rasterised at the same resolution as the subject image prior to applying the filter.</param>
        protected FilterWithRasterisableParameter(Graphics rasterisableParameter)
        {
            this.RasterisableParameter = rasterisableParameter;
        }

        /// <inheritdoc />
        public virtual void RasteriseParameter(Func<Graphics, Rectangle, double, bool, RasterImage> rasterisationMethod, double scale)
        {
            Rectangle bounds = RasterisableParameter.GetBounds();

            this.cachedRasterisation?.Dispose();

            this.cachedRasterisation = rasterisationMethod(this.RasterisableParameter, bounds, scale, true);
            this.CachedResolution = scale;
            this.CachedBounds = bounds;
        }

        /// <summary>
        /// Rasterises the filter's parameter at the specified scale, using the default rasterisation method.
        /// </summary>
        /// <param name="scale">The scale factor at which the parameter is rasterised.</param>
        /// <exception cref="NotImplementedException">This exception is thrown when there is no default rasterisation method. This occurs because neither the VectSharp.Raster
        /// assembly, nor the VectSharp.Raster.ImageSharp assembly have been loaded, and no custom implementation of <see cref="Graphics.RasterisationMethod"/> has been provided.</exception>
        protected virtual void RasteriseParameter(double scale)
        {
            Rectangle bounds = this.RasterisableParameter.GetBounds();

            this.cachedRasterisation?.Dispose();

            if (this.RasterisableParameter.TryRasterise(bounds, scale, true, out RasterImage raster))
            {
                this.cachedRasterisation = raster;
                this.CachedResolution = scale;
                this.CachedBounds = bounds;
            }
            else
            {
                throw new NotImplementedException(@"The filter could not be rasterised! You can avoid this error by doing one of the following:
• Add a reference to VectSharp.Raster or VectSharp.Raster.ImageSharp (you may also need to add a using directive somewhere to force the assembly to be loaded).
• Provide your own implementation of Graphics.RasterisationMethod.");
            }
        }

        /// <inheritdoc/>
        protected virtual void Dispose(bool disposing)
        {
            if (!disposedValue)
            {
                if (disposing)
                {
                    this.cachedRasterisation?.Dispose();
                }

                disposedValue = true;
            }
        }

        /// <inheritdoc/>
        public void Dispose()
        {
            Dispose(disposing: true);
            GC.SuppressFinalize(this);
        }
    }

```

```

}
```

8.39 GaussianBlurFilter.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Threading.Tasks;

namespace VectSharp.Filters
{
    /// <summary>
    /// Represents a filter that applies a Gaussian blur effect.
    /// </summary>
    public class GaussianBlurFilter : ILocationInvariantFilter
    {
        /// <summary>
        /// The standard deviation of the Gaussian blur.
        /// </summary>
        public double StandardDeviation { get; }

        /// <inheritdoc/>
        public Point TopLeftMargin { get; }

        /// <inheritdoc/>
        public Point BottomRightMargin { get; }

        /// <summary>
        /// Creates a new <see cref="GaussianBlurFilter"/> with the specified standard deviation.
        /// </summary>
        /// <param name="standardDeviation">The standard deviation of the Gaussian blur.</param>
        public GaussianBlurFilter(double standardDeviation)
        {
            this.StandardDeviation = standardDeviation;
            this.TopLeftMargin = new Point(standardDeviation * 3, standardDeviation * 3);
            this.BottomRightMargin = new Point(standardDeviation * 3, standardDeviation * 3);
        }

        /// <inheritdoc/>
        public RasterImage Filter(RasterImage image, double scale)
        {
            return FilterSRGB(image, scale);
        }

        private RasterImage FilterSRGB(RasterImage image, double scale)
        {
            if (this.StandardDeviation * scale > 2)
            {
                int[] boxes = BoxesForGauss(this.StandardDeviation * scale, 3);

                for (int i = 0; i < boxes.Length; i++)
                {
                    BoxBlurFilter box = new BoxBlurFilter((boxes[i] - 1) / 2);

                    image = box.Filter(image, 1);
                }
            }
            else
            {
                image = FilterSRGBExact(image, this.StandardDeviation * scale);
            }

            return image;
        }

        // Adapted from http://blog.ivank.net/fastest-gaussian-blur.html
        private static int[] BoxesForGauss(double standardDeviation, int count)
```

```
        {
            double idealWeight = Math.Sqrt((12 * standardDeviation * standardDeviation / count) + 1);

            int lowerWeight = (int)Math.Floor(idealWeight);

            if (lowerWeight % 2 == 0)
            {
                lowerWeight--;
            }

            int upperWeight = lowerWeight + 2;

            double idealSize = (12 * standardDeviation * standardDeviation - count * lowerWeight * lowerWeight - 4 * count * lowerWeight - 3 * count) / (-4 * lowerWeight - 4);

            int m = (int)Math.Round(idealSize);

            int[] sizes = new int[count];

            for (int i = 0; i < count; i++)
            {
                sizes[i] = i < m ? lowerWeight : upperWeight;
            }

            return sizes;
        }

        private RasterImage FilterSRGBExact(RasterImage image, double standardDeviation)
        {
            IntPtr intermediateData = System.Runtime.InteropServices.Marshal.AllocHGlobal(image.Width * image.Height * (image.HasAlpha ? 4 : 3));
            IntPtr tbrData = System.Runtime.InteropServices.Marshal.AllocHGlobal(image.Width * image.Height * (image.HasAlpha ? 4 : 3));
            GC.AddMemoryPressure(2 * image.Width * image.Height * (image.HasAlpha ? 4 : 3));

            int width = image.Width;
            int height = image.Height;

            int kernelSize = (int)Math.Ceiling(standardDeviation * 3);

            double[] kernel = new double[kernelSize * 2 + 1];

            double total = 0;

            for (int i = 0; i <= kernelSize; i++)
            {
                kernel[i] = Math.Exp(-(kernelSize - i) * (kernelSize - i) / (2 * standardDeviation * standardDeviation));

                if (i < kernelSize)
                {
                    kernel[2 * kernelSize - i] = kernel[i];

                    total += kernel[i] * 2;
                }
                else
                {
                    total += kernel[i];
                }
            }

            for (int i = 0; i < kernel.Length; i++)
            {
                kernel[i] /= total;
            }

            int pixelSize = image.HasAlpha ? 4 : 3;
            int stride = image.Width * pixelSize;

            int threads;

            double size = Math.Sqrt((double)image.Width * image.Height);

            if (size <= 128)
            {
                threads = 1;
            }
            else if (size <= 512)
            {
                threads = Math.Min(4, Environment.ProcessorCount);
            }
            else
            {
                threads = Math.Min(8, Environment.ProcessorCount);
            }

            unsafe
```

```
            {
                byte* input = (byte*)image.ImageDataAddress;
                byte* intermediate = (byte*)intermediateData;
                byte* output = (byte*)tbrData;

                Action<int> yLoop;

                if (image.HasAlpha)
                {
                    yLoop = y =>
                    {
                        for (int x = 0; x < width; x++)
                        {
                            double R = 0;
                            double G = 0;
                            double B = 0;
                            double weight = 0;

                            for (int targetX = 0; targetX <= kernelSize * 2; targetX++)
                            {
                                int tX = Math.Min(Math.Max(0, x + targetX - kernelSize), width - 1);

                                double a = input[y * stride + tX * 4 + 3] / 255.0;

                                weight += kernel[targetX] * a;

                                R += kernel[targetX] * input[y * stride + tX * 4] * a;
                                G += kernel[targetX] * input[y * stride + tX * 4 + 1] * a;
                                B += kernel[targetX] * input[y * stride + tX * 4 + 2] * a;
                            }

                            if (weight != 0)
                            {
                                intermediate[y * stride + x * 4] = (byte)Math.Min(255, Math.Max(0, R / weight));
                                intermediate[y * stride + x * 4 + 1] = (byte)Math.Min(255, Math.Max(0, G / weight));
                                intermediate[y * stride + x * 4 + 2] = (byte)Math.Min(255, Math.Max(0, B / weight));
                                intermediate[y * stride + x * 4 + 3] = (byte)Math.Min(255, Math.Max(0, weight * 255));
                            }
                            else
                            {
                                intermediate[y * stride + x * 4] = 0;
                                intermediate[y * stride + x * 4 + 1] = 0;
                                intermediate[y * stride + x * 4 + 2] = 0;
                                intermediate[y * stride + x * 4 + 3] = 0;
                            }
                        }
                    };
                }
                else
                {
                    yLoop = y =>
                    {
                        for (int x = 0; x < width; x++)
                        {
                            double R = 0;
                            double G = 0;
                            double B = 0;

                            for (int targetX = 0; targetX <= kernelSize * 2; targetX++)
                            {
                                int tX = Math.Min(Math.Max(0, x + targetX - kernelSize), width - 1);

                                R += kernel[targetX] * input[y * stride + tX * 3];
                                G += kernel[targetX] * input[y * stride + tX * 3 + 1];
                                B += kernel[targetX] * input[y * stride + tX * 3 + 2];
                            }

                            intermediate[y * stride + x * 3] = (byte)Math.Min(255, Math.Max(0, R));
                            intermediate[y * stride + x * 3 + 1] = (byte)Math.Min(255, Math.Max(0, G));
                            intermediate[y * stride + x * 3 + 2] = (byte)Math.Min(255, Math.Max(0, B));
                        }
                    };
                }

                Action<int> xLoop;

                if (image.HasAlpha)
                {
                    xLoop = y =>
                    {
                        for (int x = 0; x < width; x++)
```

```
                        {
                            double R = 0;
                            double G = 0;
                            double B = 0;


                            double weight = 0;

                            for (int targetY = 0; targetY <= kernelSize * 2; targetY++)
                            {
                                int tY = Math.Min(Math.Max(0, y + targetY - kernelSize), height - 1);

                                double a = intermediate[tY * stride + x * 4 + 3] / 255.0;

                                weight += kernel[targetY] * a;

                                R += kernel[targetY] * intermediate[tY * stride + x * 4] * a;
                                G += kernel[targetY] * intermediate[tY * stride + x * 4 + 1] * a;
                                B += kernel[targetY] * intermediate[tY * stride + x * 4 + 2] * a;
                            }

                            if (weight != 0)
                            {
                                output[y * stride + x * 4] = (byte)Math.Min(255, Math.Max(0, R / weight));
                                output[y * stride + x * 4 + 1] = (byte)Math.Min(255, Math.Max(0, G / weight));
                                output[y * stride + x * 4 + 2] = (byte)Math.Min(255, Math.Max(0, B / weight));
                                output[y * stride + x * 4 + 3] = (byte)Math.Min(255, Math.Max(0, weight * 255));
                            }
                            else
                            {
                                output[y * stride + x * 4] = 0;
                                output[y * stride + x * 4 + 1] = 0;
                                output[y * stride + x * 4 + 2] = 0;
                                output[y * stride + x * 4 + 3] = 0;
                            }
                        }
                    };
                }
                else
                {
                    xLoop = y =>
                    {
                        for (int x = 0; x < width; x++)
                        {
                            double R = 0;
                            double G = 0;
                            double B = 0;

                            for (int targetY = 0; targetY <= kernelSize * 2; targetY++)
                            {
                                int tY = Math.Min(Math.Max(0, y + targetY - kernelSize), height - 1);

                                R += kernel[targetY] * intermediate[tY * stride + x * 3];
                                G += kernel[targetY] * intermediate[tY * stride + x * 3 + 1];
                                B += kernel[targetY] * intermediate[tY * stride + x * 3 + 2];
                            }

                            output[y * stride + x * 3] = (byte)Math.Min(255, Math.Max(0, R));
                            output[y * stride + x * 3 + 1] = (byte)Math.Min(255, Math.Max(0, G));
                            output[y * stride + x * 3 + 2] = (byte)Math.Min(255, Math.Max(0, B));
                        }
                    };
                }



                if (threads == 1)
                {
                    for (int y = 0; y < height; y++)
                    {
                        yLoop(y);
                    }

                    for (int y = 0; y < height; y++)
                    {
                        xLoop(y);
                    }
                }
                else
                {
                    ParallelOptions options = new ParallelOptions() { MaxDegreeOfParallelism = threads };

```

```
                    Parallel.For(0, height, options, yLoop);
                    Parallel.For(0, height, options, xLoop);
                }
            }

            System.Runtime.InteropServices.Marshal.FreeHGlobal(intermediateData);
            GC.RemoveMemoryPressure(image.Width * image.Height * (image.HasAlpha ? 4 : 3));

            DisposableIntPtr disp = new DisposableIntPtr(tbrData);

            return new RasterImage(ref disp, image.Width, image.Height, image.HasAlpha, image.Interpolate);
        }

    }
}
```

8.40 MaskFilter.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Threading.Tasks;

namespace VectSharp.Filters
{
    /// <summary>
    /// Represents a filter that uses the luminance of an image to mask another image.
    /// </summary>
    public class MaskFilter : FilterWithRasterisableParameter, IFilterWithLocation
    {
        /// <inheritdoc/>
        public Point TopLeftMargin => new Point(0, 0);
        /// <inheritdoc/>
        public Point BottomRightMargin => new Point(0, 0);

        /// <summary>
        /// The image that is used to mask the input image.
        /// </summary>
        public Graphics Mask => this.RasterisableParameter;

        /// <summary>
        /// Creates a new <see cref="MaskFilter"/> with the specified mask image.
        /// </summary>
        /// <param name="mask">The image that is used to mask the input image.</param>
        public MaskFilter(Graphics mask) : base(mask) { }

        /// <inheritdoc/>
        public RasterImage Filter(RasterImage image, Rectangle bounds, double scale)
        {
            RasterImage mask = this.GetCachedRasterisation(scale);

            IntPtr tbrData = System.Runtime.InteropServices.Marshal.AllocHGlobal(image.Width * image.Height * 4);
            GC.AddMemoryPressure(image.Width * image.Height * 4);

            int width = image.Width;
            int height = image.Height;

            int pixelSizeInput = image.HasAlpha ? 4 : 3;
            int strideInput = image.Width * pixelSizeInput;

            int pixelSizeOutput = 4;
            int strideOutput = image.Width * pixelSizeOutput;

            int pixelSizeMask = 4;
            int strideMask = mask.Width * pixelSizeMask;
```

```

            int maskWidth = mask.Width;
            int maskHeight = mask.Height;

            int threads = Math.Min(8, Environment.ProcessorCount);

            unsafe
            {
                byte* input = (byte*)image.ImageDataAddress;
                byte* maskBytes = (byte*)mask.ImageDataAddress;
                byte* output = (byte*)tbrData;

                Action<int> yLoop;

                if (image.HasAlpha)
                {
                    yLoop = (y) =>
                    {
                        for (int x = 0; x < width; x++)
                        {
                            int maskX = (int)Math.Round((bounds.Location.X + (x + 0.5) * bounds.Size.Width / width - CachedBounds.Location.X) / CachedBounds.Size.Width * maskWidth);
                            int maskY = (int)Math.Round((bounds.Location.Y + (y + 0.5) * bounds.Size.Height / height - CachedBounds.Location.Y) / CachedBounds.Size.Height * maskHeight);

                            double weight = 0;

                            if (maskX >= 0 && maskX < maskWidth && maskY >= 0 && maskY < maskHeight)
                            {
                                weight = (maskBytes[maskY * strideMask + maskX * pixelSizeMask] * 0.2126 + maskBytes[maskY * strideMask + maskX * pixelSizeMask + 1] * 0.7152 + maskBytes[maskY * strideMask + maskX * pixelSizeMask + 2] * 0.0722) / 255.0;

                                if (mask.HasAlpha)
                                {
                                    weight *= maskBytes[maskY * strideMask + maskX * pixelSizeMask + 3] / 255.0;
                                }
                            }

                            if (weight > 0)
                            {
                                output[y * strideOutput + x * 4] = input[y * strideInput + x * 4];
                                output[y * strideOutput + x * 4 + 1] = input[y * strideInput + x * 4 + 1];
                                output[y * strideOutput + x * 4 + 2] = input[y * strideInput + x * 4 + 2];
                                output[y * strideOutput + x * 4 + 3] = (byte)Math.Round(input[y * strideInput + x * 4 + 3] * weight);
                            }
                            else
                            {
                                output[y * strideOutput + x * 4] = 0;
                                output[y * strideOutput + x * 4 + 1] = 0;
                                output[y * strideOutput + x * 4 + 2] = 0;
                                output[y * strideOutput + x * 4 + 3] = 0;
                            }
                        }
                    };
                }
                else
                {
                    yLoop = (y) =>
                    {
                        for (int x = 0; x < width; x++)
                        {
                            int maskX = (int)Math.Round((bounds.Location.X + (x + 0.5) * bounds.Size.Width / width - CachedBounds.Location.X) / CachedBounds.Size.Width * maskWidth);
                            int maskY = (int)Math.Round((bounds.Location.Y + (y + 0.5) * bounds.Size.Height / height - CachedBounds.Location.Y) / CachedBounds.Size.Height * maskHeight);

                            double weight = 0;

                            if (maskX >= 0 && maskX < maskWidth && maskY >= 0 && maskY < maskHeight)
                            {
                                weight = (maskBytes[maskY * strideMask + maskX * pixelSizeMask] * 0.2126 + maskBytes[maskY * strideMask + maskX * pixelSizeMask + 1] * 0.7152 + maskBytes[maskY * strideMask + maskX * pixelSizeMask + 2] * 0.0722);

                                if (mask.HasAlpha)
                                {
                                    weight *= maskBytes[maskY * strideMask + maskX * pixelSizeMask + 3] / 255.0;
                                }
                            }

                            if (weight > 0)
```

```
                            {
                                output[y * strideOutput + x * 4] = input[y * strideInput + x * 3];
                                output[y * strideOutput + x * 4 + 1] = input[y * strideInput + x * 3 + 1];
                                output[y * strideOutput + x * 4 + 2] = input[y * strideInput + x * 3 + 2];
                                output[y * strideOutput + x * 4 + 3] = (byte)Math.Round(weight);
                            }
                            else
                            {
                                output[y * strideOutput + x * 4] = 0;
                                output[y * strideOutput + x * 4 + 1] = 0;
                                output[y * strideOutput + x * 4 + 2] = 0;
                                output[y * strideOutput + x * 4 + 3] = 0;
                            }
                        }
                    };
                }

                if (threads == 1)
                {
                    for (int y = 0; y < height; y++)
                    {
                        yLoop(y);
                    }
                }
                else
                {
                    ParallelOptions options = new ParallelOptions() { MaxDegreeOfParallelism = threads };

                    Parallel.For(0, height, options, yLoop);
                }

            }

            DisposableIntPtr disp = new DisposableIntPtr(tbrData);

            return new RasterImage(ref disp, width, height, image.HasAlpha, image.Interpolate);

        }
    }
}
```

8.41 Font.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;

namespace VectSharp
{
    /// <summary>
    /// Represents a typeface with a specific size.
    /// </summary>
    public class Font
    {
        /// <summary>
        /// Determines whether text kerning is enabled. Note that, even when this is set to <see langword="false" />, text kerning will be applied on some platforms. For the best consistency, leave this set to <see langword="true"/>.
        /// </summary>
        public static bool EnableKerning = true;

        /// <summary>
```

```
        /// Represents options to underline text.
        /// </summary>
        public class FontUnderline
        {
            /// <summary>
            /// Determines whether the underline skips the parts of the glyph that would intersect with it or not.
            /// </summary>
            public bool SkipDescenders { get; set; }

            /// <summary>
            /// Determines the position of the top of the underline with respect to the text baseline. Positive values are below the baseline, negative values are above it. This is expressed as a fraction of the font size.
            /// </summary>
            public double Position { get; set; }

            /// <summary>
            /// Determines the thickness of the underline, expressed as a fraction of the font size.
            /// </summary>
            public double Thickness { get; set; }

            /// <summary>
            /// Determines the caps at the start and end of the underline.
            /// </summary>
            public LineCaps LineCap { get; set; }

            /// <summary>
            /// Determine whether the shape of the underline is slanted to follow the angle of italic fonts.
            /// </summary>
            public bool FollowItalicAngle { get; set; }

            /// <summary>
            /// Create a new underline with the default settings for the specified font family.
            /// </summary>
            /// <param name="family">The font family to which the underline will be applied.</param>
            internal FontUnderline(FontFamily family)
            {
                this.SkipDescenders = true;
                this.LineCap = LineCaps.Butt;
                this.FollowItalicAngle = true;

                if (family.TrueTypeFile != null)
                {
                    this.Position = -family.TrueTypeFile.Get1000EmUnderlinePosition() / 1000.0;
                    this.Thickness = family.TrueTypeFile.Get1000EmUnderlineThickness() / 1000.0;
                }
                else
                {
                    this.Position = 0.3;
                    this.Thickness = family.IsBold ? 0.15 : 0.075;
                }

                if (double.IsNaN(this.Position))
                {
                    this.Position = 0.3;
                }

                if (double.IsNaN(this.Thickness))
                {
                    this.Thickness = family.IsBold ? 0.15 : 0.075;
                }
            }
        }

        /// <summary>
        /// Represents detailed information about the metrics of a text string when drawn with a certain font.
        /// </summary>
        public class DetailedFontMetrics
        {
            /// <summary>
            /// Width of the text (measured on the actual glyph outlines).
            /// </summary>
            public double Width { get; }

            /// <summary>
            /// Height of the text (measured on the actual glyph outlines).
            /// </summary>
            public double Height { get; }

            /// <summary>
            /// How much the leftmost glyph in the string overhangs the glyph origin on the left. Positive for glyphs that hang past the origin (e.g. italic 'f').
            /// </summary>
            public double LeftSideBearing { get; }
```

```
            /// <summary>
            /// How much the rightmost glyph in the string overhangs the glyph end on the right. Positive for glyphs that hang past the end (e.g. italic 'f').
            /// </summary>
            public double RightSideBearing { get; }

            /// <summary>
            /// Height of the tallest glyph in the string over the baseline. Always &gt;= 0.
            /// </summary>
            public double Top { get; }

            /// <summary>
            /// Depth of the deepest glyph in the string below the baseline. Always &lt;= 0.
            /// </summary>
            public double Bottom { get; }

            /// <summary>
            /// Advance width of the text (excluding any left- or right- side bearing).
            /// </summary>
            public double AdvanceWidth { get; }

            internal DetailedFontMetrics(double width, double height, double leftSideBearing, double rightSideBearing, double top, double bottom, double advanceWidth)
            {
                this.Width = width;
                this.Height = height;
                this.LeftSideBearing = leftSideBearing;
                this.RightSideBearing = rightSideBearing;
                this.Top = top;
                this.Bottom = bottom;
                this.AdvanceWidth = advanceWidth;
            }
        }

        /// <summary>
        /// Font size, in graphics units.
        /// </summary>
        public double FontSize { get; }

        /// <summary>
        /// Font typeface.
        /// </summary>
        public FontFamily FontFamily { get; }

        /// <summary>
        /// Create a new Font object, given the base typeface and the font size.
        /// </summary>
        /// <param name="fontFamily">Base typeface. See <see cref="FontFamily"/>.</param>
        /// <param name="fontSize">The font size, in graphics units.</param>
        public Font(FontFamily fontFamily, double fontSize)
        {
            this.FontFamily = fontFamily;
            this.FontSize = fontSize;
        }

        /// <summary>
        /// Create a new Font object, given the base typeface, the font size, and a boolean value determining whether text using this font should be underlined.
        /// </summary>
        /// <param name="fontFamily">Base typeface. See <see cref="FontFamily"/>.</param>
        /// <param name="fontSize">The font size, in graphics units.</param>
        /// <param name="underlined">A boolean value determining whether text drawn using this font should be underlined. The appearance of the underline can be tweaked by changing the properties of the <see cref="Underline"/> property after the font has been created.</param>
        public Font(FontFamily fontFamily, double fontSize, bool underlined)
        {
            this.FontFamily = fontFamily;
            this.FontSize = fontSize;

            if (underlined)
            {
                this.Underline = new FontUnderline(fontFamily);
            }
        }

        /// <summary>
        /// Create a new Font object, given the base typeface, the font size, and an object describing the underline properties of text drawn using this font.
        /// </summary>
        /// <param name="fontFamily">Base typeface. See <see cref="FontFamily"/>.</param>
        /// <param name="fontSize">The font size, in graphics units.</param>
        /// <param name="underline">A <see cref="FontUnderline"/> object describing the underline properties of text drawn using this font.</param>
        public Font(FontFamily fontFamily, double fontSize, FontUnderline underline)
        {
            this.FontFamily = fontFamily;
```

```
            this.FontSize = fontSize;
            this.Underline = underline;
        }

        /// <summary>
        /// Maximum height over the baseline of the usual glyphs in the font (there may be glyphs taller than this). Always &gt;= 0.
        /// </summary>
        public double Ascent
        {
            get
            {
                if (this.FontFamily.TrueTypeFile == null)
                {
                    return 0;
                }
                else
                {
                    return this.FontFamily.TrueTypeFile.Get1000EmAscent() * this.FontSize / 1000;
                }
            }
        }

        /// <summary>
        /// Height above the baseline for a clipping region (Windows ascent). Always &gt;= 0.
        /// </summary>
        public double WinAscent
        {
            get
            {
                if (this.FontFamily.TrueTypeFile == null)
                {
                    return 0;
                }
                else
                {
                    return this.FontFamily.TrueTypeFile.Get1000EmWinAscent() * this.FontSize / 1000;
                }
            }
        }

        /// <summary>
        /// Maximum depth below the baseline of the usual glyphs in the font (there may be glyphs deeper than this). Always &lt;= 0.
        /// </summary>
        public double Descent
        {
            get
            {
                if (this.FontFamily.TrueTypeFile == null)
                {
                    return 0;
                }
                else
                {
                    return this.FontFamily.TrueTypeFile.Get1000EmDescent() * this.FontSize / 1000;
                }
            }
        }

        /// <summary>
        /// Absolute maximum height over the baseline of the glyphs in the font. Always &gt;= 0.
        /// </summary>
        public double YMax
        {
            get
            {
                if (this.FontFamily.TrueTypeFile == null)
                {
                    return 0;
                }
                else
                {
                    return this.FontFamily.TrueTypeFile.Get1000EmYMax() * this.FontSize / 1000;
                }
            }
        }

        /// <summary>
        /// Absolute maximum depth below the baseline of the glyphs in the font. Always &lt;= 0.
        /// </summary>
        public double YMin
        {
            get
            {
                if (this.FontFamily.TrueTypeFile == null)
                {
```

```
                    return 0;
                }
                else
                {
                    return this.FontFamily.TrueTypeFile.Get1000EmYMin() * this.FontSize / 1000;
                }
            }
        }

        /// <summary>
        /// Determines the underline style of text drawn using this font. If this is <see langword="null"/>, the text is not underlined.
        /// </summary>
        public FontUnderline Underline { get; }

        /// <summary>
        /// Measure the size of a text string when typeset with this font.
        /// </summary>
        /// <param name="text">The string to measure.</param>
        /// <returns>A <see cref="Size"/> object representing the width and height of the text.</returns>
        public Size MeasureText(string text)
        {
            if (this.FontFamily.TrueTypeFile != null)
            {
                double width = 0;
                double yMin = 0;
                double yMax = 0;

                Point currentGlyphPlacementDelta = new Point();
                Point currentGlyphAdvanceDelta = new Point();
                Point nextGlyphPlacementDelta = new Point();
                Point nextGlyphAdvanceDelta = new Point();

                for (int i = 0; i < text.Length; i++)
                {
                    if (Font.EnableKerning && i < text.Length - 1)
                    {
                        currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                        currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                        nextGlyphAdvanceDelta = new Point();
                        nextGlyphPlacementDelta = new Point();

                        TrueTypeFile.PairKerning kerning = this.FontFamily.TrueTypeFile.Get1000EmKerning(text[i], text[i + 1]);

                        if (kerning != null)
                        {
                            currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                            currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                            nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                            nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                        }
                    }

                    width += (this.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(text[i]) + currentGlyphAdvanceDelta.X) * this.FontSize / 1000;
                    TrueTypeFile.VerticalMetrics vMet = this.FontFamily.TrueTypeFile.Get1000EmGlyphVerticalMetrics(text[i]);

                    yMin = Math.Min(yMin, (vMet.YMin + currentGlyphPlacementDelta.Y) * this.FontSize / 1000);
                    yMax = Math.Max(yMax, (vMet.YMax + currentGlyphPlacementDelta.Y) * this.FontSize / 1000);
                }

                width -= this.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(text[0]).LeftSideBearing * this.FontSize / 1000;
                width -= this.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(text[text.Length - 1]).RightSideBearing * this.FontSize / 1000;

                return new Size(width, yMax - yMin);
            }
            else
            {
                return new Size(0, 0);
            }
        }

        /// <summary>
        /// Measure all the metrics of a text string when typeset with this font.
        /// </summary>
```

```
        /// <param name="text">The string to measure.</param>
        /// <returns>A <see cref="DetailedFontMetrics"/> object representing the metrics of the text.</returns>
        public DetailedFontMetrics MeasureTextAdvanced(string text)
        {
            if (this.FontFamily.TrueTypeFile != null)
            {
                double width = 0;
                double yMin = 0;
                double yMax = 0;

                Point currentGlyphPlacementDelta = new Point();
                Point currentGlyphAdvanceDelta = new Point();
                Point nextGlyphPlacementDelta = new Point();
                Point nextGlyphAdvanceDelta = new Point();

                for (int i = 0; i < text.Length; i++)
                {
                    if (Font.EnableKerning && i < text.Length - 1)
                    {
                        currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                        currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                        nextGlyphAdvanceDelta = new Point();
                        nextGlyphPlacementDelta = new Point();

                        TrueTypeFile.PairKerning kerning = this.FontFamily.TrueTypeFile.Get1000EmKerning(text[i], text[i + 1]);

                        if (kerning != null)
                        {
                            currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                            currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                            nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                            nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                        }
                    }

                    width += (this.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(text[i]) + currentGlyphAdvanceDelta.X) * this.FontSize / 1000;
                    TrueTypeFile.VerticalMetrics vMet = this.FontFamily.TrueTypeFile.Get1000EmGlyphVerticalMetrics(text[i]);

                    yMin = Math.Min(yMin, (vMet.YMin + currentGlyphPlacementDelta.Y) * this.FontSize / 1000);
                    yMax = Math.Max(yMax, (vMet.YMax + currentGlyphPlacementDelta.Y) * this.FontSize / 1000);
                }

                double lsb = this.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(text[0]).LeftSideBearing * this.FontSize / 1000;
                double rsb = this.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(text[text.Length - 1]).RightSideBearing * this.FontSize / 1000;

                double advanceWidth = width;

                width -= lsb;
                width -= rsb;

                return new DetailedFontMetrics(width, yMax - yMin, lsb, rsb, yMax, yMin, advanceWidth);
            }
            else
            {
                return new DetailedFontMetrics(0, 0, 0, 0, 0, 0, 0);
            }
        }
    }


    /// <summary>
    /// Represents a typeface.
    /// </summary>
    public class FontFamily
    {
        /// <summary>
        /// The default font library used to resolve font family names.
        /// </summary>
        public static IFontLibrary DefaultFontLibrary { get; set; } = new DefaultFontLibrary();

        /// <summary>
        /// Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised. Equivalent to
```

```
DefaultFontLibrary.ResolveFontFamily.
        /// </summary>
        /// <param name="fontFamily">The name of the font family to create, or the path to a TTF file.</param>
        /// <returns>If the font family name or the true type file is valid, a <see cref="FontFamily"/> object corresponding to the specified font family.</returns>
        public static FontFamily ResolveFontFamily(string fontFamily) => DefaultFontLibrary.ResolveFontFamily(fontFamily);


        /// <summary>
        /// Create a new font family from the specified standard font family name. Equivalent to DefaultFontLibrary.ResolveFontFamily.
        /// </summary>
        /// <param name="standardFontFamily">The standard name of the font family.</param>
        /// <returns>A <see cref="FontFamily"/> object corresponding to the specified font family.</returns>
        public static FontFamily ResolveFontFamily(StandardFontFamilies standardFontFamily) => DefaultFontLibrary.ResolveFontFamily(standardFontFamily);

        /// <summary>
        /// Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using
        /// the <paramref name="fallback"/>. If none of the fallback family names or true type files are valid, an exception might be raised. Equivalent to DefaultFontLibrary.ResolveFontFamily.
        /// </summary>
        /// <param name="fontFamily">The name of the font family to create, or the path to a TTF file.</param>
        /// <param name="fallback">Names of additional font families or TTF files, which will be tried if the first <paramref name="fontFamily"/> is not valid.</param>
        /// <returns>A <see cref="FontFamily"/> object corresponding to the first of the specified font families that is valid.</returns>
        public static FontFamily ResolveFontFamily(string fontFamily, params string[] fallback) => DefaultFontLibrary.ResolveFontFamily(fontFamily, fallback);

        /// <summary>
        /// Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using
        /// the <paramref name="fallback"/>. If none of the fallback family names or true type files are valid, instantiate a standard font family using the <paramref name="finalFallback"/>. Equivalent to DefaultFontLibrary.ResolveFontFamily.
        /// </summary>
        /// <param name="fontFamily">The name of the font family to create, or the path to a TTF file.</param>
        /// <param name="fallback">Names of additional font families or TTF files, which will be tried if the first <paramref name="fontFamily"/> is not valid.</param>
        /// <param name="finalFallback">The standard name of the font family that will be used if none of the fallback families are valid.</param>
        /// <returns>A <see cref="FontFamily"/> object corresponding to the first of the specified font families that is valid.</returns>
        public static FontFamily ResolveFontFamily(string fontFamily, StandardFontFamilies finalFallback, params string[] fallback) => DefaultFontLibrary.ResolveFontFamily(fontFamily, finalFallback, fallback);

        internal static object fontFamilyLock = new object();
        internal static readonly Dictionary<string, Stream> manifestResources = new Dictionary<string, Stream>();

        internal static Stream GetManifestResourceStream(string name)
        {
            if (!manifestResources.ContainsKey(name))
            {
                manifestResources.Add(name, typeof(FontFamily).Assembly.GetManifestResourceStream(name));
            }

            return manifestResources[name];
        }


        /// <summary>
        /// The names of the 14 standard families that are guaranteed to be displayed correctly.
        /// </summary>
        public static string[] StandardFamilies = new string[] { "Times-Roman", "Times-Bold", "Times-Italic", "Times-BoldItalic", "Helvetica", "Helvetica-Bold", "Helvetica-Oblique", "Helvetica-BoldOblique", "Courier", "Courier-Bold", "Courier-Oblique", "Courier-BoldOblique", "Symbol", "ZapfDingbats" };

        /// <summary>
        /// The names of the resource streams pointing to the included TrueType font files for each of the standard 14 font families.
        /// </summary>
        public static string[] StandardFontFamilyResources = new string[]
        {
            "VectSharp.StandardFonts.Tinos-Regular.ttf", "VectSharp.StandardFonts.Tinos-Bold.ttf", "VectSharp.StandardFonts.Tinos-Italic.ttf", "VectSharp.StandardFonts.Tinos-BoldItalic.ttf",
            "VectSharp.StandardFonts.Arimo-Regular.ttf", "VectSharp.StandardFonts.Arimo-Bold.ttf",
```

```
"VectSharp.StandardFonts.Arimo-Italic.ttf", "VectSharp.StandardFonts.Arimo-BoldItalic.ttf",
            "VectSharp.StandardFonts.Cousine-Regular.ttf", "VectSharp.StandardFonts.Cousine-Bold.ttf",
"VectSharp.StandardFonts.Cousine-Italic.ttf", "VectSharp.StandardFonts.Cousine-BoldItalic.ttf",
            "VectSharp.StandardFonts.SymbolNeu_GB.ttf",
"VectSharp.StandardFonts.Levibats-Regular_GB.ttf"
        };

        /// <summary>
        /// Whether this is one of the 14 standard font families or not.
        /// </summary>
        public bool IsStandardFamily { get; internal set; }

        /// <summary>
        /// The 14 standard font families.
        /// </summary>
        public enum StandardFontFamilies
        {
            /// <summary>
            /// Serif normal regular face.
            /// </summary>
            TimesRoman,

            /// <summary>
            /// Serif bold regular face.
            /// </summary>
            TimesBold,

            /// <summary>
            /// Serif normal italic face.
            /// </summary>
            TimesItalic,

            /// <summary>
            /// Serif bold italic face.
            /// </summary>
            TimesBoldItalic,

            /// <summary>
            /// Sans-serif normal regular face.
            /// </summary>
            Helvetica,

            /// <summary>
            /// Sans-serif bold regular face.
            /// </summary>
            HelveticaBold,

            /// <summary>
            /// Sans-serif normal oblique face.
            /// </summary>
            HelveticaOblique,

            /// <summary>
            /// Sans-serif bold oblique face.
            /// </summary>
            HelveticaBoldOblique,

            /// <summary>
            /// Monospace normal regular face.
            /// </summary>
            Courier,

            /// <summary>
            /// Monospace bold regular face.
            /// </summary>
            CourierBold,

            /// <summary>
            /// Monospace normal oblique face.
            /// </summary>
            CourierOblique,

            /// <summary>
            /// Monospace bold oblique face.
            /// </summary>
            CourierBoldOblique,

            /// <summary>
            /// Symbol font.
            /// </summary>
            Symbol,

            /// <summary>
            /// Dingbat font.
            /// </summary>
            ZapfDingbats
        }
```

```

        /// <summary>
        /// Full path to the TrueType font file for this font family (or, if this is a standard font family, name of the font family).
        /// </summary>
        public string FileName { get; internal set; }

        /// <summary>
        /// Name of the font family, including any variantes.
        /// </summary>
        public string FamilyName { get; internal set; }

        /// <summary>
        /// Parsed TrueType font file for this font family.
        /// See also: <seealso cref="VectSharp.TrueTypeFile"/>.
        /// </summary>
        public TrueTypeFile TrueTypeFile { get; }

        /// <summary>
        /// Whether this font is bold or not. This is set based on the information included in the OS/2 table of the TrueType file.
        /// </summary>
        public bool IsBold { get; internal set; }

        /// <summary>
        /// Whether this font is italic or oblique or not. This is set based on the information included in the OS/2 table of the TrueType file.
        /// </summary>
        public bool IsItalic { get; internal set; }

        /// <summary>
        /// Whether this font is oblique or not. This is set based on the information included in the OS/2 table of the TrueType file.
        /// </summary>
        public bool IsOblique { get; internal set; }

        /// <summary>
        /// Create a new <see cref="FontFamily"/>.
        /// </summary>
        /// <param name="fileName">The full path to the TrueType font file for this font family or the name of a standard font family.</param>
        [Obsolete("Please use the FontFamily.ResolveFontFamily(string) method instead!", true)]
        public FontFamily(string fileName)
        {
            lock (fontFamilyLock)
            {
                FontFamily resolved = DefaultFontLibrary.ResolveFontFamily(fileName);

                this.FileName = resolved.FileName;
                this.FamilyName = resolved.FamilyName;
                this.TrueTypeFile = resolved.TrueTypeFile;
                this.IsOblique = resolved.IsOblique;
                this.IsBold = resolved.IsBold;
                this.IsItalic = resolved.IsItalic;
                this.IsStandardFamily = resolved.IsStandardFamily;
            }
        }

        internal FontFamily()
        {

        }

        /// <summary>
        /// Create a new <see cref="FontFamily"/>.
        /// </summary>
        /// <param name="ttfStream">A stream containing a file in TTF format.</param>
        public FontFamily(Stream ttfStream)
        {
            lock (fontFamilyLock)
            {
                IsStandardFamily = false;

                TrueTypeFile = TrueTypeFile.CreateTrueTypeFile(ttfStream);

                FileName = TrueTypeFile.GetFontFamilyName();
                FamilyName = TrueTypeFile.GetFullFontFamilyName() ?? FileName;
                this.IsBold = TrueTypeFile.IsBold();
                this.IsItalic = TrueTypeFile.IsItalic();
                this.IsOblique = TrueTypeFile.IsOblique();
            }
        }

        /// <summary>
        /// Create a new <see cref="FontFamily"/>.
        /// </summary>
        /// <param name="ttf">A font file in TTF format.</param>
```

```
        public FontFamily(TrueTypeFile ttf)
        {
            lock (fontFamilyLock)
            {
                IsStandardFamily = false;

                TrueTypeFile = ttf;

                FileName = TrueTypeFile.GetFontFamilyName();
                FamilyName = TrueTypeFile.GetFullFontFamilyName() ?? FileName;
                this.IsBold = TrueTypeFile.IsBold();
                this.IsItalic = TrueTypeFile.IsItalic();
                this.IsOblique = TrueTypeFile.IsOblique();
            }
        }

        /// <summary>
        /// Create a new standard <see cref="FontFamily"/>.
        /// </summary>
        /// <param name="standardFontFamily">The standard font family.</param>
        [Obsolete("Please use the FontFamily.ResolveFontFamily(StandardFontFamilies) method instead!", true)]
        public FontFamily(StandardFontFamilies standardFontFamily)
        {
            lock (fontFamilyLock)
            {
                FontFamily resolved = DefaultFontLibrary.ResolveFontFamily(standardFontFamily);

                this.FileName = resolved.FileName;
                this.FamilyName = resolved.FamilyName;
                this.TrueTypeFile = resolved.TrueTypeFile;
                this.IsOblique = resolved.IsOblique;
                this.IsBold = resolved.IsBold;
                this.IsItalic = resolved.IsItalic;
                this.IsStandardFamily = resolved.IsStandardFamily;
            }
        }
    }

    /// <summary>
    /// Represents a FontFamily created from a resource stream.
    /// </summary>
    public class ResourceFontFamily : FontFamily
    {
        /// <summary>
        /// The name of the embedded resource, which will be parsed using <code>Avalonia.Media.FontFamily.Parse(string, Uri)</code>.
        /// </summary>
        public string ResourceName;

        /// <summary>
        /// Create a new <see cref="ResourceFontFamily"/> from the specified <paramref name="resourceStream"/> containing a TTF file, passing the specified <paramref name="resourceName"/> to the <code>Avalonia.Media.FontFamily.Parse(string, Uri)"</code> method.
        /// </summary>
        /// <param name="resourceStream">A resource stream containing a TTF file.</param>
        /// <param name="resourceName">The name of the embedded resource, which will be parsed using <code>Avalonia.Media.FontFamily.Parse(string, Uri)</code>.</param>
        public ResourceFontFamily(System.IO.Stream resourceStream, string resourceName) : base(resourceStream)
        {
            this.ResourceName = resourceName;
        }
    }

}
```

8.42 FontLibrary.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
```

```
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;
using System.IO;

namespace VectSharp
{
    /// <summary>
    /// Represents a font library with methods to create <see cref="FontFamily"/> objects from a string or from <see cref="FontFamily.StandardFontFamilies"/>.
    /// </summary>
    public interface IFontLibrary
    {
        /// <summary>
        /// Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, an exception might be raised.
        /// </summary>
        /// <param name="fontFamily">The name of the font family to create, or the path to a TTF file.</param>
        /// <returns>If the font family name or the true type file is valid, a <see cref="FontFamily"/> object corresponding to the specified font family.</returns>
        FontFamily ResolveFontFamily(string fontFamily);

        /// <summary>
        /// Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using
        /// the <paramref name="fallback"/>. If none of the fallback family names or true type files are valid, an exception might be raised.
        /// </summary>
        /// <param name="fontFamily">The name of the font family to create, or the path to a TTF file.</param>
        /// <param name="fallback">Names of additional font families or TTF files, which will be tried if the first <paramref name="fontFamily"/> is not valid.</param>
        /// <returns>A <see cref="FontFamily"/> object corresponding to the first of the specified font families that is valid.</returns>
        FontFamily ResolveFontFamily(string fontFamily, params string[] fallback);

        /// <summary>
        /// Create a new font family from the specified standard font family name.
        /// </summary>
        /// <param name="standardFontFamily">The standard name of the font family.</param>
        /// <returns>A <see cref="FontFamily"/> object corresponding to the specified font family.</returns>
        FontFamily ResolveFontFamily(FontFamily.StandardFontFamilies standardFontFamily);

        /// <summary>
        /// Create a new font family from the specified family name or true type file. If the family name or the true type file are not valid, try to instantiate the font family using
        /// the <paramref name="fallback"/>. If none of the fallback family names or true type files are valid, instantiate a standard font family using the <paramref name="finalFallback"/>.
        /// </summary>
        /// <param name="fontFamily">The name of the font family to create, or the path to a TTF file.</param>
        /// <param name="fallback">Names of additional font families or TTF files, which will be tried if the first <paramref name="fontFamily"/> is not valid.</param>
        /// <param name="finalFallback">The standard name of the font family that will be used if none of the fallback families are valid.</param>
        /// <returns>A <see cref="FontFamily"/> object corresponding to the first of the specified font families that is valid.</returns>
        FontFamily ResolveFontFamily(string fontFamily, FontFamily.StandardFontFamilies finalFallback, params string[] fallback);
    }

    /// <summary>
    /// Abstract class with a default implementation of font family fallbacks.
    /// </summary>
    public abstract class FontLibrary : IFontLibrary
    {
        /// <inheritdoc/>
        public abstract FontFamily ResolveFontFamily(string fontFamily);

        /// <inheritdoc/>
        public abstract FontFamily ResolveFontFamily(FontFamily.StandardFontFamilies standardFontFamily);

        /// <inheritdoc/>
        public virtual FontFamily ResolveFontFamily(string fontFamily, params string[] fallback)
        {
            bool found = false;

            FontFamily tbr = null;

            try
```

```
            {
                tbr = ResolveFontFamily(fontFamily);
                if (tbr.TrueTypeFile != null)
                {
                    found = true;
                }
            }
            catch
            {
                tbr = null;
                found = false;
            }

            if (!found)
            {
                for (int i = 0; i < fallback.Length; i++)
                {
                    try
                    {
                        tbr = ResolveFontFamily(fontFamily);
                        if (tbr.TrueTypeFile != null)
                        {
                            found = true;
                        }
                    }
                    catch
                    {
                        tbr = null;
                        found = false;
                    }

                    if (found)
                    {
                        return tbr;
                    }
                }
            }
            else
            {
                return tbr;
            }

            throw new FontFamilyCreationException(fontFamily);
        }

        /// <inheritdoc/>
        public virtual FontFamily ResolveFontFamily(string fontFamily, FontFamily.StandardFontFamilies finalFallback, params string[] fallback)
        {
            bool found = false;

            FontFamily tbr = null;

            try
            {
                tbr = ResolveFontFamily(fontFamily);
                if (tbr.TrueTypeFile != null)
                {
                    found = true;
                }
            }
            catch
            {
                tbr = null;
                found = false;
            }

            if (!found)
            {
                for (int i = 0; i < fallback.Length; i++)
                {
                    try
                    {
                        tbr = ResolveFontFamily(fontFamily);
                        if (tbr.TrueTypeFile != null)
                        {
                            found = true;
                        }
                    }
                    catch
                    {
                        tbr = null;
                        found = false;
                    }

                    if (found)
                    {
```

```
                        return tbr;
                    }
                }
            }
            else
            {
                return tbr;
            }

            return ResolveFontFamily(finalFallback);
        }
    }

    /// <summary>
    /// A font library that can be used to cache and resolve font family names.
    /// </summary>
    public class SimpleFontLibrary : FontLibrary
    {
        private Dictionary<string, string> KnownFonts = new Dictionary<string, string>();
        private Dictionary<string, string> NotLoadedFonts = new Dictionary<string, string>();
        private Dictionary<string, FontFamily> LoadedFonts = new Dictionary<string, FontFamily>();
        private Dictionary<FontFamily.StandardFontFamilies, FontFamily> Fallbacks = new Dictionary<FontFamily.StandardFontFamilies, FontFamily>();
        private DefaultFontLibrary defaultLibrary = new DefaultFontLibrary();

        /// <summary>
        /// Create a new <see cref="SimpleFontLibrary"/> instance.
        /// </summary>
        /// <param name="standardFontLibrary">An existing font library that will be used to resolve the standard font families.</param>
        public SimpleFontLibrary(IFontLibrary standardFontLibrary)
        {
            for (int i = 0; i < 14; i++)
            {
                FontFamily.StandardFontFamilies stdFF = (FontFamily.StandardFontFamilies)i;
                FontFamily resolved = standardFontLibrary.ResolveFontFamily(stdFF);

                Fallbacks[stdFF] = resolved;
                LoadedFonts[FontFamily.StandardFamilies[i]] = resolved;
                KnownFonts[FontFamily.StandardFamilies[i]] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.FamilyName] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.TrueTypeFile.GetFontFamilyName()] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.TrueTypeFile.GetFullFontFamilyName()] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.TrueTypeFile.GetFontName()] = FontFamily.StandardFamilies[i];
            }
        }

        /// <summary>
        /// Create a new <see cref="SimpleFontLibrary"/> instance, using the default font library to resolve the standard font families.
        /// </summary>
        public SimpleFontLibrary() : this(FontFamily.DefaultFontLibrary)
        {

        }

        /// <summary>
        /// Create a new <see cref="SimpleFontLibrary"/> instance, with the specified replacements for the standard font families.
        /// </summary>
        /// <param name="timesRoman">The font family to use for the Times-Roman standard font.</param>
        /// <param name="timesBold">The font family to use for the Times-Bold standard font.</param>
        /// <param name="timesItalic">The font family to use for the Times-Italic standard font.</param>
        /// <param name="timesBoldItalic">The font family to use for the Times-BoldItalic standard font.</param>
        /// <param name="helvetica">The font family to use for the Helvetica standard font.</param>
        /// <param name="helveticaBold">The font family to use for the Helvetica-Bold standard font.</param>
        /// <param name="helveticaOblique">The font family to use for the Helvetica-Oblique standard font.</param>
        /// <param name="helveticaBoldOblique">The font family to use for the Helvetica-BoldOblique standard font.</param>
        /// <param name="courier">The font family to use for the Courier standard font.</param>
        /// <param name="courierBold">The font family to use for the Courier-Bold standard font.</param>
        /// <param name="courierOblique">The font family to use for the Courier-Oblique standard font.</param>
        /// <param name="courierBoldOblique">The font family to use for the Courier-BoldOblique standard font.</param>
        /// <param name="symbol">The font family to use for the Symbol standard font.</param>
        /// <param name="zapfdingbats">The font family to use for the Zapfdingbats standard font.</param>
        public SimpleFontLibrary(FontFamily timesRoman, FontFamily timesBold, FontFamily timesItalic, FontFamily timesBoldItalic,
```

```
            FontFamily helvetica, FontFamily helveticaBold, FontFamily helveticaOblique, FontFamily helveticaBoldOblique,
            FontFamily courier, FontFamily courierBold, FontFamily courierOblique, FontFamily courierBoldOblique,
            FontFamily symbol, FontFamily zapfdingbats)
        {
            Fallbacks[FontFamily.StandardFontFamilies.TimesRoman] = timesRoman;
            Fallbacks[FontFamily.StandardFontFamilies.TimesBold] = timesBold;
            Fallbacks[FontFamily.StandardFontFamilies.TimesItalic] = timesItalic;
            Fallbacks[FontFamily.StandardFontFamilies.TimesBoldItalic] = timesBoldItalic;

            Fallbacks[FontFamily.StandardFontFamilies.Helvetica] = helvetica;
            Fallbacks[FontFamily.StandardFontFamilies.HelveticaBold] = helveticaBold;
            Fallbacks[FontFamily.StandardFontFamilies.HelveticaOblique] = helveticaOblique;
            Fallbacks[FontFamily.StandardFontFamilies.HelveticaBoldOblique] = helveticaBoldOblique;

            Fallbacks[FontFamily.StandardFontFamilies.Courier] = courier;
            Fallbacks[FontFamily.StandardFontFamilies.CourierBold] = courierBold;
            Fallbacks[FontFamily.StandardFontFamilies.CourierOblique] = courierOblique;
            Fallbacks[FontFamily.StandardFontFamilies.CourierBoldOblique] = courierBoldOblique;

            Fallbacks[FontFamily.StandardFontFamilies.Symbol] = symbol;
            Fallbacks[FontFamily.StandardFontFamilies.ZapfDingbats] = zapfdingbats;

            for (int i = 0; i < 14; i++)
            {
                FontFamily.StandardFontFamilies stdFF = (FontFamily.StandardFontFamilies)i;
                FontFamily resolved = Fallbacks[stdFF];
                resolved.IsStandardFamily = true;
                resolved.FileName = FontFamily.StandardFamilies[i];
                resolved.FamilyName = FontFamily.StandardFamilies[i].Replace("-", " ");

                LoadedFonts[FontFamily.StandardFamilies[i]] = resolved;
                KnownFonts[FontFamily.StandardFamilies[i]] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.FamilyName] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.TrueTypeFile.GetFontFamilyName()] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.TrueTypeFile.GetFullFontFamilyName()] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.TrueTypeFile.GetFontName()] = FontFamily.StandardFamilies[i];
            }
        }


        /// <summary>
        /// Create a new <see cref="SimpleFontLibrary"/> instance, with the specified replacements for the standard font families.
        /// </summary>
        /// <param name="timesRoman">The font family to use for the Times-Roman standard font.</param>
        /// <param name="timesBold">The font family to use for the Times-Bold standard font.</param>
        /// <param name="timesItalic">The font family to use for the Times-Italic standard font.</param>
        /// <param name="timesBoldItalic">The font family to use for the Times-BoldItalic standard font.</param>
        /// <param name="helvetica">The font family to use for the Helvetica standard font.</param>
        /// <param name="helveticaBold">The font family to use for the Helvetica-Bold standard font.</param>
        /// <param name="helveticaOblique">The font family to use for the Helvetica-Oblique standard font.</param>
        /// <param name="helveticaBoldOblique">The font family to use for the Helvetica-BoldOblique standard font.</param>
        /// <param name="courier">The font family to use for the Courier standard font.</param>
        /// <param name="courierBold">The font family to use for the Courier-Bold standard font.</param>
        /// <param name="courierOblique">The font family to use for the Courier-Oblique standard font.</param>
        /// <param name="courierBoldOblique">The font family to use for the Courier-BoldOblique standard font.</param>
        /// <param name="symbol">The font family to use for the Symbol standard font.</param>
        /// <param name="zapfdingbats">The font family to use for the Zapfdingbats standard font.</param>
        public SimpleFontLibrary(string timesRoman, string timesBold, string timesItalic, string timesBoldItalic,
            string helvetica, string helveticaBold, string helveticaOblique, string helveticaBoldOblique,
            string courier, string courierBold, string courierOblique, string courierBoldOblique,
            string symbol, string zapfdingbats)
        {
            Fallbacks[FontFamily.StandardFontFamilies.TimesRoman] = FontFamily.DefaultFontLibrary.ResolveFontFamily(timesRoman);
            Fallbacks[FontFamily.StandardFontFamilies.TimesBold]= FontFamily.DefaultFontLibrary.ResolveFontFamily(timesBold);
            Fallbacks[FontFamily.StandardFontFamilies.TimesItalic]= FontFamily.DefaultFontLibrary.ResolveFontFamily(timesItalic);
            Fallbacks[FontFamily.StandardFontFamilies.TimesBoldItalic]= FontFamily.DefaultFontLibrary.ResolveFontFamily(timesBoldItalic);

```

```
            Fallbacks[FontFamily.StandardFontFamilies.Helvetica]= FontFamily.DefaultFontLibrary.ResolveFontFamily(helvetica);
            Fallbacks[FontFamily.StandardFontFamilies.HelveticaBold]= FontFamily.DefaultFontLibrary.ResolveFontFamily(helveticaBold);
            Fallbacks[FontFamily.StandardFontFamilies.HelveticaOblique]= FontFamily.DefaultFontLibrary.ResolveFontFamily(helveticaOblique);
            Fallbacks[FontFamily.StandardFontFamilies.HelveticaBoldOblique]= FontFamily.DefaultFontLibrary.ResolveFontFamily(helveticaBoldOblique);

            Fallbacks[FontFamily.StandardFontFamilies.Courier]= FontFamily.DefaultFontLibrary.ResolveFontFamily(courier);
            Fallbacks[FontFamily.StandardFontFamilies.CourierBold]= FontFamily.DefaultFontLibrary.ResolveFontFamily(courierBold);
            Fallbacks[FontFamily.StandardFontFamilies.CourierOblique]= FontFamily.DefaultFontLibrary.ResolveFontFamily(courierOblique);
            Fallbacks[FontFamily.StandardFontFamilies.CourierBoldOblique]= FontFamily.DefaultFontLibrary.ResolveFontFamily(courierBoldOblique);

            Fallbacks[FontFamily.StandardFontFamilies.Symbol]= FontFamily.DefaultFontLibrary.ResolveFontFamily(symbol);
            Fallbacks[FontFamily.StandardFontFamilies.ZapfDingbats]= FontFamily.DefaultFontLibrary.ResolveFontFamily(zapfdingbats);

            for (int i = 0; i < 14; i++)
            {
                FontFamily.StandardFontFamilies stdFF = (FontFamily.StandardFontFamilies)i;
                FontFamily resolved = Fallbacks[stdFF];
                resolved.IsStandardFamily = true;
                resolved.FileName = FontFamily.StandardFamilies[i];
                resolved.FamilyName = FontFamily.StandardFamilies[i].Replace("-", " ");

                LoadedFonts[FontFamily.StandardFamilies[i]] = resolved;
                KnownFonts[FontFamily.StandardFamilies[i]] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.FamilyName] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.TrueTypeFile.GetFontFamilyName()] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.TrueTypeFile.GetFullFontFamilyName()] = FontFamily.StandardFamilies[i];
                KnownFonts[resolved.TrueTypeFile.GetFontName()] = FontFamily.StandardFamilies[i];
            }
        }


        /// <summary>
        /// Add the specified font family to the library with the specified name.
        /// </summary>
        /// <param name="fontFamilyName">The name of the font family.</param>
        /// <param name="fontFamily">The font family to add.</param>
        public void Add(string fontFamilyName, FontFamily fontFamily)
        {
            this.LoadedFonts[fontFamilyName] = fontFamily;
            this.KnownFonts[fontFamilyName] = fontFamilyName;

            if (fontFamily.TrueTypeFile != null)
            {
                this.KnownFonts[fontFamily.TrueTypeFile.GetFontFamilyName()] = fontFamilyName;
                this.KnownFonts[fontFamily.TrueTypeFile.GetFullFontFamilyName()] = fontFamilyName;
                this.KnownFonts[fontFamily.TrueTypeFile.GetFontName()] = fontFamilyName;
            }
        }

        /// <summary>
        /// Add the specified font family to the library.
        /// </summary>
        /// <param name="fontFamily">The font family to add.</param>
        public void Add(FontFamily fontFamily)
        {
            if (fontFamily.TrueTypeFile != null)
            {
                this.Add(fontFamily.TrueTypeFile.GetFullFontFamilyName(), fontFamily);
            }
        }

        /// <summary>
        /// Add the font family contained in the specified True Type Font file to the library.
        /// </summary>
        /// <param name="fileName">The path to the TTF file containing the font family.</param>
        public void Add(string fileName)
        {
            FontFamily fontFamily = FontFamily.DefaultFontLibrary.ResolveFontFamily(fileName);

            if (fontFamily.TrueTypeFile != null)
            {
                this.Add(fontFamily.TrueTypeFile.GetFullFontFamilyName(), fontFamily);
            }
        }

```

```
        /// <summary>
        /// Add the font family contained in the specified True Type Font file to the library, with the specified name. The font family is not loaded until it is requested for the first time.
        /// </summary>
        /// <param name="fontFamily">The name of the font family.</param>
        /// <param name="fileName">The path to the TTF file containing the font family.</param>
        public void Add(string fontFamily, string fileName)
        {
            this.KnownFonts[fontFamily] = fontFamily;
            this.NotLoadedFonts[fontFamily] = fileName;
        }

        /// <inheritdoc/>
        public override FontFamily ResolveFontFamily(FontFamily.StandardFontFamilies standardFontFamily)
        {
            return Fallbacks[standardFontFamily];
        }

        /// <inheritdoc/>
        public override FontFamily ResolveFontFamily(string fontFamily)
        {
            if (KnownFonts.TryGetValue(fontFamily, out string knownFontName))
            {
                if (LoadedFonts.TryGetValue(knownFontName, out FontFamily tbr))
                {
                    return tbr;
                }
                else
                {
                    if (NotLoadedFonts.TryGetValue(knownFontName, out string ttfFile))
                    {
                        tbr = defaultLibrary.ResolveFontFamily(ttfFile);

                        if (tbr.TrueTypeFile != null)
                        {
                            string familyName = tbr.TrueTypeFile.GetFontFamilyName();

                            this.LoadedFonts[familyName] = tbr;
                            this.KnownFonts[familyName] = familyName;
                            this.KnownFonts[tbr.TrueTypeFile.GetFontName()] = familyName;
                        }

                        return tbr;
                    }
                    else
                    {
                        return defaultLibrary.ResolveFontFamily(fontFamily);
                    }
                }
            }
            else
            {
                return defaultLibrary.ResolveFontFamily(fontFamily);
            }
        }
    }

    /// <summary>
    /// An exception that occurs while creating a <see cref="FontFamily"/>.
    /// </summary>
    public class FontFamilyCreationException : Exception
    {
        /// <summary>
        /// The name of the font family that was being created.
        /// </summary>
        public string FontFamily { get; }

        /// <summary>
        /// Create a new <see cref="FontFamilyCreationException"/> instance.
        /// </summary>
        /// <param name="fontFamily">The name of the font family that was being created.</param>
        public FontFamilyCreationException(string fontFamily) : base("The font family \"" + fontFamily + "\" could not be created!")
        {
            this.FontFamily = fontFamily;
        }
    }

    /// <summary>
    /// A default font library that resolves standard families using the embedded fonts.
    /// </summary>
    public class DefaultFontLibrary : FontLibrary
    {
        /// <inheritdoc/>
        public override FontFamily ResolveFontFamily(string fontFamily)
        {
```

```
            lock (FontFamily.fontFamilyLock)
            {
                bool isStandardFamily;

                if (FontFamily.StandardFamilies.Contains(fontFamily) || FontFamily.StandardFamilies.Contains(fontFamily.Replace(" ", "-")))
                {
                    isStandardFamily = true;
                }
                else
                {
                    isStandardFamily = false;
                }

                if (isStandardFamily)
                {
                    fontFamily = fontFamily.Replace(" ", "-");
                    Stream ttfStream = FontFamily.GetManifestResourceStream(FontFamily.StandardFontFamilyResources[Array.IndexOf(FontFamily.StandardFamilies, fontFamily)]);

                    FontFamily tbr = new FontFamily(ttfStream);
                    tbr.IsStandardFamily = true;
                    tbr.FileName = fontFamily;
                    tbr.FamilyName = tbr.FileName.Replace("-", " ");

                    if (fontFamily == "Times-Italic" || fontFamily == "Times-BoldItalic" || fontFamily == "Helvetica-Oblique" || fontFamily == "Helvetica-BoldOblique" || fontFamily == "Courier-Oblique" || fontFamily == "Courier-BoldOblique")
                    {
                        tbr.IsItalic = true;
                        tbr.IsOblique = (fontFamily == "Courier-Oblique" || fontFamily == "Courier-BoldOblique");
                    }
                    else
                    {
                        tbr.IsItalic = false;
                        tbr.IsOblique = false;
                    }

                    return tbr;
                }
                else
                {
                    try
                    {
                        FontFamily tbr = new FontFamily(TrueTypeFile.CreateTrueTypeFile(fontFamily));
                        tbr.FileName = fontFamily;
                        tbr.FamilyName = tbr.TrueTypeFile?.GetFullFontFamilyName() ?? tbr.FileName;
                        return tbr;
                    }
                    catch
                    {
                        FontFamily tbr = new FontFamily();
                        tbr.FileName = fontFamily;
                        tbr.FamilyName = fontFamily;
                        return tbr;
                    }
                }
            }
        }

        /// <inheritdoc/>
        public override FontFamily ResolveFontFamily(FontFamily.StandardFontFamilies standardFontFamily)
        {
            lock (FontFamily.fontFamilyLock)
            {
                Stream ttfStream = FontFamily.GetManifestResourceStream(FontFamily.StandardFontFamilyResources[(int)standardFontFamily]);
                FontFamily tbr = new FontFamily(ttfStream);


                tbr.IsStandardFamily = true;

                tbr.FileName = FontFamily.StandardFamilies[(int)standardFontFamily];
                tbr.FamilyName = tbr.FileName.Replace("-", " ");

                if (tbr.FileName == "Times-Italic" || tbr.FileName == "Times-BoldItalic" || tbr.FileName == "Helvetica-Oblique" || tbr.FileName == "Helvetica-BoldOblique" || tbr.FileName == "Courier-Oblique" || tbr.FileName == "Courier-BoldOblique")
                {
                    tbr.IsItalic = true;
                    tbr.IsOblique = (tbr.FileName == "Courier-Oblique" || tbr.FileName == "Courier-BoldOblique");
                }
                else
```

```
                {
                    tbr.IsItalic = false;
                    tbr.IsOblique = false;
                }

                return tbr;
            }
        }
    }
}
```

8.43 FormattedText.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;
using static System.Net.Mime.MediaTypeNames;

namespace VectSharp
{
    /// <summary>
    /// Represents the position of the text.
    /// </summary>
    public enum Script
    {
        /// <summary>
        /// The text is normal text.
        /// </summary>
        Normal,

        /// <summary>
        /// The text is a superscript.
        /// </summary>
        Superscript,

        /// <summary>
        /// The text is a subscript.
        /// </summary>
        Subscript
    }

    /// <summary>
    /// Represents a run of text that should be drawn with the same style.
    /// </summary>
    public class FormattedText
    {
        /// <summary>
        /// Represents the text represented by this instance.
        /// </summary>
        public string Text { get; }

        /// <summary>
        /// Represents the font that should be used to draw the text.
        /// </summary>
        public Font Font { get; }

        /// <summary>
        /// Represents the position of the text.
        /// </summary>
        public Script Script { get; }

        /// <summary>
        /// Represents the brush that should be used to draw the text. If this is null, the default brush is used.
        /// </summary>
```

```
        public Brush Brush { get; }

        /// <summary>
        /// Creates a new <see cref="FormattedText"/> instance with the specified <paramref name="text"/>, <paramref name="font"/>, <paramref name="script"/> position and <paramref name="brush"/>.
        /// </summary>
        /// <param name="text">The text that will be contained in the new <see cref="FormattedText"/>.</param>
        /// <param name="font">The font that will be used by the new <see cref="FormattedText"/>.</param>
        /// <param name="script">The script position of the new <see cref="FormattedText"/>.</param>
        /// <param name="brush">The brush that will be used by the new <see cref="FormattedText"/>.</param>
        public FormattedText(string text, Font font, Script script = Script.Normal, Brush brush = null)
        {
            this.Text = text;
            this.Font = font;
            this.Script = script;
            this.Brush = brush;
        }

        /// <summary>
        /// Parse the formatting information contained in a text string into a collection of <see cref="FormattedText"/> objects.
        /// </summary>
        /// <param name="text">
        /// The string containing formatting information. Format information is specified using HTML-like tags:
        /// <list type="bullet">
        /// <item><c>&lt;b&gt;&lt;/b&gt;</c> or <c>&lt;strong&gt;&lt;/strong&gt;</c> are used for bold text;</item>
        /// <item><c>&lt;i&gt;&lt;/i&gt;</c> or <c>&lt;em&gt;&lt;/em&gt;</c> are used for text in italics;</item>
        /// <item><c>&lt;u&gt;&lt;/u&gt;</c></item> are used for underlined text;
        /// <item><c>&lt;sup&gt;&lt;/sup&gt;</c> and <c>&lt;sub&gt;&lt;/sub&gt;</c> are used, respectively, for superscript and subscript text;</item>
        /// <item><c>&lt;#COLOUR&gt;&lt;/#&gt;</c> is used to specify the colour of the text, where <c>COLOUR</c> is a CSS colour string (e.g. <c>&lt;#red&gt;</c>, <c>&lt;#0080FF&gt;</c>, or <c>&lt;#rgba(128, 80, 52, 0.5)&gt;</c>).</item>
        /// </list></param>
        /// <param name="normalFont">The font that will be used for text that is neither bold nor italic.</param>
        /// <param name="boldFont">The font that will be used for text that is bold. Note that this does not necessarily have to be a bold font; this is just the font that is applied to text contained within <c>&lt;b&gt;&lt;/b&gt;</c> tags.</param>
        /// <param name="italicFont">The font that will be used for text that is in italics. Note that this does not necessarily have to be an italic font; this is just the font that is applied to text contained within <c>&lt;i&gt;&lt;/i&gt;</c> tags.</param>
        /// <param name="boldItalicFont">The font that will be used for text that is both in bold and in italics.</param>
        /// <param name="defaultBrush">The default <see cref="Brush"/> that will be used for text runs that do not specify a colour. If this is <see langword="null"/>, the default <see cref="Brush"/> will be the one specified in the painting call.</param>
        /// <returns>A lazy collection of <see cref="FormattedText"/> objects. Note that every enumeration of this collection causes the text to be parsed again; if you need to enumerate this collection more than once, you should probably convert it e.g. to a <see cref="List{T}"/>.</returns>
        public static IEnumerable<FormattedText> Format(string text, Font normalFont, Font boldFont, Font italicFont, Font boldItalicFont, Brush defaultBrush = null)
        {
            StringBuilder currentRun = new StringBuilder();
            int boldDepth = 0;
            int italicsDepth = 0;
            int underlineDepth = 0;
            int superscriptDepth = 0;
            int subscriptDepth = 0;
            Stack<Brush> brushes = new Stack<Brush>();
            brushes.Push(defaultBrush);

            for (int i = 0; i < text.Length; i++)
            {
                if (text[i] != '<')
                {
                    currentRun.Append(text[i]);
                }
                else
                {
                    Tags tag = GetTag(text, i, out int tagEnd, out Brush tagBrush);

                    if (tag == Tags.None)
                    {
                        currentRun.Append(text[i]);
                    }
                    else
                    {
                        i = tagEnd;
```

```
                        string txt = currentRun.ToString();

                        switch (tag)
                        {
                            case Tags.BoldOpen:
                                if (!string.IsNullOrEmpty(txt))
                                {
                                    yield return new FormattedText(txt, GetFont(boldDepth % 2 == 1, italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont), superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ? Script.Subscript : Script.Normal, brushes.Peek());
                                }
                                currentRun.Clear();
                                boldDepth++;
                                break;
                            case Tags.BoldClose:
                                if (boldDepth > 0)
                                {
                                    if (!string.IsNullOrEmpty(txt))
                                    {
                                        yield return new FormattedText(txt, GetFont(boldDepth % 2 == 1, italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont), superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ? Script.Subscript : Script.Normal, brushes.Peek());
                                    }
                                    currentRun.Clear();
                                    boldDepth--;
                                }
                                break;
                            case Tags.UnderlineOpen:
                                if (!string.IsNullOrEmpty(txt))
                                {
                                    yield return new FormattedText(txt, GetFont(boldDepth % 2 == 1, italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont), superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ? Script.Subscript : Script.Normal, brushes.Peek());
                                }
                                currentRun.Clear();
                                underlineDepth++;
                                break;
                            case Tags.UnderlineClose:
                                if (underlineDepth > 0)
                                {
                                    if (!string.IsNullOrEmpty(txt))
                                    {
                                        yield return new FormattedText(txt, GetFont(boldDepth % 2 == 1, italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont), superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ? Script.Subscript : Script.Normal, brushes.Peek());
                                    }
                                    currentRun.Clear();
                                    underlineDepth--;
                                }
                                break;

                            case Tags.ItalicsOpen:
                                if (!string.IsNullOrEmpty(txt))
                                {
                                    yield return new FormattedText(txt, GetFont(boldDepth % 2 == 1, italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont), superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ? Script.Subscript : Script.Normal, brushes.Peek());
                                }
                                currentRun.Clear();
                                italicsDepth++;
                                break;
                            case Tags.ItalicsClose:
                                if (italicsDepth > 0)
                                {
                                    if (!string.IsNullOrEmpty(txt))
                                    {
                                        yield return new FormattedText(txt, GetFont(boldDepth % 2 == 1, italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont), superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ? Script.Subscript : Script.Normal, brushes.Peek());
                                    }
                                    currentRun.Clear();
                                    italicsDepth--;
                                }
                                break;

                            case Tags.SupOpen:
                                if (!string.IsNullOrEmpty(txt))
                                {
                                    yield return new FormattedText(txt, GetFont(boldDepth % 2 == 1, italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont),
```

```
superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ?
Script.Subscript : Script.Normal, brushes.Peek());
                                }
                                currentRun.Clear();
                                superscriptDepth++;
                                break;
                            case Tags.SupClose:
                                if (superscriptDepth > 0)
                                {
                                    if (!string.IsNullOrEmpty(txt))
                                    {
                                        yield return new FormattedText(txt, GetFont(boldDepth % 2 ==
1, italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont),
superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ?
Script.Subscript : Script.Normal, brushes.Peek());
                                    }
                                    currentRun.Clear();
                                    superscriptDepth--;
                                }
                                break;

                            case Tags.SubOpen:
                                if (!string.IsNullOrEmpty(txt))
                                {
                                    yield return new FormattedText(txt, GetFont(boldDepth % 2 == 1,
italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont),
superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ?
Script.Subscript : Script.Normal, brushes.Peek());
                                }
                                currentRun.Clear();
                                subscriptDepth++;
                                break;
                            case Tags.SubClose:
                                if (subscriptDepth > 0)
                                {
                                    if (!string.IsNullOrEmpty(txt))
                                    {
                                        yield return new FormattedText(txt, GetFont(boldDepth % 2 ==
1, italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont),
superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ?
Script.Subscript : Script.Normal, brushes.Peek());
                                    }
                                    currentRun.Clear();
                                    subscriptDepth--;
                                }
                                break;
                            case Tags.ColourOpen:
                                if (!string.IsNullOrEmpty(txt))
                                {
                                    yield return new FormattedText(txt, GetFont(boldDepth % 2 == 1,
italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont),
superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ?
Script.Subscript : Script.Normal, brushes.Peek());
                                }
                                currentRun.Clear();
                                brushes.Push(tagBrush);
                                break;
                            case Tags.ColourClose:
                                if (brushes.Count > 1)
                                {
                                    if (!string.IsNullOrEmpty(txt))
                                    {
                                        yield return new FormattedText(txt, GetFont(boldDepth % 2 ==
1, italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont),
superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ?
Script.Subscript : Script.Normal, brushes.Peek());
                                    }
                                    currentRun.Clear();
                                    brushes.Pop();
                                }
                                break;
                            }
                        }
                    }
            }

            if (currentRun.Length > 0)
            {
                yield return new FormattedText(currentRun.ToString(), GetFont(boldDepth % 2 == 1,
italicsDepth % 2 == 1, underlineDepth % 2 == 1, normalFont, boldFont, italicFont, boldItalicFont),
superscriptDepth > subscriptDepth ? Script.Superscript : subscriptDepth > superscriptDepth ?
Script.Subscript : Script.Normal, brushes.Peek());
            }
        }

        /// <summary>
        /// Parse the formatting information contained in a text string into a collection of <see
```

```
cref="FormattedText"/> objects, using fonts from a standard font family.
        /// </summary>
        /// <param name="text">The string containing formatting information. Format information is specified using HTML-like tags:
        /// <list type="bullet">
        /// <item><c>&lt;b&gt;&lt;/b&gt;</c> or <c>&lt;strong&gt;&lt;/strong&gt;</c> are used for bold text;</item>
        /// <item><c>&lt;i&gt;&lt;/i&gt;</c> or <c>&lt;em&gt;&lt;/em&gt;</c> are used for text in italics;</item>
        /// <item><c>&lt;u&gt;&lt;/u&gt;</c></item> are used for underlined text;
        /// <item><c>&lt;sup&gt;&lt;/sup&gt;</c> and <c>&lt;sub&gt;&lt;/sub&gt;</c> are used, respectively, for superscript and subscript text;</item>
        /// <item><c>&lt;#COLOUR&gt;&lt;/#&gt;</c> is used to specify the colour of the text, where <c>COLOUR</c> is a CSS colour string (e.g. <c>&lt;#red&gt;</c>, <c>&lt;#0080FF&gt;</c>, or <c>&lt;#rgba(128, 80, 52, 0.5)&gt;</c>).</item>
        /// </list></param>
        /// <param name="fontFamily">The font family from which the fonts will be created. If this is a regular font family, the bold, italic and bold-italic versions of the font will be used for the formatted text. Otherwise, the relevant font styles will be toggled (e.g. if the supplied font family is bold, then regular text in the formatted string will be displayed as bold, while bold text in the formatted string will be displayed as regular text).</param>
        /// <param name="fontSize">The size of the fonts to use.</param>
        /// <param name="defaultUnderline">Determines whether text should be underlined by default. This is toggled by <c>&lt;u&gt;&lt;/u&gt;</c> tags.</param>
        /// <param name="defaultBrush">The default <see cref="Brush"/> that will be used for text runs that do not specify a colour. If this is <see langword="null"/>, the default <see cref="Brush"/> will be the one specified in the painting call.</param>
        /// <returns>A lazy collection of <see cref="FormattedText"/> objects. Note that every enumeration of this collection causes the text to be parsed again; if you need to enumerate this collection more than once, you should probably convert it e.g. to a <see cref="List{T}"/>.</returns>
        public static IEnumerable<FormattedText> Format(string text, FontFamily.StandardFontFamilies fontFamily, double fontSize, bool defaultUnderline = false, Brush defaultBrush = null)
        {
            Font normalFont = new Font(FontFamily.ResolveFontFamily(fontFamily), fontSize, defaultUnderline);

            Font boldFont = normalFont;
            Font italicFont = normalFont;
            Font boldItalicFont = normalFont;

            switch (fontFamily)
            {
                case FontFamily.StandardFontFamilies.Courier:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBold), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierOblique), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBoldOblique), fontSize, defaultUnderline);
                    break;
                case FontFamily.StandardFontFamilies.CourierBold:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Courier), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBoldOblique), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierOblique), fontSize, defaultUnderline);
                    break;
                case FontFamily.StandardFontFamilies.CourierOblique:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBoldOblique), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Courier), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBold), fontSize, defaultUnderline);
                    break;
                case FontFamily.StandardFontFamilies.CourierBoldOblique:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierOblique), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.CourierBold), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Courier), fontSize, defaultUnderline);
                    break;

```

```
                case FontFamily.StandardFontFamilies.Helvetica:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBold), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaOblique), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBoldOblique), fontSize, defaultUnderline);
                    break;
                case FontFamily.StandardFontFamilies.HelveticaBold:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBoldOblique), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaOblique), fontSize, defaultUnderline);
                    break;
                case FontFamily.StandardFontFamilies.HelveticaOblique:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBoldOblique), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBold), fontSize, defaultUnderline);
                    break;
                case FontFamily.StandardFontFamilies.HelveticaBoldOblique:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaOblique), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.HelveticaBold), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.Helvetica), fontSize, defaultUnderline);
                    break;

                case FontFamily.StandardFontFamilies.TimesRoman:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesBold), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesItalic), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesBoldItalic), fontSize, defaultUnderline);
                    break;
                case FontFamily.StandardFontFamilies.TimesBold:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesRoman), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesBoldItalic), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesItalic), fontSize, defaultUnderline);
                    break;
                case FontFamily.StandardFontFamilies.TimesItalic:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesBoldItalic), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesRoman), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesBold), fontSize, defaultUnderline);
                    break;
                case FontFamily.StandardFontFamilies.TimesBoldItalic:
                    boldFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesItalic), fontSize, defaultUnderline);
                    italicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesBold), fontSize, defaultUnderline);
                    boldItalicFont = new Font(FontFamily.ResolveFontFamily(FontFamily.StandardFontFamilies.TimesRoman), fontSize,
```

```
defaultUnderline);
                    break;
            }

            return Format(text, normalFont, boldFont, italicFont, boldItalicFont, defaultBrush);
        }

        private static Font GetFont(bool bold, bool italic, bool underlined, Font normalFont, Font boldFont, Font italicFont, Font boldItalicFont)
        {
            if (!bold && !italic)
            {
                if (underlined)
                {
                    return new Font(normalFont.FontFamily, normalFont.FontSize, normalFont.Underline == null);
                }
                else
                {
                    return normalFont;
                }
            }
            else if (bold && !italic)
            {
                if (underlined)
                {
                    return new Font(boldFont.FontFamily, boldFont.FontSize, boldFont.Underline == null);
                }
                else
                {
                    return boldFont;
                }
            }
            else if (!bold && italic)
            {
                if (underlined)
                {
                    return new Font(italicFont.FontFamily, italicFont.FontSize, italicFont.Underline == null);
                }
                else
                {
                    return italicFont;
                }
            }
            else
            {
                if (underlined)
                {
                    return new Font(boldItalicFont.FontFamily, boldItalicFont.FontSize, boldItalicFont.Underline == null);
                }
                else
                {
                    return boldItalicFont;
                }
            }
        }

        private enum Tags
        {
            BoldOpen,
            BoldClose,
            ItalicsOpen,
            ItalicsClose,
            UnderlineOpen,
            UnderlineClose,
            SupOpen,
            SupClose,
            SubOpen,
            SubClose,
            ColourOpen,
            ColourClose,
            None
        }

        private static Tags GetTag(string text, int start, out int tagEnd, out Brush tagBrush)
        {
            StringBuilder tag = new StringBuilder();
            bool closed = false;

            int i = start;

            for (; i < text.Length; i++)
            {
                tag.Append(text[i]);
```

```

                if (text[i] == '>')
                {
                    closed = true;
                    break;
                }
            }

            tagEnd = i;
            tagBrush = null;

            if (!closed)
            {
                return Tags.None;
            }
            else
            {
                string tagString = tag.Replace(" ", "").ToString().ToLowerInvariant();

                if (tagString == "<b>" || tagString == "<strong>")
                {
                    return Tags.BoldOpen;
                }
                else if (tagString == "</b>" || tagString == "</strong>")
                {
                    return Tags.BoldClose;
                }
                else if (tagString == "<i>" || tagString == "<em>")
                {
                    return Tags.ItalicsOpen;
                }
                else if (tagString == "</i>" || tagString == "</em>")
                {
                    return Tags.ItalicsClose;
                }
                else if (tagString == "<u>")
                {
                    return Tags.UnderlineOpen;
                }
                else if (tagString == "</u>")
                {
                    return Tags.UnderlineClose;
                }
                else if (tagString == "<sup>")
                {
                    return Tags.SupOpen;
                }
                else if (tagString == "</sup>")
                {
                    return Tags.SupClose;
                }
                else if (tagString == "<sub>")
                {
                    return Tags.SubOpen;
                }
                else if (tagString == "</sub>")
                {
                    return Tags.SubClose;
                }
                else if (tagString.StartsWith("<#"))
                {
                    string colour = tagString.Substring(1, tagString.Length - 2);

                    Colour? col = null;

                    try
                    {
                        col = Colour.FromCSSString(colour);
                    }
                    catch { }

                    if (col == null)
                    {
                        colour = tagString.Substring(2, tagString.Length - 3);

                        col = Colour.FromCSSString(colour);
                    }

                    if (col != null)
                    {
                        tagBrush = col.Value;
                        return Tags.ColourOpen;
                    }
                    else
                    {
                        return Tags.None;
                    }
```

```
                }
                else if (tagString == "</#>")
                {
                    return Tags.ColourClose;
                }
                else
                {
                    return Tags.None;
                }
            }
        }
    }

    /// <summary>
    /// Contains extension methods for collections of <see cref="FormattedText"/> objects.
    /// </summary>
    public static class FormattedTextExtensions
    {
        internal static Font.DetailedFontMetrics Measure(this IEnumerable<FormattedText> text,
List<FormattedText> items, List<Font.DetailedFontMetrics> allMetrics)
        {
            double width = 0;
            double advanceWidth = 0;

            double lsb = 0;
            double rsb = 0;

            double top = 0;
            double bottom = 0;

            bool isFirst = true;

            foreach (FormattedText txt in text)
            {
                items?.Add(txt);

                if (txt.Script == Script.Normal)
                {
                    Font.DetailedFontMetrics metrics = txt.Font.MeasureTextAdvanced(txt.Text);
                    allMetrics?.Add(metrics);

                    top = Math.Max(top, metrics.Top);
                    bottom = Math.Min(bottom, metrics.Bottom);
                    rsb = metrics.RightSideBearing;

                    advanceWidth += metrics.AdvanceWidth;

                    if (!isFirst)
                    {
                        width += metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing;
                    }
                    else
                    {
                        width += metrics.Width + metrics.RightSideBearing;
                        lsb = metrics.LeftSideBearing;
                    }
                }
                else
                {
                    Font newFont = new Font(txt.Font.FontFamily, txt.Font.FontSize * 0.7);

                    Font.DetailedFontMetrics metrics = newFont.MeasureTextAdvanced(txt.Text);
                    allMetrics?.Add(metrics);

                    advanceWidth += metrics.AdvanceWidth;

                    if (txt.Script == Script.Subscript)
                    {
                        top = Math.Max(top, metrics.Top - txt.Font.FontSize * 0.14);
                        bottom = Math.Min(bottom, metrics.Bottom - txt.Font.FontSize * 0.14);
                    }
                    else if (txt.Script == Script.Superscript)
                    {
                        top = Math.Max(top, metrics.Top + txt.Font.FontSize * 0.33);
                        bottom = Math.Min(bottom, metrics.Bottom + txt.Font.FontSize * 0.33);
                    }

                    rsb = metrics.RightSideBearing;

                    if (!isFirst)
                    {
                        width += metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing;
                    }
                    else
                    {
                        width += metrics.Width + metrics.RightSideBearing;
                        lsb = metrics.LeftSideBearing;
```

```
                    }
                }

                if (isFirst)
                {
                    isFirst = false;
                }
            }

            width -= rsb;

            return new Font.DetailedFontMetrics(width, top - bottom, lsb, rsb, top, bottom, advanceWidth);
        }

        /// <summary>
        /// Measures a collection of <see cref="FormattedText"/> objects.
        /// </summary>
        /// <param name="text">The collection of <see cref="FormattedText"/> objects to be measured.</param>
        /// <returns>A <see cref="Font.DetailedFontMetrics"/> containing detailed measurements for the text obtained by composing the elements in the <see cref="FormattedText"/> collection.</returns>
        public static Font.DetailedFontMetrics Measure(this IEnumerable<FormattedText> text)
        {
            return text.Measure(null, null);
        }
    }

}
```

8.44 Graphics.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using VectSharp.Filters;

namespace VectSharp
{
    internal static class Utils
    {
        public static Point RotatePoint(Point point, double angle)
        {
            return new Point(point.X * Math.Cos(angle) - point.Y * Math.Sin(angle), point.X * Math.Sin(angle) + point.Y * Math.Cos(angle));
        }
    }

    /// <summary>
    /// This interface should be implemented by classes intended to provide graphics output capability to a <see cref="Graphics"/> object.
    /// </summary>
    public interface IGraphicsContext
    {
        /// <summary>
        /// Width of the graphic surface.
        /// </summary>
        double Width { get; }

        /// <summary>
        /// Height of the graphic surface.
        /// </summary>
        double Height { get; }

        /// <summary>
        /// Save the current transform state (rotation, translation, scale). This should be implemented as a LIFO stack.
```

```
        /// </summary>
        void Save();

        /// <summary>
        /// Restore the previous transform state (rotation, translation, scale). This should be implemented as a LIFO stack.
        /// </summary>
        void Restore();

        /// <summary>
        /// Translate the coordinate system origin.
        /// </summary>
        /// <param name="x">The horizontal translation.</param>
        /// <param name="y">The vertical translation.</param>
        void Translate(double x, double y);

        /// <summary>
        /// Rotate the coordinate system around the origin.
        /// </summary>
        /// <param name="angle">The angle (in radians) by which to rotate the coordinate system.</param>
        void Rotate(double angle);

        /// <summary>
        /// Scale the coordinate system with respect to the origin.
        /// </summary>
        /// <param name="scaleX">The horizontal scale.</param>
        /// <param name="scaleY">The vertical scale.</param>
        void Scale(double scaleX, double scaleY);

        /// <summary>
        /// Transform the coordinate system with the specified transformation matrix [ [a, c, e], [b, d, f], [0, 0, 1] ].
        /// </summary>
        /// <param name="a">The first element of the first column.</param>
        /// <param name="b">The second element of the first column.</param>
        /// <param name="c">The first element of the second column.</param>
        /// <param name="d">The second element of the second column.</param>
        /// <param name="e">The first element of the third column.</param>
        /// <param name="f">The second element of the third column.</param>
        void Transform(double a, double b, double c, double d, double e, double f);

        /// <summary>
        /// The current font.
        /// </summary>
        Font Font { get; set; }

        /// <summary>
        /// The current text baseline.
        /// </summary>
        TextBaselines TextBaseline { get; set; }

        /// <summary>
        /// Fill a text string using the current <see cref="Font"/> and <see cref="TextBaseline"/>.
        /// </summary>
        /// <param name="text">The string to draw.</param>
        /// <param name="x">The horizontal coordinate of the text origin.</param>
        /// <param name="y">The vertical coordinate of the text origin.</param>
        void FillText(string text, double x, double y);

        /// <summary>
        /// Stroke the outline of a text string using the current <see cref="Font"/> and <see cref="TextBaseline"/>.
        /// </summary>
        /// <param name="text">The string to draw.</param>
        /// <param name="x">The horizontal coordinate of the text origin.</param>
        /// <param name="y">The vertical coordinate of the text origin.</param>
        void StrokeText(string text, double x, double y);

        /// <summary>
        /// Change the current point without drawing a line from the previous point. If necessary, start a new figure.
        /// </summary>
        /// <param name="x">The horizontal coordinate of the point.</param>
        /// <param name="y">The vertical coordinate of the point.</param>
        void MoveTo(double x, double y);

        /// <summary>
        /// Draw a line from the previous point to the specified point.
        /// </summary>
        /// <param name="x">The horizontal coordinate of the point.</param>
        /// <param name="y">The vertical coordinate of the point.</param>
        void LineTo(double x, double y);

        /// <summary>
        /// Close the current figure.
        /// </summary>
```

```
        void Close();

        /// <summary>
        /// Stroke the current path using the current <see cref="StrokeStyle"/>, <see cref="LineWidth"/>, <see cref="LineCap"/>, <see cref="LineJoin"/> and <see cref="LineDash"/>.
        /// </summary>
        void Stroke();

        /// <summary>
        /// Set the current clipping path as the intersection of the previous clipping path and the current path.
        /// </summary>
        void SetClippingPath();

        /// <summary>
        /// Current brush used to fill paths.
        /// </summary>
        Brush FillStyle { get; }

        /// <summary>
        /// Set the current <see cref="FillStyle"/>.
        /// </summary>
        /// <param name="style">A <see cref="ValueTuple{Int32, Int32, Int32, Double}"/> containing component information for the colour. For r, g, and b, range: [0, 255]; for a, range: [0, 1].</param>
        void SetFillStyle((int r, int g, int b, double a) style);

        /// <summary>
        /// Set the current <see cref="FillStyle"/>.
        /// </summary>
        /// <param name="style">The new fill style.</param>
        void SetFillStyle(Brush style);

        /// <summary>
        /// Current brush used to stroke paths.
        /// </summary>
        Brush StrokeStyle { get; }

        /// <summary>
        /// Set the current <see cref="StrokeStyle"/>.
        /// </summary>
        /// <param name="style">A <see cref="ValueTuple{Int32, Int32, Int32, Double}"/> containing component information for the colour. For r, g, and b, range: [0, 255]; for a, range: [0, 1].</param>
        void SetStrokeStyle((int r, int g, int b, double a) style);

        /// <summary>
        /// Set the current <see cref="StrokeStyle"/>.
        /// </summary>
        /// <param name="style">The new stroke style.</param>
        void SetStrokeStyle(Brush style);

        /// <summary>
        /// Add to the current figure a cubic Bezier from the current point to a destination point, with two control points.
        /// </summary>
        /// <param name="p1X">The horizontal coordinate of the first control point.</param>
        /// <param name="p1Y">The vertical coordinate of the first control point.</param>
        /// <param name="p2X">The horizontal coordinate of the second control point.</param>
        /// <param name="p2Y">The vertical coordinate of the second control point.</param>
        /// <param name="p3X">The horizontal coordinate of the destination point.</param>
        /// <param name="p3Y">The vertical coordinate of the destination point.</param>
        void CubicBezierTo(double p1X, double p1Y, double p2X, double p2Y, double p3X, double p3Y);

        /// <summary>
        /// Add a rectangle figure to the current path.
        /// </summary>
        /// <param name="x0">The horizontal coordinate of the top-left corner of the rectangle.</param>
        /// <param name="y0">The vertical coordinate of the top-left corner of the rectangle.</param>
        /// <param name="width">The width of corner of the rectangle.</param>
        /// <param name="height">The height of corner of the rectangle.</param>
        void Rectangle(double x0, double y0, double width, double height);

        /// <summary>
        /// Fill the current path using the current <see cref="FillStyle"/>.
        /// </summary>
        void Fill();

        /// <summary>
        /// Current line width used to stroke paths.
        /// </summary>
        double LineWidth { get; set; }

        /// <summary>
        /// Current line cap used to stroke paths.
        /// </summary>
        LineCaps LineCap { set; }

```

```
        /// <summary>
        /// Current line join used to stroke paths.
        /// </summary>
        LineJoins LineJoin { set; }

        /// <summary>
        /// Set the current line dash pattern.
        /// </summary>
        /// <param name="dash">The line dash pattern.</param>
        void SetLineDash(LineDash dash);

        /// <summary>
        /// The current tag. How this can be used depends on each implementation.
        /// </summary>
        string Tag { get; set; }

        /// <summary>
        /// Draw a raster image.
        /// </summary>
        /// <param name="sourceX">The horizontal coordinate of the top-left corner of the rectangle delimiting the source area of the image.</param>
        /// <param name="sourceY">The vertical coordinate of the top-left corner of the rectangle delimiting the source area of the image.</param>
        /// <param name="sourceWidth">The width of the rectangle delimiting the source area of the image.</param>
        /// <param name="sourceHeight">The height of the rectangle delimiting the source area of the image.</param>
        /// <param name="destinationX">The horizontal coordinate of the top-left corner of the rectangle delimiting the destination area of the image.</param>
        /// <param name="destinationY">The vertical coordinate of the top-left corner of the rectangle delimiting the destination area of the image.</param>
        /// <param name="destinationWidth">The width of the rectangle delimiting the destination area of the image.</param>
        /// <param name="destinationHeight">The height of the rectangle delimiting the destination area of the image.</param>
        /// <param name="image">The image to draw.</param>
        void DrawRasterImage(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image);

        /// <summary>
        /// Draws a <see cref="Graphics"/> object, applying the specified <paramref name="filter"/>.
        /// </summary>
        /// <param name="graphics">The <see cref="Graphics"/> object to draw on the current <see cref="Graphics"/> object.</param>
        /// <param name="filter">An <see cref="IFilter"/> object, representing the filter to apply to the <paramref name="graphics"/> object.</param>
        void DrawFilteredGraphics(Graphics graphics, IFilter filter);
    }

    /// <summary>
    /// The exception that is thrown when an unbalanced graphics state stack occurs.
    /// </summary>
    public class UnbalancedStackException : Exception
    {
        internal UnbalancedStackException(int excessSave, int excessRestore) : base("The graphics state stack is unbalanced!\nThere are " + excessSave + " calls to Graphics.Save() and " + excessRestore + " calls to Graphics.Restore() in excess!") { }
    }

    /// <summary>
    /// Represents an abstract drawing surface.
    /// </summary>
    public partial class Graphics
    {
        /// <summary>
        /// Determines how an unbalanced graphics state stack (which occurs if the number of calls to <see cref="Save"/> and <see cref="Restore"/> is not equal) will be treated. The default is <see cref="UnbalancedStackActions.Throw"/>.
        /// </summary>
        public static UnbalancedStackActions UnbalancedStackAction { get; set; } = UnbalancedStackActions.Throw;

        internal List<IGraphicsAction> Actions = new List<IGraphicsAction>();

        /// <summary>
        /// Fill a <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="path">The <see cref="GraphicsPath"/> to fill.</param>
        /// <param name="fillColour">The <see cref="Brush"/> with which to fill the <see cref="GraphicsPath"/>.</param>
        /// <param name="tag">A tag to identify the filled path.</param>
        public void FillPath(GraphicsPath path, Brush fillColour, string tag = null)
        {
            Actions.Add(new PathAction(path, fillColour, null, 0, LineCaps.Butt, LineJoins.Miter, LineDash.SolidLine, tag, false));
        }
```

```
        /// <summary>
        /// Stroke a <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="path">The <see cref="GraphicsPath"/> to stroke.</param>
        /// <param name="strokeColour">The <see cref="Brush"/> with which to stroke the <see cref="GraphicsPath"/>.</param>
        /// <param name="lineWidth">The width of the line with which the path is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the path.</param>
        /// <param name="lineJoin">The line join to use to stroke the path.</param>
        /// <param name="lineDash">The line dash to use to stroke the path.</param>
        /// <param name="tag">A tag to identify the stroked path.</param>
        public void StrokePath(GraphicsPath path, Brush strokeColour, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            Actions.Add(new PathAction(path, null, strokeColour, lineWidth, lineCap, lineJoin, lineDash ?? LineDash.SolidLine, tag, false));
        }

        /// <summary>
        /// Intersect the current clipping path with the specified <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="path">The <see cref="GraphicsPath"/> to intersect with the current clipping path.</param>
        /// <param name="tag">A tag to identify the clipping path.</param>
        public void SetClippingPath(GraphicsPath path, string tag = null)
        {
            Actions.Add(new PathAction(path, null, null, 0, LineCaps.Butt, LineJoins.Miter, LineDash.SolidLine, tag, true));
        }

        /// <summary>
        /// Intersect the current clipping path with the specified rectangle.
        /// </summary>
        /// <param name="leftX">The horizontal coordinate of the top-left corner of the rectangle.</param>
        /// <param name="topY">The vertical coordinate of the top-left corner of the rectangle.</param>
        /// <param name="width">The width of the rectangle.</param>
        /// <param name="height">The height of the rectangle.</param>
        /// <param name="tag">A tag to identify the clipping path.</param>
        public void SetClippingPath(double leftX, double topY, double width, double height, string tag = null)
        {
            SetClippingPath(new Point(leftX, topY), new Size(width, height), tag);
        }

        /// <summary>
        /// Intersect the current clipping path with the specified rectangle.
        /// </summary>
        /// <param name="topLeft">The top-left corner of the rectangle.</param>
        /// <param name="size">The size of the rectangle.</param>
        /// <param name="tag">A tag to identify the clipping path.</param>
        public void SetClippingPath(Point topLeft, Size size, string tag = null)
        {
            Actions.Add(new PathAction(new GraphicsPath().MoveTo(topLeft).LineTo(topLeft.X + size.Width, topLeft.Y).LineTo(topLeft.X + size.Width, topLeft.Y + size.Height).LineTo(topLeft.X, topLeft.Y + size.Height).Close(), null, null, 0, LineCaps.Butt, LineJoins.Miter, LineDash.SolidLine, tag, true));
        }

        /// <summary>
        /// Rotate the coordinate system around the origin.
        /// </summary>
        /// <param name="angle">The angle (in radians) by which to rotate the coordinate system.</param>
        /// <param name="tag">A tag to identify the transform.</param>
        public void Rotate(double angle, string tag = null)
        {
            Actions.Add(new TransformAction(angle, tag));
        }

        /// <summary>
        /// Rotate the coordinate system around a pivot point.
        /// </summary>
        /// <param name="angle">The angle (in radians) by which to rotate the coordinate system.</param>
        /// <param name="pivot">The pivot around which the coordinate system is to be rotated.</param>
        /// <param name="tag">A tag to identify the transform.</param>
        public void RotateAt(double angle, Point pivot, string tag = null)
        {
            string tag1 = null;
            string tag2 = null;
            string tag3 = null;

```

```
            if (!string.IsNullOrEmpty(tag))
            {
                tag1 = tag + "_@1";
                tag2 = tag + "_@2";
                tag3 = tag + "_@3";
            }

            Actions.Add(new TransformAction(pivot, tag1));
            Actions.Add(new TransformAction(angle, tag2));
            Actions.Add(new TransformAction(new Point(-pivot.X, -pivot.Y), tag3));
        }


        /// <summary>
        /// Transform the coordinate system with the specified transformation matrix [ [a, c, e], [b, d, f], [0, 0, 1] ].
        /// </summary>
        /// <param name="a">The first element of the first column.</param>
        /// <param name="b">The second element of the first column.</param>
        /// <param name="c">The first element of the second column.</param>
        /// <param name="d">The second element of the second column.</param>
        /// <param name="e">The first element of the third column.</param>
        /// <param name="f">The second element of the third column.</param>
        /// <param name="tag">A tag to identify the transform.</param>
        public void Transform(double a, double b, double c, double d, double e, double f, string tag = null)
        {
            double[,] matrix = new double[,] { { a, c, e }, { b, d, f }, { 0, 0, 1 } };
            Actions.Add(new TransformAction(matrix, tag));
        }

        /// <summary>
        /// Translate the coordinate system origin.
        /// </summary>
        /// <param name="x">The horizontal translation.</param>
        /// <param name="y">The vertical translation.</param>
        /// <param name="tag">A tag to identify the transform.</param>
        public void Translate(double x, double y, string tag = null)
        {
            Actions.Add(new TransformAction(new Point(x, y), tag));
        }

        /// <summary>
        /// Translate the coordinate system origin.
        /// </summary>
        /// <param name="delta">The new origin point.</param>
        /// <param name="tag">A tag to identify the transform.</param>
        public void Translate(Point delta, string tag = null)
        {
            Actions.Add(new TransformAction(delta, tag));
        }

        /// <summary>
        /// Scale the coordinate system with respect to the origin.
        /// </summary>
        /// <param name="scaleX">The horizontal scale.</param>
        /// <param name="scaleY">The vertical scale.</param>
        /// <param name="tag">A tag to identify the transform.</param>
        public void Scale(double scaleX, double scaleY, string tag = null)
        {
            Actions.Add(new TransformAction(new Size(scaleX, scaleY), tag));
        }

        /// <summary>
        /// Fill a rectangle.
        /// </summary>
        /// <param name="topLeft">The top-left corner of the rectangle.</param>
        /// <param name="size">The size of the rectangle.</param>
        /// <param name="fillColour">The colour with which to fill the rectangle.</param>
        /// <param name="tag">A tag to identify the filled rectangle.</param>
        public void FillRectangle(Point topLeft, Size size, Brush fillColour, string tag = null)
        {
            Actions.Add(new RectangleAction(topLeft, size, fillColour, null, 0, LineCaps.Butt, LineJoins.Miter, LineDash.SolidLine, tag));
        }

        /// <summary>
        /// Fill a rectangle.
        /// </summary>
        /// <param name="leftX">The horizontal coordinate of the top-left corner of the rectangle.</param>
        /// <param name="topY">The vertical coordinate of the top-left corner of the rectangle.</param>
        /// <param name="width">The width of the rectangle.</param>
        /// <param name="height">The height of the rectangle.</param>
        /// <param name="fillColour">The colour with which to fill the rectangle.</param>
        /// <param name="tag">A tag to identify the filled rectangle.</param>
```

```
        public void FillRectangle(double leftX, double topY, double width, double height, Brush fillColour, string tag = null)
        {
            Actions.Add(new RectangleAction(new Point(leftX, topY), new Size(width, height), fillColour, null, 0, LineCaps.Butt, LineJoins.Miter, LineDash.SolidLine, tag));
        }

        /// <summary>
        /// Stroke a rectangle.
        /// </summary>
        /// <param name="topLeft">The top-left corner of the rectangle.</param>
        /// <param name="size">The size of the rectangle.</param>
        /// <param name="strokeColour">The colour with which to stroke the rectangle.</param>
        /// <param name="lineWidth">The width of the line with which the rectangle is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the rectangle.</param>
        /// <param name="lineJoin">The line join to use to stroke the rectangle.</param>
        /// <param name="lineDash">The line dash to use to stroke the rectangle.</param>
        /// <param name="tag">A tag to identify the filled rectangle.</param>
        public void StrokeRectangle(Point topLeft, Size size, Brush strokeColour, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            Actions.Add(new RectangleAction(topLeft, size, null, strokeColour, lineWidth, lineCap, lineJoin, lineDash ?? LineDash.SolidLine, tag));
        }

        /// <summary>
        /// Stroke a rectangle.
        /// </summary>
        /// <param name="leftX">The horizontal coordinate of the top-left corner of the rectangle.</param>
        /// <param name="topY">The vertical coordinate of the top-left corner of the rectangle.</param>
        /// <param name="width">The width of the rectangle.</param>
        /// <param name="height">The height of the rectangle.</param>
        /// <param name="strokeColour">The colour with which to stroke the rectangle.</param>
        /// <param name="lineWidth">The width of the line with which the rectangle is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the rectangle.</param>
        /// <param name="lineJoin">The line join to use to stroke the rectangle.</param>
        /// <param name="lineDash">The line dash to use to stroke the rectangle.</param>
        /// <param name="tag">A tag to identify the filled rectangle.</param>
        public void StrokeRectangle(double leftX, double topY, double width, double height, Brush strokeColour, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            Actions.Add(new RectangleAction(new Point(leftX, topY), new Size(width, height), null, strokeColour, lineWidth, lineCap, lineJoin, lineDash ?? LineDash.SolidLine, tag));
        }

        /// <summary>
        /// Draw a raster image.
        /// </summary>
        /// <param name="sourceX">The horizontal coordinate of the top-left corner of the rectangle delimiting the source area of the image.</param>
        /// <param name="sourceY">The vertical coordinate of the top-left corner of the rectangle delimiting the source area of the image.</param>
        /// <param name="sourceWidth">The width of the rectangle delimiting the source area of the image.</param>
        /// <param name="sourceHeight">The height of the rectangle delimiting the source area of the image.</param>
        /// <param name="destinationX">The horizontal coordinate of the top-left corner of the rectangle delimiting the destination area of the image.</param>
        /// <param name="destinationY">The vertical coordinate of the top-left corner of the rectangle delimiting the destination area of the image.</param>
        /// <param name="destinationWidth">The width of the rectangle delimiting the destination area of the image.</param>
        /// <param name="destinationHeight">The height of the rectangle delimiting the destination area of the image.</param>
        /// <param name="image">The image to draw.</param>
        /// <param name="tag">A tag to identify the drawn image.</param>
        public void DrawRasterImage(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image, string tag = null)
        {
            Actions.Add(new RasterImageAction(sourceX, sourceY, sourceWidth, sourceHeight, destinationX, destinationY, destinationWidth, destinationHeight, image, tag));
        }

        /// <summary>
        /// Draw a raster image.
        /// </summary>
        /// <param name="x">The horizontal coordinate of the top-left corner of the rectangle delimiting the destination area of the image.</param>
        /// <param name="y">The vertical coordinate of the top-left corner of the rectangle delimiting the destination area of the image.</param>
        /// <param name="image">The image to draw.</param>
        /// <param name="tag">A tag to identify the drawn image.</param>
```

```
        public void DrawRasterImage(double x, double y, RasterImage image, string tag = null)
        {
            DrawRasterImage(0, 0, image.Width, image.Height, x, y, image.Width, image.Height, image, tag);
        }

        /// <summary>
        /// Draw a raster image.
        /// </summary>
        /// <param name="position">The the top-left corner of the rectangle delimiting the destination area of the image.</param>
        /// <param name="image">The image to draw.</param>
        /// <param name="tag">A tag to identify the drawn image.</param>
        public void DrawRasterImage(Point position, RasterImage image, string tag = null)
        {
            DrawRasterImage(0, 0, image.Width, image.Height, position.X, position.Y, image.Width, image.Height, image, tag);
        }

        /// <summary>
        /// Draw a raster image.
        /// </summary>
        /// <param name="x">The horizontal coordinate of the top-left corner of the rectangle delimiting the destination area of the image.</param>
        /// <param name="y">The vertical coordinate of the top-left corner of the rectangle delimiting the destination area of the image.</param>
        /// <param name="width">The width of the rectangle delimiting the destination area of the image.</param>
        /// <param name="height">The height of the rectangle delimiting the destination area of the image.</param>
        /// <param name="image">The image to draw.</param>
        /// <param name="tag">A tag to identify the drawn image.</param>
        public void DrawRasterImage(double x, double y, double width, double height, RasterImage image, string tag = null)
        {
            DrawRasterImage(0, 0, image.Width, image.Height, x, y, width, height, image, tag);
        }

        /// <summary>
        /// Draw a raster image.
        /// </summary>
        /// <param name="position">The the top-left corner of the rectangle delimiting the destination area of the image.</param>
        /// <param name="size">The size of the rectangle delimiting the destination area of the image.</param>
        /// <param name="image">The image to draw.</param>
        /// <param name="tag">A tag to identify the drawn image.</param>
        public void DrawRasterImage(Point position, Size size, RasterImage image, string tag = null)
        {
            DrawRasterImage(0, 0, image.Width, image.Height, position.X, position.Y, size.Width, size.Height, image, tag);
        }

        /// <summary>
        /// Save the current transform state (rotation, translation, scale).
        /// </summary>
        public void Save()
        {
            Actions.Add(new StateAction(StateAction.StateActionTypes.Save));
        }

        /// <summary>
        /// Restore the previous transform state (rotation, translation scale).
        /// </summary>
        public void Restore()
        {
            Actions.Add(new StateAction(StateAction.StateActionTypes.Restore));
        }

        private void FixGraphicsStateStack()
        {
            if (UnbalancedStackAction == UnbalancedStackActions.Ignore)
            {
                return;
            }

            int count = 0;
            List<int> toBeRemoved = new List<int>();

            for (int i = 0; i < this.Actions.Count; i++)
            {
                if (this.Actions[i] is StateAction st)
                {
                    if (st.StateActionType == StateAction.StateActionTypes.Save)
                    {
                        count++;
                    }
```

```
                    else if (st.StateActionType == StateAction.StateActionTypes.Restore)
                    {
                        if (count == 0)
                        {
                            toBeRemoved.Add(i);
                        }
                        else
                        {
                            count--;
                        }
                    }
                }
            }

            if (UnbalancedStackAction == UnbalancedStackActions.Throw)
            {
                if (count > 0 || toBeRemoved.Count > 0)
                {
                    throw new UnbalancedStackException(count, toBeRemoved.Count);
                }
            }
            else if (UnbalancedStackAction == UnbalancedStackActions.SilentlyFix)
            {
                if (count > 0 || toBeRemoved.Count > 0)
                {
                    for (int i = toBeRemoved.Count - 1; i >= 0; i--)
                    {
                        this.Actions.RemoveAt(toBeRemoved[i]);
                    }

                    for (int i = 0; i < count; i++)
                    {
                        this.Restore();
                    }

                    this.FixGraphicsStateStack();
                }
            }
        }

        /// <summary>
        /// Copy the current graphics to an instance of a class implementing <see cref="IGraphicsContext"/>.
        /// </summary>
        /// <param name="destinationContext">The <see cref="IGraphicsContext"/> on which the graphics are to be copied.</param>
        public void CopyToIGraphicsContext(IGraphicsContext destinationContext)
        {
            this.FixGraphicsStateStack();

            for (int i = 0; i < this.Actions.Count; i++)
            {
                if (this.Actions[i] is RectangleAction)
                {
                    RectangleAction rec = this.Actions[i] as RectangleAction;

                    destinationContext.Tag = rec.Tag;
                    destinationContext.Rectangle(rec.TopLeft.X, rec.TopLeft.Y, rec.Size.Width, rec.Size.Height);

                    if (rec.Fill != null)
                    {
                        if (destinationContext.FillStyle != rec.Fill)
                        {
                            destinationContext.SetFillStyle(rec.Fill);
                        }
                        destinationContext.Fill();
                    }
                    else if (rec.Stroke != null)
                    {
                        if (destinationContext.StrokeStyle != rec.Stroke)
                        {
                            destinationContext.SetStrokeStyle(rec.Stroke);
                        }
                        if (destinationContext.LineWidth != rec.LineWidth)
                        {
                            destinationContext.LineWidth = rec.LineWidth;
                        }
                        destinationContext.SetLineDash(rec.LineDash);
                        destinationContext.LineCap = rec.LineCap;
                        destinationContext.LineJoin = rec.LineJoin;

                        destinationContext.Stroke();
                    }
                }
                else if (this.Actions[i] is PathAction)
                {
```

```
                    PathAction pth = this.Actions[i] as PathAction;

                    destinationContext.Tag = pth.Tag;

                    for (int j = 0; j < pth.Path.Segments.Count; j++)
                    {
                        switch (pth.Path.Segments[j].Type)
                        {
                            case SegmentType.Move:
                                destinationContext.MoveTo(pth.Path.Segments[j].Point.X, pth.Path.Segments[j].Point.Y);
                                break;
                            case SegmentType.Line:
                                destinationContext.LineTo(pth.Path.Segments[j].Point.X, pth.Path.Segments[j].Point.Y);
                                break;
                            case SegmentType.CubicBezier:
                                destinationContext.CubicBezierTo(pth.Path.Segments[j].Points[0].X, pth.Path.Segments[j].Points[0].Y, pth.Path.Segments[j].Points[1].X, pth.Path.Segments[j].Points[1].Y, pth.Path.Segments[j].Points[2].X, pth.Path.Segments[j].Points[2].Y);
                                break;
                            case SegmentType.Arc:
                                {
                                    ArcSegment seg = pth.Path.Segments[j] as ArcSegment;
                                    Segment[] segs = seg.ToBezierSegments();
                                    for (int k = 0; k < segs.Length; k++)
                                    {
                                        switch (segs[k].Type)
                                        {
                                            case SegmentType.Move:
                                                destinationContext.MoveTo(segs[k].Point.X, segs[k].Point.Y);
                                                break;
                                            case SegmentType.Line:
                                                destinationContext.LineTo(segs[k].Point.X, segs[k].Point.Y);
                                                break;
                                            case SegmentType.CubicBezier:
                                                destinationContext.CubicBezierTo(segs[k].Points[0].X, segs[k].Points[0].Y, segs[k].Points[1].X, segs[k].Points[1].Y, segs[k].Points[2].X, segs[k].Points[2].Y);
                                                break;
                                        }
                                    }
                                }
                                break;
                            case SegmentType.Close:
                                destinationContext.Close();
                                break;
                        }
                    }

                    if (pth.IsClipping)
                    {
                        destinationContext.SetClippingPath();
                    }
                    else
                    {
                        if (pth.Fill != null)
                        {
                            if (destinationContext.FillStyle != pth.Fill)
                            {
                                destinationContext.SetFillStyle(pth.Fill);
                            }
                            destinationContext.Fill();
                        }
                        else if (pth.Stroke != null)
                        {
                            if (destinationContext.StrokeStyle != pth.Stroke)
                            {
                                destinationContext.SetStrokeStyle(pth.Stroke);
                            }
                            if (destinationContext.LineWidth != pth.LineWidth)
                            {
                                destinationContext.LineWidth = pth.LineWidth;
                            }
                            destinationContext.SetLineDash(pth.LineDash);
                            destinationContext.LineCap = pth.LineCap;
                            destinationContext.LineJoin = pth.LineJoin;

                            destinationContext.Stroke();
                        }
                    }
                }
                else if (this.Actions[i] is TextAction)
                {
                    TextAction txt = this.Actions[i] as TextAction;
```

```

                    destinationContext.Tag = txt.Tag;
                    if (destinationContext.TextBaseline != txt.TextBaseline)
                    {
                        destinationContext.TextBaseline = txt.TextBaseline;
                    }
                    destinationContext.Font = txt.Font;

                    if (txt.Fill != null)
                    {
                        if (destinationContext.FillStyle != txt.Fill)
                        {
                            destinationContext.SetFillStyle(txt.Fill);
                        }
                        destinationContext.FillText(txt.Text, txt.Origin.X, txt.Origin.Y);
                    }
                    else if (txt.Stroke != null)
                    {
                        if (destinationContext.StrokeStyle != txt.Stroke)
                        {
                            destinationContext.SetStrokeStyle(txt.Stroke);
                        }
                        if (destinationContext.LineWidth != txt.LineWidth)
                        {
                            destinationContext.LineWidth = txt.LineWidth;
                        }
                        destinationContext.SetLineDash(txt.LineDash);
                        destinationContext.LineCap = txt.LineCap;
                        destinationContext.LineJoin = txt.LineJoin;

                        destinationContext.StrokeText(txt.Text, txt.Origin.X, txt.Origin.Y);
                    }
                }
                else if (this.Actions[i] is TransformAction)
                {
                    TransformAction trf = this.Actions[i] as TransformAction;

                    destinationContext.Tag = trf.Tag;

                    if (trf.Delta != null)
                    {
                        destinationContext.Translate(((Point)trf.Delta).X, ((Point)trf.Delta).Y);
                    }
                    else if (trf.Angle != null)
                    {
                        destinationContext.Rotate((double)trf.Angle);
                    }
                    else if (trf.Scale != null)
                    {
                        destinationContext.Scale(((Size)trf.Scale).Width, ((Size)trf.Scale).Height);
                    }
                    else if (trf.Matrix != null)
                    {
                        destinationContext.Transform(trf.Matrix[0, 0], trf.Matrix[1, 0], trf.Matrix[0,
1], trf.Matrix[1, 1], trf.Matrix[0, 2], trf.Matrix[1, 2]);
                    }
                }
                else if (this.Actions[i] is StateAction)
                {
                    destinationContext.Tag = ((StateAction)this.Actions[i]).Tag;

                    if (((StateAction)this.Actions[i]).StateActionType ==
StateAction.StateActionTypes.Save)
                    {
                        destinationContext.Save();
                    }
                    else
                    {
                        destinationContext.Restore();
                    }
                }
                else if (this.Actions[i] is RasterImageAction)
                {
                    RasterImageAction img = this.Actions[i] as RasterImageAction;
                    destinationContext.Tag = img.Tag;
                    destinationContext.DrawRasterImage(img.SourceX, img.SourceY, img.SourceWidth,
img.SourceHeight, img.DestinationX, img.DestinationY, img.DestinationWidth, img.DestinationHeight,
img.Image);
                }
                else if (this.Actions[i] is FilteredGraphicsAction)
                {
                    FilteredGraphicsAction fil = this.Actions[i] as FilteredGraphicsAction;
                    destinationContext.Tag = fil.Tag;
                    destinationContext.DrawFilteredGraphics(fil.Content, fil.Filter);
                }
            }
        }
```

```
        /// <summary>
        /// Draws a <see cref="Graphics"/> object on the current <see cref="Graphics"/> object.
        /// </summary>
        /// <param name="origin">The point at which to place the origin of <paramref name="graphics"/>.</param>
        /// <param name="graphics">The <see cref="Graphics"/> object to draw on the current <see cref="Graphics"/> object.</param>
        public void DrawGraphics(Point origin, Graphics graphics)
        {
            this.Save();
            this.Translate(origin);

            graphics.FixGraphicsStateStack();

            this.Actions.AddRange(graphics.Actions);

            this.Restore();
        }

        /// <summary>
        /// Draws a <see cref="Graphics"/> object on the current <see cref="Graphics"/> object, prepending the supplied <paramref name="tag"/> to the tags contained in the <see cref="Graphics"/> object being drawn.
        /// </summary>
        /// <param name="originX">The horizontal coordinate at which to place the origin of <paramref name="graphics"/>.</param>
        /// <param name="originY">The vertical coordinate at which to place the origin of <paramref name="graphics"/>.</param>
        /// <param name="graphics">The <see cref="Graphics"/> object to draw on the current <see cref="Graphics"/> object.</param>
        /// <param name="tag">The tag to prepend to the tags contained in the <paramref name="graphics"/> object.</param>
        public void DrawGraphics(double originX, double originY, Graphics graphics, string tag)
        {
            this.DrawGraphics(new Point(originX, originY), graphics, tag);
        }

        /// <summary>
        /// Draws a <see cref="Graphics"/> object on the current <see cref="Graphics"/> object, prepending the supplied <paramref name="tag"/> to the tags contained in the <see cref="Graphics"/> object being drawn.
        /// </summary>
        /// <param name="origin">The point at which to place the origin of <paramref name="graphics"/>.</param>
        /// <param name="graphics">The <see cref="Graphics"/> object to draw on the current <see cref="Graphics"/> object.</param>
        /// <param name="tag">The tag to prepend to the tags contained in the <paramref name="graphics"/> object.</param>
        public void DrawGraphics(Point origin, Graphics graphics, string tag)
        {
            this.Save();

            if (!string.IsNullOrEmpty(tag))
            {
                this.Translate(origin, tag + "@t");
            }
            else
            {
                this.Translate(origin, tag);
            }

            graphics.FixGraphicsStateStack();

            foreach (IGraphicsAction action in graphics.Actions)
            {
                IGraphicsAction clone = action.ShallowClone();

                if (clone is IPrintableAction print)
                {
                    if (string.IsNullOrEmpty(print.Tag))
                    {
                        print.Tag = tag;
                    }
                    else
                    {
                        print.Tag = tag + "/" + print.Tag;
                    }
                }

                this.Actions.Add(clone);
            }

            this.Restore();
        }

        /// <summary>
```

```
        /// Draws a <see cref="Graphics"/> object on the current <see cref="Graphics"/> object.
        /// </summary>
        /// <param name="originX">The horizontal coordinate at which to place the origin of <paramref name="graphics"/>.</param>
        /// <param name="originY">The vertical coordinate at which to place the origin of <paramref name="graphics"/>.</param>
        /// <param name="graphics">The <see cref="Graphics"/> object to draw on the current <see cref="Graphics"/> object.</param>
        public void DrawGraphics(double originX, double originY, Graphics graphics)
        {
            this.DrawGraphics(new Point(originX, originY), graphics);
        }

        /// <summary>
        /// Draws a <see cref="Graphics"/> object on the current <see cref="Graphics"/> object, applying the specified <paramref name="filter"/>.
        /// </summary>
        /// <param name="origin">The point at which to place the origin of <paramref name="graphics"/>.</param>
        /// <param name="graphics">The <see cref="Graphics"/> object to draw on the current <see cref="Graphics"/> object.</param>
        /// <param name="filter">An <see cref="IFilter"/> object, representing the filter to apply to the <paramref name="graphics"/> object.</param>
        /// <param name="tag">A tag to identify the filter.</param>
        public void DrawGraphics(Point origin, Graphics graphics, IFilter filter, string tag = null)
        {
            this.Save();

            if (!string.IsNullOrEmpty(tag))
            {
                this.Translate(origin, tag + "@t");
            }
            else
            {
                this.Translate(origin, tag);
            }

            Graphics clone = new Graphics();
            clone.Actions.AddRange(graphics.Actions);
            clone.FixGraphicsStateStack();

            this.Actions.Add(new FilteredGraphicsAction(clone, filter) { Tag = tag });

            this.Restore();
        }

        /// <summary>
        /// Draws a <see cref="Graphics"/> object on the current <see cref="Graphics"/> object, applying the specified <paramref name="filter"/>.
        /// </summary>
        /// <param name="originX">The horizontal coordinate at which to place the origin of <paramref name="graphics"/>.</param>
        /// <param name="originY">The vertical coordinate at which to place the origin of <paramref name="graphics"/>.</param>
        /// <param name="graphics">The <see cref="Graphics"/> object to draw on the current <see cref="Graphics"/> object.</param>
        /// <param name="filter">An <see cref="IFilter"/> object, representing the filter to apply to the <paramref name="graphics"/> object.</param>
        /// <param name="tag">A tag to identify the filter.</param>
        public void DrawGraphics(double originX, double originY, Graphics graphics, IFilter filter, string tag = null)
        {
            this.DrawGraphics(new Point(originX, originY), graphics, filter, tag);
        }


        internal static Point Multiply(double[,] matrix, Point pt)
        {
            double x = matrix[0, 0] * pt.X + matrix[0, 1] * pt.Y + matrix[0, 2];
            double y = matrix[1, 0] * pt.X + matrix[1, 1] * pt.Y + matrix[1, 2];
            double z = matrix[2, 0] * pt.X + matrix[2, 1] * pt.Y + matrix[2, 2];

            return new Point(x / z, y / z);
        }

        internal static double[,] Multiply(double[,] m1, double[,] m2)
        {
            return new double[3, 3]
            {
                { m1[0,0] * m2[0,0] + m1[0,1] * m2[1,0] + m1[0,2] *m2[2,0], m1[0,0] * m2[0,1] + m1[0,1] * m2[1,1] + m1[0,2] * m2[2,1], m1[0,0] * m2[0,2] + m1[0,1] * m2[1,2] + m1[0,2] * m2[2,2] },
                { m1[1,0] * m2[0,0] + m1[1,1] * m2[1,0] + m1[1,2] *m2[2,0], m1[1,0] * m2[0,1] + m1[1,1] * m2[1,1] + m1[1,2] * m2[2,1], m1[1,0] * m2[0,2] + m1[1,1] * m2[1,2] + m1[1,2] * m2[2,2] },
                { m1[2,0] * m2[0,0] + m1[2,1] * m2[1,0] + m1[2,2] *m2[2,0], m1[2,0] * m2[0,1] + m1[2,1] * m2[1,1] + m1[2,2] * m2[2,1], m1[2,0] * m2[0,2] + m1[2,1] * m2[1,2] + m1[2,2] * m2[2,2] }
            };
        }
```

```
        internal static double[,] TranslationMatrix(double dx, double dy)
        {
            return new double[3, 3]
            {
                {1, 0, dx },
                {0, 1, dy },
                {0, 0, 1 }
            };
        }

        internal static double[,] ScaleMatrix(double sx, double sy)
        {
            return new double[3, 3]
            {
                {sx, 0, 0 },
                {0, sy, 0 },
                {0, 0, 1 }
            };
        }

        internal static double[,] RotationMatrix(double theta)
        {
            return new double[3, 3]
            {
                {Math.Cos(theta), -Math.Sin(theta), 0 },
                {Math.Sin(theta), Math.Cos(theta), 0 },
                {0, 0, 1 }
            };
        }

        internal static double[,] Invert(double[,] m)
        {
            double[,] tbr = new double[3, 3];

            tbr[0, 0] = (m[1, 1] * m[2, 2] - m[1, 2] * m[2, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[0, 1] = -(m[0, 1] * m[2, 2] - m[0, 2] * m[2, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[0, 2] = (m[0, 1] * m[1, 2] - m[0, 2] * m[1, 1]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 0] = -(m[1, 0] * m[2, 2] - m[1, 2] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 1] = (m[0, 0] * m[2, 2] - m[0, 2] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[1, 2] = -(m[0, 0] * m[1, 2] - m[0, 2] * m[1, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 0] = (m[1, 0] * m[2, 1] - m[1, 1] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 1] = -(m[0, 0] * m[2, 1] - m[0, 1] * m[2, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);
            tbr[2, 2] = (m[0, 0] * m[1, 1] - m[0, 1] * m[1, 0]) / (m[0, 0] * m[1, 1] * m[2, 2] - m[0, 0] * m[1, 2] * m[2, 1] - m[1, 0] * m[0, 1] * m[2, 2] + m[2, 0] * m[0, 1] * m[1, 2] + m[1, 0] * m[0, 2] * m[2, 1] - m[2, 0] * m[0, 2] * m[1, 1]);

            return tbr;
        }

        /// <summary>
        /// Creates a new <see cref="Graphics"/> object in which all the graphics actions have been transformed using an arbitrary transformation function. Raster images are replaced by grey rectangles.
        /// </summary>
        /// <param name="transformationFunction">An arbitrary transformation function.</param>
        /// <param name="linearisationResolution">The resolution that will be used to linearise curve segments.</param>
        /// <returns>A new <see cref="Graphics"/> object in which all graphics actions have been linearised and transformed using the <paramref name="transformationFunction"/>.</returns>
        public Graphics Transform(Func<Point, Point> transformationFunction, double linearisationResolution)
        {
            Graphics destinationGraphics = new Graphics();

            Stack<double[,]> transformMatrix = new Stack<double[,]>();
            double[,] currMatrix = new double[3, 3] { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };

            for (int i = 0; i < this.Actions.Count; i++)
            {
                if (this.Actions[i] is RectangleAction)
```

```
                {
                    RectangleAction rec = this.Actions[i] as RectangleAction;

                    GraphicsPath rectanglePath = new GraphicsPath();

                    Point pt1 = transformationFunction(Multiply(currMatrix, rec.TopLeft));
                    Point pt2 = transformationFunction(Multiply(currMatrix, new Point(rec.TopLeft.X + rec.Size.Width, rec.TopLeft.Y)));
                    Point pt3 = transformationFunction(Multiply(currMatrix, new Point(rec.TopLeft.X + rec.Size.Width, rec.TopLeft.Y + rec.Size.Height)));
                    Point pt4 = transformationFunction(Multiply(currMatrix, new Point(rec.TopLeft.X, rec.TopLeft.Y + rec.Size.Height)));

                    rectanglePath.MoveTo(pt1).LineTo(pt2).LineTo(pt3).LineTo(pt4).Close();

                    if (rec.Fill != null)
                    {
                        destinationGraphics.FillPath(rectanglePath, rec.Fill, rec.Tag);
                    }
                    else if (rec.Stroke != null)
                    {
                        destinationGraphics.StrokePath(rectanglePath, rec.Stroke, rec.LineWidth, rec.LineCap, rec.LineJoin, rec.LineDash, rec.Tag);
                    }
                }
                else if (this.Actions[i] is PathAction)
                {
                    PathAction pth = this.Actions[i] as PathAction;

                    GraphicsPath newPath = pth.Path.Linearise(linearisationResolution).Transform(pt => transformationFunction(Multiply(currMatrix, pt)));

                    if (pth.IsClipping)
                    {
                        destinationGraphics.SetClippingPath(newPath);
                    }
                    else
                    {
                        if (pth.Fill != null)
                        {
                            destinationGraphics.FillPath(newPath, pth.Fill, pth.Tag);
                        }
                        else if (pth.Stroke != null)
                        {
                            destinationGraphics.StrokePath(newPath, pth.Stroke, pth.LineWidth, pth.LineCap, pth.LineJoin, pth.LineDash, pth.Tag);
                        }
                    }
                }
                else if (this.Actions[i] is TextAction)
                {
                    TextAction txt = this.Actions[i] as TextAction;

                    GraphicsPath textPath = new GraphicsPath().AddText(txt.Origin, txt.Text, txt.Font, txt.TextBaseline).Linearise(linearisationResolution).Transform(pt => transformationFunction(Multiply(currMatrix, pt)));

                    if (txt.Fill != null)
                    {
                        destinationGraphics.FillPath(textPath, txt.Fill, txt.Tag);
                    }
                    else if (txt.Stroke != null)
                    {
                        destinationGraphics.StrokePath(textPath, txt.Stroke, txt.LineWidth, txt.LineCap, txt.LineJoin, txt.LineDash, txt.Tag);
                    }
                }
                else if (this.Actions[i] is TransformAction)
                {
                    TransformAction trf = this.Actions[i] as TransformAction;

                    if (trf.Delta != null)
                    {
                        currMatrix = Multiply(currMatrix, TranslationMatrix(trf.Delta.Value.X, trf.Delta.Value.Y));
                    }
                    else if (trf.Angle != null)
                    {
                        currMatrix = Multiply(currMatrix, RotationMatrix(trf.Angle.Value));
                    }
                    else if (trf.Scale != null)
                    {
                        currMatrix = Multiply(currMatrix, ScaleMatrix(trf.Scale.Value.Width, trf.Scale.Value.Height));
                    }
                    else if (trf.Matrix != null)
                    {
```

```
                        currMatrix = Multiply(currMatrix, trf.Matrix);
                    }
                }
                else if (this.Actions[i] is StateAction)
                {
                    if (((StateAction)this.Actions[i]).StateActionType == StateAction.StateActionTypes.Save)
                    {
                        transformMatrix.Push(currMatrix);
                    }
                    else
                    {
                        currMatrix = transformMatrix.Pop();
                    }
                }
                else if (this.Actions[i] is RasterImageAction)
                {
                    RasterImageAction img = this.Actions[i] as RasterImageAction;

                    GraphicsPath rectanglePath = new GraphicsPath();

                    Point pt1 = transformationFunction(Multiply(currMatrix, new Point(img.DestinationX, img.DestinationY)));
                    Point pt2 = transformationFunction(Multiply(currMatrix, new Point(img.DestinationX + img.DestinationWidth, img.DestinationY)));
                    Point pt3 = transformationFunction(Multiply(currMatrix, new Point(img.DestinationX + img.DestinationWidth, img.DestinationY + img.DestinationHeight)));
                    Point pt4 = transformationFunction(Multiply(currMatrix, new Point(img.DestinationX, img.DestinationY + img.DestinationHeight)));

                    rectanglePath.MoveTo(pt1).LineTo(pt2).LineTo(pt3).LineTo(pt4).Close();

                    destinationGraphics.FillPath(rectanglePath, Colour.FromRgb(220, 220, 220), img.Tag);
                }
            }

            return destinationGraphics;
        }

        internal static GraphicsPath ReduceMaximumLength(GraphicsPath path, double maxLength)
        {
            GraphicsPath shortLinearisedPath = new GraphicsPath();

            // Square of the max length - so that we avoid the square roots.
            double maxLengthSq = maxLength * maxLength;

            Point currPoint = new Point();
            Point startFigurePoint = new Point();

            foreach (Segment seg in path.Segments)
            {
                if (seg is MoveSegment mov)
                {
                    shortLinearisedPath.MoveTo(mov.Point);
                    startFigurePoint = mov.Point;
                    currPoint = mov.Point;
                }
                else if (seg is LineSegment line)
                {
                    double lengthSq = (line.Point.X - currPoint.X) * (line.Point.X - currPoint.X) +
                                      (line.Point.Y - currPoint.Y) * (line.Point.Y - currPoint.Y);

                    if (lengthSq < maxLengthSq)
                    {
                        shortLinearisedPath.LineTo(line.Point);
                    }
                    else
                    {
                        int segmentCount = (int)Math.Ceiling(Math.Sqrt(lengthSq / maxLengthSq));

                        for (int j = 0; j < segmentCount - 1; j++)
                        {
                            Point endPoint = new Point(currPoint.X + (line.Point.X - currPoint.X) * (j + 1) / segmentCount,
                                                       currPoint.Y + (line.Point.Y - currPoint.Y) * (j + 1) / segmentCount);
                            shortLinearisedPath.LineTo(endPoint);
                        }

                        shortLinearisedPath.LineTo(line.Point);
                    }
                    currPoint = line.Point;
                }
                else if (seg is CloseSegment close)
                {
                    double lengthSq = (startFigurePoint.X - currPoint.X) * (startFigurePoint.X -
```

```
currPoint.X) +
                                    (startFigurePoint.Y - currPoint.Y) * (startFigurePoint.Y - currPoint.Y);

                    if (lengthSq < maxLengthSq)
                    {
                        shortLinearisedPath.Close();
                    }
                    else
                    {
                        int segmentCount = (int)Math.Ceiling(Math.Sqrt(lengthSq / maxLengthSq));

                        for (int j = 0; j < segmentCount - 1; j++)
                        {
                            Point endPoint = new Point(currPoint.X + (startFigurePoint.X - currPoint.X) * (j + 1) / segmentCount,
                                                       currPoint.Y + (startFigurePoint.Y - currPoint.Y) * (j + 1) / segmentCount);
                            shortLinearisedPath.LineTo(endPoint);
                        }

                        shortLinearisedPath.Close();
                    }
                    currPoint = startFigurePoint;
                }
            }

            return shortLinearisedPath;
        }

        /// <summary>
        /// Creates a new <see cref="Graphics"/> object in which all the graphics actions have been transformed using an arbitrary transformation function. Raster images are replaced by grey rectangles.
        /// </summary>
        /// <param name="transformationFunction">An arbitrary transformation function.</param>
        /// <param name="linearisationResolution">The resolution that will be used to linearise curve segments.</param>
        /// <param name="maxSegmentLength">The maximum length of line segments.</param>
        /// <returns>A new <see cref="Graphics"/> object in which all graphics actions have been linearised and transformed using the <paramref name="transformationFunction"/>.</returns>
        public Graphics Transform(Func<Point, Point> transformationFunction, double linearisationResolution, double maxSegmentLength)
        {
            Graphics destinationGraphics = new Graphics();

            Stack<double[,]> transformMatrix = new Stack<double[,]>();
            double[,] currMatrix = new double[3, 3] { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };

            for (int i = 0; i < this.Actions.Count; i++)
            {
                if (this.Actions[i] is RectangleAction)
                {
                    RectangleAction rec = this.Actions[i] as RectangleAction;

                    GraphicsPath rectanglePath = new GraphicsPath();

                    Point pt1 = rec.TopLeft;
                    Point pt2 = new Point(rec.TopLeft.X + rec.Size.Width, rec.TopLeft.Y);
                    Point pt3 = new Point(rec.TopLeft.X + rec.Size.Width, rec.TopLeft.Y + rec.Size.Height);
                    Point pt4 = new Point(rec.TopLeft.X, rec.TopLeft.Y + rec.Size.Height);

                    rectanglePath.MoveTo(pt1).LineTo(pt2).LineTo(pt3).LineTo(pt4).Close();

                    rectanglePath = ReduceMaximumLength(rectanglePath, maxSegmentLength).Transform(pt => transformationFunction(Multiply(currMatrix, pt)));

                    if (rec.Fill != null)
                    {
                        destinationGraphics.FillPath(rectanglePath, rec.Fill, rec.Tag);
                    }
                    else if (rec.Stroke != null)
                    {
                        destinationGraphics.StrokePath(rectanglePath, rec.Stroke, rec.LineWidth, rec.LineCap, rec.LineJoin, rec.LineDash, rec.Tag);
                    }
                }
                else if (this.Actions[i] is PathAction)
                {
                    PathAction pth = this.Actions[i] as PathAction;

                    GraphicsPath newPath = ReduceMaximumLength(pth.Path.Linearise(linearisationResolution), maxSegmentLength).Transform(pt => transformationFunction(Multiply(currMatrix, pt)));

                    if (pth.IsClipping)
```

```
                    {
                        destinationGraphics.SetClippingPath(newPath);
                    }
                    else
                    {
                        if (pth.Fill != null)
                        {
                            destinationGraphics.FillPath(newPath, pth.Fill, pth.Tag);
                        }
                        else if (pth.Stroke != null)
                        {
                            destinationGraphics.StrokePath(newPath, pth.Stroke, pth.LineWidth, pth.LineCap, pth.LineJoin, pth.LineDash, pth.Tag);
                        }
                    }
                }
                else if (this.Actions[i] is TextAction)
                {
                    TextAction txt = this.Actions[i] as TextAction;

                    GraphicsPath textPath = ReduceMaximumLength(new GraphicsPath().AddText(txt.Origin, txt.Text, txt.Font, txt.TextBaseline).Linearise(linearisationResolution), maxSegmentLength).Transform(pt => transformationFunction(Multiply(currMatrix, pt)));

                    if (txt.Fill != null)
                    {
                        destinationGraphics.FillPath(textPath, txt.Fill, txt.Tag);
                    }
                    else if (txt.Stroke != null)
                    {
                        destinationGraphics.StrokePath(textPath, txt.Stroke, txt.LineWidth, txt.LineCap, txt.LineJoin, txt.LineDash, txt.Tag);
                    }
                }
                else if (this.Actions[i] is TransformAction)
                {
                    TransformAction trf = this.Actions[i] as TransformAction;

                    if (trf.Delta != null)
                    {
                        currMatrix = Multiply(currMatrix, TranslationMatrix(trf.Delta.Value.X, trf.Delta.Value.Y));
                    }
                    else if (trf.Angle != null)
                    {
                        currMatrix = Multiply(currMatrix, RotationMatrix(trf.Angle.Value));
                    }
                    else if (trf.Scale != null)
                    {
                        currMatrix = Multiply(currMatrix, ScaleMatrix(trf.Scale.Value.Width, trf.Scale.Value.Height));
                    }
                    else if (trf.Matrix != null)
                    {
                        currMatrix = Multiply(currMatrix, trf.Matrix);
                    }
                }
                else if (this.Actions[i] is StateAction)
                {
                    if (((StateAction)this.Actions[i]).StateActionType == StateAction.StateActionTypes.Save)
                    {
                        transformMatrix.Push(currMatrix);
                    }
                    else
                    {
                        currMatrix = transformMatrix.Pop();
                    }
                }
                else if (this.Actions[i] is RasterImageAction)
                {
                    RasterImageAction img = this.Actions[i] as RasterImageAction;

                    GraphicsPath rectanglePath = new GraphicsPath();

                    Point pt1 = new Point(img.DestinationX, img.DestinationY);
                    Point pt2 = new Point(img.DestinationX + img.DestinationWidth, img.DestinationY);
                    Point pt3 = new Point(img.DestinationX + img.DestinationWidth, img.DestinationY + img.DestinationHeight);
                    Point pt4 = new Point(img.DestinationX, img.DestinationY + img.DestinationHeight);

                    rectanglePath.MoveTo(pt1).LineTo(pt2).LineTo(pt3).LineTo(pt4).Close();

                    rectanglePath = ReduceMaximumLength(rectanglePath, maxSegmentLength).Transform(pt => transformationFunction(Multiply(currMatrix, pt)));

                    destinationGraphics.FillPath(rectanglePath, Colour.FromRgb(220, 220, 220),
```

```
img.Tag);
                }
            }

            return destinationGraphics;
        }

        /// <summary>
        /// Creates a new <see cref="Graphics"/> object by linearising all of the elements of the current instance, i.e. replacing curve segments with series of line segments that approximate them. Raster images are left unchanged.
        /// </summary>
        /// <param name="resolution">The resolution that will be used to linearise curve segments.</param>
        /// <returns>A new <see cref="Graphics"/> object containing the linearised elements.</returns>
        public Graphics Linearise(double resolution)
        {
            Graphics destinationGraphics = new Graphics();

            for (int i = 0; i < this.Actions.Count; i++)
            {
                if (this.Actions[i] is RectangleAction || this.Actions[i] is TransformAction || this.Actions[i] is StateAction || this.Actions[i] is RasterImageAction)
                {
                    destinationGraphics.Actions.Add(this.Actions[i]);
                }
                else if (this.Actions[i] is PathAction)
                {
                    PathAction pth = this.Actions[i] as PathAction;

                    GraphicsPath newPath = pth.Path.Linearise(resolution);

                    if (pth.IsClipping)
                    {
                        destinationGraphics.SetClippingPath(newPath);
                    }
                    else
                    {
                        if (pth.Fill != null)
                        {
                            destinationGraphics.FillPath(newPath, pth.Fill, pth.Tag);
                        }
                        else if (pth.Stroke != null)
                        {
                            destinationGraphics.StrokePath(newPath, pth.Stroke, pth.LineWidth, pth.LineCap, pth.LineJoin, pth.LineDash, pth.Tag);
                        }
                    }
                }
                else if (this.Actions[i] is TextAction)
                {
                    TextAction txt = this.Actions[i] as TextAction;

                    GraphicsPath textPath = new GraphicsPath().AddText(txt.Origin, txt.Text, txt.Font, txt.TextBaseline).Linearise(resolution);

                    if (txt.Fill != null)
                    {
                        destinationGraphics.FillPath(textPath, txt.Fill, txt.Tag);
                    }
                    else if (txt.Stroke != null)
                    {
                        destinationGraphics.StrokePath(textPath, txt.Stroke, txt.LineWidth, txt.LineCap, txt.LineJoin, txt.LineDash, txt.Tag);
                    }
                }
            }

            return destinationGraphics;
        }

        /// <summary>
        /// Computes the rectangular bounds of the region affected by the drawing operations performed on the <see cref="Graphics"/> object.
        /// </summary>
        /// <returns>The smallest rectangle that contains all the elements drawn on the <see cref="Graphics"/>.</returns>
        public Rectangle GetBounds()
        {
            Rectangle tbr = Rectangle.NaN;
            bool initialised = false;

            Stack<Rectangle> clippingPaths = new Stack<Rectangle>();
            Rectangle currClippingPath = Rectangle.NaN;

            Stack<double[,]> transformMatrix = new Stack<double[,]>();
            double[,] currMatrix = new double[3, 3] { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };
```

```

            for (int i = 0; i < this.Actions.Count; i++)
            {
                if (this.Actions[i] is IPrintableAction act)
                {
                    Rectangle bounds = act.GetBounds();

                    double lineThickness = double.IsNaN(act.LineWidth) ? 0 : act.LineWidth;

                    Point pt1 = Multiply(currMatrix, new Point(bounds.Location.X - lineThickness * 0.5, bounds.Location.Y - lineThickness * 0.5));
                    Point pt2 = Multiply(currMatrix, new Point(bounds.Location.X + bounds.Size.Width + lineThickness * 0.5, bounds.Location.Y - lineThickness * 0.5));
                    Point pt3 = Multiply(currMatrix, new Point(bounds.Location.X + bounds.Size.Width + lineThickness * 0.5, bounds.Location.Y + bounds.Size.Height + lineThickness * 0.5));
                    Point pt4 = Multiply(currMatrix, new Point(bounds.Location.X - lineThickness * 0.5, bounds.Location.Y + bounds.Size.Height + lineThickness * 0.5));

                    bounds = Point.Bounds(pt1, pt2, pt3, pt4);

                    if (act is PathAction pth && pth.IsClipping)
                    {
                        currClippingPath = bounds;
                    }
                    else
                    {
                        if (!currClippingPath.IsNaN())
                        {
                            bounds = Rectangle.Intersection(bounds, currClippingPath);
                        }

                        if (!double.IsNaN(bounds.Location.X) && !double.IsNaN(bounds.Location.Y) && !double.IsNaN(bounds.Size.Width) && !double.IsNaN(bounds.Size.Height))
                        {
                            if (!initialised)
                            {
                                tbr = bounds;
                                initialised = true;
                            }
                            else
                            {
                                tbr = Rectangle.Union(tbr, bounds);
                            }
                        }
                    }
                }
                else if (this.Actions[i] is TransformAction)
                {
                    TransformAction trf = this.Actions[i] as TransformAction;

                    if (trf.Delta != null)
                    {
                        currMatrix = Multiply(currMatrix, TranslationMatrix(trf.Delta.Value.X, trf.Delta.Value.Y));
                    }
                    else if (trf.Angle != null)
                    {
                        currMatrix = Multiply(currMatrix, RotationMatrix(trf.Angle.Value));
                    }
                    else if (trf.Scale != null)
                    {
                        currMatrix = Multiply(currMatrix, ScaleMatrix(trf.Scale.Value.Width, trf.Scale.Value.Height));
                    }
                    else if (trf.Matrix != null)
                    {
                        currMatrix = Multiply(currMatrix, trf.Matrix);
                    }
                }
                else if (this.Actions[i] is StateAction)
                {
                    if (((StateAction)this.Actions[i]).StateActionType == StateAction.StateActionTypes.Save)
                    {
                        transformMatrix.Push(currMatrix);
                        clippingPaths.Push(currClippingPath);
                    }
                    else
                    {
                        currMatrix = transformMatrix.Pop();
                        currClippingPath = clippingPaths.Pop();
                    }
                }
            }

            return tbr;
        }
```

```
        /// <summary>
        /// A method that is used to rasterise a region of a <see cref="Graphics"/> object. Set this to <see langword="null"/> if you wish to use the default
        /// rasterisation methods (implemented by either VectSharp.Raster, or VectSharp.Raster.ImageSharp). You will have to provide your own implementation
        /// of this method if neither VectSharp.Raster nor VectSharp.Raster.ImageSharp are referenced by your project. The first argument of this method is
        /// the <see cref="Graphics"/> to be rasterised, the second is a <see cref="Rectangle"/> representing the region to rasterise, the third is a
        /// <see cref="double"/> representing the scale, and the third is a boolean value indicating whether the resulting <see cref="RasterImage"/> should
        /// be interpolated.
        /// </summary>
        public static Func<Graphics, Rectangle, double, bool, RasterImage> RasterisationMethod = null;

        /// <summary>
        /// Tries to rasterise specified region of this <see cref="Graphics"/> object using the default rasterisation method.
        /// </summary>
        /// <param name="region">The region of the <see cref="Graphics"/> to rasterise.</param>
        /// <param name="scale">The scale at which the image is rasterised.</param>
        /// <param name="interpolate">Determines whether the resulting <see cref="RasterImage"/> should be interpolated or not.</param>
        /// <param name="output">When this method returns, this will contain the rasterised image (or <see langword="null"/> if the image could not be rasterised.</param>
        /// <returns><see langword="true"/> if the image could be rasterised; <see langword="false"/> if it could not be rasterised.</returns>
        public bool TryRasterise(Rectangle region, double scale, bool interpolate, out RasterImage output)
        {
            if (RasterisationMethod != null)
            {
                output = RasterisationMethod(this, region, scale, interpolate);
                return true;
            }
            else
            {
                System.Reflection.Assembly raster;

                try
                {
                    raster = System.Reflection.Assembly.Load("VectSharp.Raster");
                    if (raster != null)
                    {
                        System.Reflection.MethodInfo rasteriser = raster.GetType("VectSharp.Raster.Raster").GetMethod("Rasterise", System.Reflection.BindingFlags.Public | System.Reflection.BindingFlags.Static);
                        output = (RasterImage)rasteriser.Invoke(null, new object[] { this, region, scale, interpolate });
                        return true;
                    }
                }
                catch { }

                try
                {
                    raster = System.Reflection.Assembly.Load("VectSharp.Raster.ImageSharp");
                    if (raster != null)
                    {
                        System.Reflection.MethodInfo rasteriser = raster.GetType("VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter").GetMethod("Rasterise", System.Reflection.BindingFlags.Public | System.Reflection.BindingFlags.Static);
                        output = (RasterImage)rasteriser.Invoke(null, new object[] { this, region, scale, interpolate });
                        return true;
                    }
                }
                catch { }


                output = null;
                return false;
            }
        }

        /// <summary>
        /// Removes graphics actions that fall completely outside of the specified <paramref name="region"/>.
        /// </summary>
        /// <param name="region">The area to preserve.</param>
        public void Crop(Rectangle region)
        {
            List<int> itemsToRemove = new List<int>();

            Stack<Rectangle> clippingPaths = new Stack<Rectangle>();
            Rectangle currClippingPath = Rectangle.NaN;
```

```
            Stack<double[,]> transformMatrix = new Stack<double[,]>();
            double[,] currMatrix = new double[3, 3] { { 1, 0, 0 }, { 0, 1, 0 }, { 0, 0, 1 } };

            for (int i = 0; i < this.Actions.Count; i++)
            {
                if (this.Actions[i] is IPrintableAction act)
                {
                    Rectangle bounds = act.GetBounds();

                    double lineThickness = double.IsNaN(act.LineWidth) ? 0 : act.LineWidth;

                    Point pt1 = Multiply(currMatrix, new Point(bounds.Location.X - lineThickness * 0.5, bounds.Location.Y - lineThickness * 0.5));
                    Point pt2 = Multiply(currMatrix, new Point(bounds.Location.X + bounds.Size.Width + lineThickness * 0.5, bounds.Location.Y - lineThickness * 0.5));
                    Point pt3 = Multiply(currMatrix, new Point(bounds.Location.X + bounds.Size.Width + lineThickness * 0.5, bounds.Location.Y + bounds.Size.Height + lineThickness * 0.5));
                    Point pt4 = Multiply(currMatrix, new Point(bounds.Location.X - lineThickness * 0.5, bounds.Location.Y + bounds.Size.Height + lineThickness * 0.5));

                    bounds = Point.Bounds(pt1, pt2, pt3, pt4);

                    if (act is PathAction pth && pth.IsClipping)
                    {
                        currClippingPath = bounds;
                    }
                    else
                    {
                        if (!currClippingPath.IsNaN())
                        {
                            bounds = Rectangle.Intersection(bounds, currClippingPath);
                        }

                        if (double.IsNaN(bounds.Location.X) || double.IsNaN(bounds.Location.Y) || double.IsNaN(bounds.Size.Width) || double.IsNaN(bounds.Size.Height) || Rectangle.Intersection(bounds, region).IsNaN())
                        {
                            itemsToRemove.Add(i);
                        }
                    }
                }
                else if (this.Actions[i] is TransformAction)
                {
                    TransformAction trf = this.Actions[i] as TransformAction;

                    if (trf.Delta != null)
                    {
                        currMatrix = Multiply(currMatrix, TranslationMatrix(trf.Delta.Value.X, trf.Delta.Value.Y));
                    }
                    else if (trf.Angle != null)
                    {
                        currMatrix = Multiply(currMatrix, RotationMatrix(trf.Angle.Value));
                    }
                    else if (trf.Scale != null)
                    {
                        currMatrix = Multiply(currMatrix, ScaleMatrix(trf.Scale.Value.Width, trf.Scale.Value.Height));
                    }
                    else if (trf.Matrix != null)
                    {
                        currMatrix = Multiply(currMatrix, trf.Matrix);
                    }
                }
                else if (this.Actions[i] is StateAction)
                {
                    if (((StateAction)this.Actions[i]).StateActionType == StateAction.StateActionTypes.Save)
                    {
                        transformMatrix.Push(currMatrix);
                        clippingPaths.Push(currClippingPath);
                    }
                    else
                    {
                        currMatrix = transformMatrix.Pop();
                        currClippingPath = clippingPaths.Pop();
                    }
                }
            }

            for (int i = itemsToRemove.Count - 1; i >= 0; i--)
            {
                this.Actions.RemoveAt(itemsToRemove[i]);
            }
        }

```

```
        /// <summary>
        /// Removes graphics actions that fall completely outside of the specified region.
        /// </summary>
        /// <param name="topLeft">The top-left corner of the area to preserve.</param>
        /// <param name="size">The size of the area to preserve.</param>
        public void Crop(Point topLeft, Size size)
        {
            Rectangle region = new Rectangle(topLeft, size);
            this.Crop(region);
        }

        /// <summary>
        /// Gets all the tags that have been defined in the <see cref="Graphics"/>.
        /// </summary>
        /// <returns>An <see cref="IEnumerable{T}"/> of <see cref="string"/>s that, when enumerated, returns all the tags that have been defined in the <see cref="Graphics"/>.</returns>
        public IEnumerable<string> GetTags()
        {
            foreach (IGraphicsAction action in this.Actions)
            {
                if (!string.IsNullOrEmpty(action.Tag))
                {
                    yield return action.Tag;
                }
            }
        }
    }
}
```

8.45 Graphics.Text.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using static System.Net.Mime.MediaTypeNames;

namespace VectSharp
{
    public partial class Graphics
    {
        /// <summary>
        /// Fill a text string.
        /// </summary>
        /// <param name="origin">The text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string to draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="fillColour">The <see cref="Brush"/> to use to fill the text.</param>
        /// <param name="textBaseline">The text baseline (determines what the vertical component of <paramref name="origin"/> represents).</param>
        /// <param name="tag">A tag to identify the filled text.</param>
        public void FillText(Point origin, string text, Font font, Brush fillColour, TextBaselines textBaseline = TextBaselines.Top, string tag = null)
        {
            Actions.Add(new TextAction(origin, text, font, textBaseline, fillColour, null, 0, LineCaps.Butt, LineJoins.Miter, LineDash.SolidLine, tag));

            if (font.Underline != null)
            {
                FillTextUnderline(origin, text, font, fillColour, textBaseline, tag + "@underline");
            }
        }

        /// <summary>
        /// Fill a text string.
        /// </summary>
```

```
        /// <param name="originX">The horizontal coordinate of the text origin.</param>
        /// <param name="originY">The vertical coordinate of the text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string to draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="fillColour">The <see cref="Brush"/> to use to fill the text.</param>
        /// <param name="textBaseline">The text baseline (determines what <paramref name="originY"/> represents).</param>
        /// <param name="tag">A tag to identify the filled text.</param>
        public void FillText(double originX, double originY, string text, Font font, Brush fillColour, TextBaselines textBaseline = TextBaselines.Top, string tag = null)
        {
            Actions.Add(new TextAction(new Point(originX, originY), text, font, textBaseline, fillColour, null, 0, LineCaps.Butt, LineJoins.Miter, LineDash.SolidLine, tag));

            if (font.Underline != null)
            {
                FillTextUnderline(originX, originY, text, font, fillColour, textBaseline, tag + "@underline");
            }
        }

        /// <summary>
        /// Stroke a text string.
        /// </summary>
        /// <param name="origin">The text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string to draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="strokeColour">The <see cref="Brush"/> with which to stroke the text.</param>
        /// <param name="lineWidth">The width of the line with which the text is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the text.</param>
        /// <param name="lineJoin">The line join to use to stroke the text.</param>
        /// <param name="lineDash">The line dash to use to stroke the text.</param>
        /// <param name="textBaseline">The text baseline (determines what the vertical component of <paramref name="origin"/> represents).</param>
        /// <param name="tag">A tag to identify the stroked text.</param>
        public void StrokeText(Point origin, string text, Font font, Brush strokeColour, TextBaselines textBaseline = TextBaselines.Top, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            Actions.Add(new TextAction(origin, text, font, textBaseline, null, strokeColour, lineWidth, lineCap, lineJoin, lineDash ?? LineDash.SolidLine, tag));

            if (font.Underline != null)
            {
                StrokeTextUnderline(origin, text, font, strokeColour, textBaseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@underline");
            }
        }

        /// <summary>
        /// Stroke a text string.
        /// </summary>
        /// <param name="originX">The horizontal coordinate of the text origin.</param>
        /// <param name="originY">The vertical coordinate of the text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string to draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="strokeColour">The <see cref="Brush"/> with which to stroke the text.</param>
        /// <param name="lineWidth">The width of the line with which the text is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the text.</param>
        /// <param name="lineJoin">The line join to use to stroke the text.</param>
        /// <param name="lineDash">The line dash to use to stroke the text.</param>
        /// <param name="textBaseline">The text baseline (determines what <paramref name="originY"/> represents).</param>
        /// <param name="tag">A tag to identify the stroked text.</param>
        public void StrokeText(double originX, double originY, string text, Font font, Brush strokeColour, TextBaselines textBaseline = TextBaselines.Top, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            Actions.Add(new TextAction(new Point(originX, originY), text, font, textBaseline, null, strokeColour, lineWidth, lineCap, lineJoin, lineDash ?? LineDash.SolidLine, tag));

            if (font.Underline != null)
            {
                StrokeTextUnderline(originX, originY, text, font, strokeColour, textBaseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@underline");
            }
        }

        /// <summary>
        /// Fill a text string along a <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="path">The <see cref="GraphicsPath"/> along which the text will flow.</param>
        /// <param name="text">The string to draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="fillColour">The <see cref="Brush"/> to use to fill the text.</param>
```

```
        /// <param name="reference">The (relative) starting point on the path starting from which the text should be drawn (0 is the start of the path, 1 is the end of the path).</param>
        /// <param name="anchor">The anchor in the text string that will correspond to the point specified by the <paramref name="reference"/>.</param>
        /// <param name="textBaseline">The text baseline (determines which the position of the text in relation to the <paramref name="path"/>.</param>
        /// <param name="tag">A tag to identify the filled text.</param>
        public void FillTextOnPath(GraphicsPath path, string text, Font font, Brush fillColour, double reference = 0, TextAnchors anchor = TextAnchors.Left, TextBaselines textBaseline = TextBaselines.Top, string tag = null)
        {
            if (font.Underline != null)
            {
                font = new Font(font.FontFamily, font.FontSize, false);
            }

            double currDelta = 0;
            double pathLength = path.MeasureLength();

            Font.DetailedFontMetrics fullMetrics = font.MeasureTextAdvanced(text);

            switch (anchor)
            {
                case TextAnchors.Left:
                    break;
                case TextAnchors.Center:
                    currDelta = -fullMetrics.Width * 0.5 / pathLength;
                    break;
                case TextAnchors.Right:
                    currDelta = -fullMetrics.Width / pathLength;
                    break;
            }

            Point currentGlyphPlacementDelta = new Point();
            Point currentGlyphAdvanceDelta = new Point();
            Point nextGlyphPlacementDelta = new Point();
            Point nextGlyphAdvanceDelta = new Point();

            for (int i = 0; i < text.Length; i++)
            {
                string c = text.Substring(i, 1);

                if (Font.EnableKerning && i < text.Length - 1)
                {
                    currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                    currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                    nextGlyphAdvanceDelta = new Point();
                    nextGlyphPlacementDelta = new Point();

                    TrueTypeFile.PairKerning kerning = font.FontFamily.TrueTypeFile.Get1000EmKerning(text[i], text[i + 1]);

                    if (kerning != null)
                    {
                        currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                        currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                        nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                        nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                    }
                }

                Font.DetailedFontMetrics metrics = font.MeasureTextAdvanced(c);

                Point origin = path.GetPointAtRelative(reference + currDelta + currentGlyphPlacementDelta.X * font.FontSize / 1000);

                Point tangent = path.GetTangentAtRelative(reference + currDelta + currentGlyphPlacementDelta.X * font.FontSize / 1000 + (metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing) / pathLength * 0.5);

                origin = new Point(origin.X - tangent.Y * currentGlyphPlacementDelta.Y * font.FontSize / 1000, origin.Y + tangent.X * currentGlyphPlacementDelta.Y * font.FontSize / 1000);

                this.Save();

                this.Translate(origin);
                this.Rotate(Math.Atan2(tangent.Y, tangent.X));

                switch (textBaseline)
                {
                    case TextBaselines.Top:
                        if (i > 0)
```

```
                        {
                            this.FillText(new Point(metrics.LeftSideBearing, fullMetrics.Top), c, font, fillColour, textBaseline: TextBaselines.Baseline, tag + "@" + i.ToString());
                        }
                        else
                        {
                            this.FillText(new Point(0, fullMetrics.Top), c, font, fillColour, textBaseline: TextBaselines.Baseline, tag + "@" + i.ToString());
                        }
                        break;
                    case TextBaselines.Baseline:
                        if (i > 0)
                        {
                            this.FillText(new Point(metrics.LeftSideBearing, 0), c, font, fillColour, textBaseline: TextBaselines.Baseline, tag + "@" + i.ToString());
                        }
                        else
                        {
                            this.FillText(new Point(0, 0), c, font, fillColour, textBaseline: TextBaselines.Baseline, tag + "@" + i.ToString());
                        }
                        break;
                    case TextBaselines.Bottom:
                        if (i > 0)
                        {
                            this.FillText(new Point(metrics.LeftSideBearing, fullMetrics.Bottom), c, font, fillColour, textBaseline: TextBaselines.Baseline, tag + "@" + i.ToString());
                        }
                        else
                        {
                            this.FillText(new Point(0, fullMetrics.Bottom), c, font, fillColour, textBaseline: TextBaselines.Baseline, tag + "@" + i.ToString());
                        }
                        break;
                    case TextBaselines.Middle:
                        if (i > 0)
                        {
                            this.FillText(new Point(metrics.LeftSideBearing, fullMetrics.Bottom + fullMetrics.Height / 2), c, font, fillColour, textBaseline: TextBaselines.Baseline, tag + "@" + i.ToString());
                        }
                        else
                        {
                            this.FillText(new Point(0, fullMetrics.Bottom + fullMetrics.Height / 2), c, font, fillColour, textBaseline: TextBaselines.Baseline, tag + "@" + i.ToString());
                        }
                        break;
                }

                this.Restore();

                if (i > 0)
                {
                    currDelta += (metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing + currentGlyphAdvanceDelta.X * font.FontSize / 1000) / pathLength;
                }
                else
                {
                    currDelta += (metrics.Width + metrics.RightSideBearing + currentGlyphAdvanceDelta.X * font.FontSize / 1000) / pathLength;
                }
            }
        }

        /// <summary>
        /// Stroke a text string along a <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="path">The <see cref="GraphicsPath"/> along which the text will flow.</param>
        /// <param name="text">The string to draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="strokeColour">The <see cref="Brush"/> with which to stroke the text.</param>
        /// <param name="lineWidth">The width of the line with which the text is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the text.</param>
        /// <param name="lineJoin">The line join to use to stroke the text.</param>
        /// <param name="lineDash">The line dash to use to stroke the text.</param>
        /// <param name="reference">The (relative) starting point on the path starting from which the text should be drawn (0 is the start of the path, 1 is the end of the path).</param>
        /// <param name="anchor">The anchor in the text string that will correspond to the point specified by the <paramref name="reference"/>.</param>
        /// <param name="textBaseline">The text baseline (determines which the position of the text in relation to the <paramref name="path"/>.</param>
        /// <param name="tag">A tag to identify the stroked text.</param>
        public void StrokeTextOnPath(GraphicsPath path, string text, Font font, Brush strokeColour, double reference = 0, TextAnchors anchor = TextAnchors.Left, TextBaselines textBaseline = TextBaselines.Top, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
```

```
            if (font.Underline != null)
            {
                font = new Font(font.FontFamily, font.FontSize, false);
            }

            double currDelta = 0;
            double pathLength = path.MeasureLength();

            Font.DetailedFontMetrics fullMetrics = font.MeasureTextAdvanced(text);

            switch (anchor)
            {
                case TextAnchors.Left:
                    break;
                case TextAnchors.Center:
                    currDelta = -fullMetrics.Width * 0.5 / pathLength;
                    break;
                case TextAnchors.Right:
                    currDelta = -fullMetrics.Width / pathLength;
                    break;
            }

            Point currentGlyphPlacementDelta = new Point();
            Point currentGlyphAdvanceDelta = new Point();
            Point nextGlyphPlacementDelta = new Point();
            Point nextGlyphAdvanceDelta = new Point();

            for (int i = 0; i < text.Length; i++)
            {
                string c = text.Substring(i, 1);

                if (Font.EnableKerning && i < text.Length - 1)
                {
                    currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                    currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                    nextGlyphAdvanceDelta = new Point();
                    nextGlyphPlacementDelta = new Point();

                    TrueTypeFile.PairKerning kerning = font.FontFamily.TrueTypeFile.Get1000EmKerning(text[i], text[i + 1]);

                    if (kerning != null)
                    {
                        currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                        currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                        nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                        nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                    }
                }

                Font.DetailedFontMetrics metrics = font.MeasureTextAdvanced(c);

                Point origin = path.GetPointAtRelative(reference + currDelta + currentGlyphPlacementDelta.X * font.FontSize / 1000);

                Point tangent = path.GetTangentAtRelative(reference + currDelta + currentGlyphPlacementDelta.X * font.FontSize / 1000 + (metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing) / pathLength * 0.5);

                origin = new Point(origin.X - tangent.Y * currentGlyphPlacementDelta.Y * font.FontSize / 1000, origin.Y + tangent.X * currentGlyphPlacementDelta.Y * font.FontSize / 1000);

                this.Save();

                this.Translate(origin);
                this.Rotate(Math.Atan2(tangent.Y, tangent.X));

                switch (textBaseline)
                {
                    case TextBaselines.Top:
                        if (i > 0)
                        {
                            this.StrokeText(new Point(metrics.LeftSideBearing, fullMetrics.Top), c, font, strokeColour, textBaseline: TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                        }
                        else
                        {
                            this.StrokeText(new Point(0, fullMetrics.Top), c, font, strokeColour, textBaseline: TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                        }
```

```
                        break;
                    case TextBaselines.Baseline:
                        if (i > 0)
                        {
                            this.StrokeText(new Point(metrics.LeftSideBearing, 0), c, font, strokeColour, textBaseline: TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                        }
                        else
                        {
                            this.StrokeText(new Point(0, 0), c, font, strokeColour, textBaseline: TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                        }
                        break;
                    case TextBaselines.Bottom:
                        if (i > 0)
                        {
                            this.StrokeText(new Point(metrics.LeftSideBearing, fullMetrics.Bottom), c, font, strokeColour, textBaseline: TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                        }
                        else
                        {
                            this.StrokeText(new Point(0, fullMetrics.Bottom), c, font, strokeColour, textBaseline: TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                        }
                        break;
                    case TextBaselines.Middle:
                        if (i > 0)
                        {
                            this.StrokeText(new Point(metrics.LeftSideBearing, fullMetrics.Bottom + fullMetrics.Height / 2), c, font, strokeColour, textBaseline: TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                        }
                        else
                        {
                            this.StrokeText(new Point(0, fullMetrics.Bottom + fullMetrics.Height / 2), c, font, strokeColour, textBaseline: TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                        }
                        break;
                }

                this.Restore();

                if (i > 0)
                {
                    currDelta += (metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing + currentGlyphAdvanceDelta.X * font.FontSize / 1000) / pathLength;
                }
                else
                {
                    currDelta += (metrics.Width + metrics.RightSideBearing + currentGlyphAdvanceDelta.X * font.FontSize / 1000) / pathLength;
                }
            }
        }

        /// <summary>
        /// Fill a formatted text string.
        /// </summary>
        /// <param name="origin">The text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The <see cref="FormattedText"/> to draw.</param>
        /// <param name="fillColour">The default <see cref="Brush"/> to use to fill the text. This can be overridden by each <paramref name="text"/> element.</param>
        /// <param name="textBaseline">The text baseline (determines what the vertical component of <paramref name="origin"/> represents).</param>
        /// <param name="tag">A tag to identify the filled text.</param>
        public void FillText(Point origin, IEnumerable<FormattedText> text, Brush fillColour, TextBaselines textBaseline = TextBaselines.Top, string tag = null)
        {
            List<FormattedText> enumeratedText = new List<FormattedText>();
            List<Font.DetailedFontMetrics> allMetrics = new List<Font.DetailedFontMetrics>();

            Font.DetailedFontMetrics fullMetrics = text.Measure(enumeratedText, allMetrics);

            Point baselineOrigin = origin;

            switch (textBaseline)
            {
                case TextBaselines.Baseline:
                    baselineOrigin = origin;
                    break;
                case TextBaselines.Top:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Top);
                    break;
```

```
                case TextBaselines.Bottom:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Bottom);
                    break;
                case TextBaselines.Middle:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Top * 0.5 + fullMetrics.Bottom * 0.5);
                    break;
            }

            for (int i = 0; i < enumeratedText.Count; i++)
            {
                FormattedText txt = enumeratedText[i];

                if (txt.Script == Script.Normal)
                {
                    Font.DetailedFontMetrics metrics = allMetrics[i];

                    if (i > 0)
                    {
                        FillText(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y, txt.Text, txt.Font, txt.Brush ?? fillColour, TextBaselines.Baseline, tag + "@" + i.ToString());
                    }
                    else
                    {
                        FillText(baselineOrigin, txt.Text, txt.Font, txt.Brush ?? fillColour, TextBaselines.Baseline, tag + "@" + i.ToString());
                    }

                    if (i > 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing, baselineOrigin.Y);
                    }
                    else
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing, baselineOrigin.Y);
                    }
                }
                else
                {
                    Font newFont = new Font(txt.Font.FontFamily, txt.Font.FontSize * 0.7, txt.Font.Underline);

                    Font.DetailedFontMetrics metrics = allMetrics[i];

                    if (i == 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X - metrics.LeftSideBearing, baselineOrigin.Y);
                    }

                    if (txt.Script == Script.Subscript)
                    {
                        FillText(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + txt.Font.FontSize * 0.14, txt.Text, newFont, txt.Brush ?? fillColour, TextBaselines.Baseline, tag + "@" + i.ToString());
                    }
                    else if (txt.Script == Script.Superscript)
                    {
                        FillText(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y - txt.Font.FontSize * 0.33, txt.Text, newFont, txt.Brush ?? fillColour, TextBaselines.Baseline, tag + "@" + i.ToString());
                    }


                    if (i > 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing, baselineOrigin.Y);
                    }
                    else
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing, baselineOrigin.Y);
                    }
                }
            }
        }

        /// <summary>
        /// Fill a formatted text string.
        /// </summary>
        /// <param name="originX">The horizontal coordinate of the text origin.</param>
        /// <param name="originY">The vertical coordinate of the text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The <see cref="FormattedText"/> to draw.</param>
```

```
        /// <param name="fillColour">The default <see cref="Brush"/> to use to fill the text. This can be overridden by each <paramref name="text"/> element.</param>
        /// <param name="textBaseline">The text baseline (determines what <paramref name="originY"/> represents).</param>
        /// <param name="tag">A tag to identify the filled text.</param>
        public void FillText(double originX, double originY, IEnumerable<FormattedText> text, Brush fillColour, TextBaselines textBaseline = TextBaselines.Top, string tag = null)
        {
            FillText(new Point(originX, originY), text, fillColour, textBaseline, tag);
        }

        /// <summary>
        /// Stroke a formatted text string.
        /// </summary>
        /// <param name="origin">The text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The <see cref="FormattedText"/> to draw.</param>
        /// <param name="strokeColour">The default <see cref="Brush"/> with which to stroke the text.</param>
        /// <param name="lineWidth">The width of the line with which the text is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the text.</param>
        /// <param name="lineJoin">The line join to use to stroke the text.</param>
        /// <param name="lineDash">The line dash to use to stroke the text.</param>
        /// <param name="textBaseline">The text baseline (determines what the vertical component of <paramref name="origin"/> represents).</param>
        /// <param name="tag">A tag to identify the stroked text.</param>
        public void StrokeText(Point origin, IEnumerable<FormattedText> text, Brush strokeColour, TextBaselines textBaseline = TextBaselines.Top, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            List<FormattedText> enumeratedText = new List<FormattedText>();
            List<Font.DetailedFontMetrics> allMetrics = new List<Font.DetailedFontMetrics>();

            Font.DetailedFontMetrics fullMetrics = text.Measure(enumeratedText, allMetrics);

            Point baselineOrigin = origin;

            switch (textBaseline)
            {
                case TextBaselines.Baseline:
                    baselineOrigin = origin;
                    break;
                case TextBaselines.Top:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Top);
                    break;
                case TextBaselines.Bottom:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Bottom);
                    break;
                case TextBaselines.Middle:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Top * 0.5 + fullMetrics.Bottom * 0.5);
                    break;
            }

            for (int i = 0; i < enumeratedText.Count; i++)
            {
                FormattedText txt = enumeratedText[i];

                if (txt.Script == Script.Normal)
                {
                    Font.DetailedFontMetrics metrics = allMetrics[i];

                    if (i > 0)
                    {
                        StrokeText(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y, txt.Text, txt.Font, txt.Brush ?? strokeColour, TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                    }
                    else
                    {
                        StrokeText(baselineOrigin, txt.Text, txt.Font, txt.Brush ?? strokeColour, TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                    }

                    if (i > 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing, baselineOrigin.Y);
                    }
                    else
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing, baselineOrigin.Y);
                    }
                }
                else
                {
                    Font newFont = new Font(txt.Font.FontFamily, txt.Font.FontSize * 0.7,
```

```
txt.Font.Underline);

                    Font.DetailedFontMetrics metrics = allMetrics[i];

                    if (i == 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X - metrics.LeftSideBearing, baselineOrigin.Y);
                    }

                    if (txt.Script == Script.Subscript)
                    {
                        StrokeText(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + txt.Font.FontSize * 0.14, txt.Text, newFont, txt.Brush ?? strokeColour, TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                    }
                    else if (txt.Script == Script.Superscript)
                    {
                        StrokeText(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y - txt.Font.FontSize * 0.33, txt.Text, newFont, txt.Brush ?? strokeColour, TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                    }


                    if (i > 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing, baselineOrigin.Y);
                    }
                    else
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing, baselineOrigin.Y);
                    }
                }
            }
        }

        /// <summary>
        /// Stroke a formatted text string.
        /// </summary>
        /// <param name="originX">The horizontal coordinate of the text origin.</param>
        /// <param name="originY">The vertical coordinate of the text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The <see cref="FormattedText"/> to draw.</param>
        /// <param name="strokeColour">The default <see cref="Brush"/> with which to stroke the text.</param>
        /// <param name="lineWidth">The width of the line with which the text is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the text.</param>
        /// <param name="lineJoin">The line join to use to stroke the text.</param>
        /// <param name="lineDash">The line dash to use to stroke the text.</param>
        /// <param name="textBaseline">The text baseline (determines what <paramref name="originY"/> represents).</param>
        /// <param name="tag">A tag to identify the stroked text.</param>
        public void StrokeText(double originX, double originY, IEnumerable<FormattedText> text, Brush strokeColour, TextBaselines textBaseline = TextBaselines.Top, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            StrokeText(new Point(originX, originY), text, strokeColour, textBaseline, lineWidth, lineCap, lineJoin, lineDash, tag);
        }

        /// <summary>
        /// Measure a text string.
        /// See also <seealso cref="Font.MeasureText(string)"/> and <seealso cref="Font.MeasureTextAdvanced(string)"/>.
        /// </summary>
        /// <param name="text">The string to measure.</param>
        /// <param name="font">The font to use to measure the string.</param>
        /// <returns>The size of the measured <paramref name="text"/>.</returns>
        public Size MeasureText(string text, Font font)
        {
            return font.MeasureText(text);
        }

        /// <summary>
        /// Measure a formatted text string.
        /// See also <seealso cref="FormattedTextExtensions.Measure(IEnumerable{FormattedText})"/>.
        /// </summary>
        /// <param name="text">The collection of <see cref="FormattedText"/> objects to measure.</param>
        /// <returns>The size of the measured <paramref name="text"/>.</returns>
        public Size MeasureText(IEnumerable<FormattedText> text)
        {
            Font.DetailedFontMetrics metrics = text.Measure();

            return new Size(metrics.Width, metrics.Height);
```

```
        }

        /// <summary>
        /// Fills the underline of the specified text string.
        /// </summary>
        /// <param name="originX">The horizontal coordinate of the text origin.</param>
        /// <param name="originY">The vertical coordinate of the text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string whose underline will be draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="fillColour">The <see cref="Brush"/> to use to fill the underline.</param>
        /// <param name="textBaseline">The text baseline (determines what <paramref name="originY"/> represents).</param>
        /// <param name="tag">A tag to identify the filled underline.</param>
        public void FillTextUnderline(double originX, double originY, string text, Font font, Brush fillColour, TextBaselines textBaseline = TextBaselines.Top, string tag = null)
        {
            FillTextUnderline(new Point(originX, originY), text, font, fillColour, textBaseline, tag);
        }

        /// <summary>
        /// Fills the underline of the specified text string.
        /// </summary>
        /// <param name="origin">The text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string whose underline will be draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="fillColour">The <see cref="Brush"/> to use to fill the underline.</param>
        /// <param name="textBaseline">The text baseline (determines what the vertical component of <paramref name="origin"/> represents).</param>
        /// <param name="tag">A tag to identify the filled underline.</param>
        public void FillTextUnderline(Point origin, string text, Font font, Brush fillColour, TextBaselines textBaseline = TextBaselines.Top, string tag = null)
        {
            GraphicsPath underline = new GraphicsPath().AddTextUnderline(origin, text, font, textBaseline);
            FillPath(underline, fillColour, tag);
        }

        /// <summary>
        /// Stroke the underline of the specified text string.
        /// </summary>
        /// <param name="originX">The horizontal coordinate of the text origin.</param>
        /// <param name="originY">The vertical coordinate of the text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string whose underline will be drawn.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="strokeColour">The <see cref="Brush"/> with which to stroke the underline.</param>
        /// <param name="lineWidth">The width of the line with which the underline is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the underline.</param>
        /// <param name="lineJoin">The line join to use to stroke the underline.</param>
        /// <param name="lineDash">The line dash to use to stroke the underline.</param>
        /// <param name="textBaseline">The text baseline (determines what <paramref name="originY"/> represents).</param>
        /// <param name="tag">A tag to identify the stroked underline.</param>
        public void StrokeTextUnderline(double originX, double originY, string text, Font font, Brush strokeColour, TextBaselines textBaseline = TextBaselines.Top, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            StrokeTextUnderline(new Point(originX, originY), text, font, strokeColour, textBaseline, lineWidth, lineCap, lineJoin, lineDash, tag);
        }

        /// <summary>
        /// Stroke the underline of the specified text string.
        /// </summary>
        /// <param name="origin">The text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string whose underline will be drawn.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="strokeColour">The <see cref="Brush"/> with which to stroke the underline.</param>
        /// <param name="lineWidth">The width of the line with which the underline is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the underline.</param>
        /// <param name="lineJoin">The line join to use to stroke the underline.</param>
        /// <param name="lineDash">The line dash to use to stroke the underline.</param>
        /// <param name="textBaseline">The text baseline (determines what the vertical component of <paramref name="origin"/> represents).</param>
        /// <param name="tag">A tag to identify the stroked underline.</param>
        public void StrokeTextUnderline(Point origin, string text, Font font, Brush strokeColour, TextBaselines textBaseline = TextBaselines.Top, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            GraphicsPath underline = new GraphicsPath().AddTextUnderline(origin, text, font, textBaseline);
            StrokePath(underline, strokeColour, lineWidth, lineCap, lineJoin, lineDash, tag);
        }

```

```
        /// <summary>
        /// Fill the underline of the specified formatted text string.
        /// </summary>
        /// <param name="originX">The horizontal coordinate of the text origin.</param>
        /// <param name="originY">The vertical coordinate of the text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The <see cref="FormattedText"/> whose underline will be drawn.</param>
        /// <param name="fillColour">The default <see cref="Brush"/> to use to fill the underline. This can be overridden by each <paramref name="text"/> element.</param>
        /// <param name="textBaseline">The text baseline (determines what <paramref name="originY"/> represents).</param>
        /// <param name="tag">A tag to identify the filled underlined.</param>
        public void FillTextUnderline(double originX, double originY, IEnumerable<FormattedText> text, Brush fillColour, TextBaselines textBaseline = TextBaselines.Top, string tag = null)
        {
            FillTextUnderline(new Point(originX, originY), text, fillColour, textBaseline, tag);
        }

        /// <summary>
        /// Fill the underline of the specified formatted text string.
        /// </summary>
        /// <param name="origin">The text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The <see cref="FormattedText"/> whose underline will be drawn.</param>
        /// <param name="fillColour">The default <see cref="Brush"/> to use to fill the underline. This can be overridden by each <paramref name="text"/> element.</param>
        /// <param name="textBaseline">The text baseline (determines what the vertical component of <paramref name="origin"/> represents).</param>
        /// <param name="tag">A tag to identify the filled underlined.</param>
        public void FillTextUnderline(Point origin, IEnumerable<FormattedText> text, Brush fillColour, TextBaselines textBaseline = TextBaselines.Top, string tag = null)
        {
            List<FormattedText> enumeratedText = new List<FormattedText>();
            List<Font.DetailedFontMetrics> allMetrics = new List<Font.DetailedFontMetrics>();

            Font.DetailedFontMetrics fullMetrics = text.Measure(enumeratedText, allMetrics);

            Point baselineOrigin = origin;

            switch (textBaseline)
            {
                case TextBaselines.Baseline:
                    baselineOrigin = origin;
                    break;
                case TextBaselines.Top:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Top);
                    break;
                case TextBaselines.Bottom:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Bottom);
                    break;
                case TextBaselines.Middle:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Top * 0.5 + fullMetrics.Bottom * 0.5);
                    break;
            }

            for (int i = 0; i < enumeratedText.Count; i++)
            {
                FormattedText txt = enumeratedText[i];

                if (txt.Script == Script.Normal)
                {
                    Font.DetailedFontMetrics metrics = allMetrics[i];

                    if (i > 0)
                    {
                        FillTextUnderline(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y, txt.Text, txt.Font, txt.Brush ?? fillColour, TextBaselines.Baseline, tag + "@" + i.ToString());
                    }
                    else
                    {
                        FillTextUnderline(baselineOrigin, txt.Text, txt.Font, txt.Brush ?? fillColour, TextBaselines.Baseline, tag + "@" + i.ToString());
                    }

                    if (i > 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing, baselineOrigin.Y);
                    }
                    else
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing, baselineOrigin.Y);
                    }
                }
```

```
                else
                {
                    Font newFont = new Font(txt.Font.FontFamily, txt.Font.FontSize * 0.7, txt.Font.Underline);

                    Font.DetailedFontMetrics metrics = allMetrics[i];

                    if (i == 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X - metrics.LeftSideBearing, baselineOrigin.Y);
                    }

                    if (txt.Script == Script.Subscript)
                    {
                        FillTextUnderline(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + txt.Font.FontSize * 0.14, txt.Text, newFont, txt.Brush ?? fillColour, TextBaselines.Baseline, tag + "@" + i.ToString());
                    }
                    else if (txt.Script == Script.Superscript)
                    {
                        FillTextUnderline(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y - txt.Font.FontSize * 0.33, txt.Text, newFont, txt.Brush ?? fillColour, TextBaselines.Baseline, tag + "@" + i.ToString());
                    }


                    if (i > 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing, baselineOrigin.Y);
                    }
                    else
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing, baselineOrigin.Y);
                    }
                }
            }
        }

        /// <summary>
        /// Stroke the underline of the specified formatted text string.
        /// </summary>
        /// <param name="originX">The horizontal coordinate of the text origin.</param>
        /// <param name="originY">The vertical coordinate of the text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The <see cref="FormattedText"/> to draw.</param>
        /// <param name="strokeColour">The default <see cref="Brush"/> with which to stroke the underline.</param>
        /// <param name="lineWidth">The width of the line with which the underline is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the underline.</param>
        /// <param name="lineJoin">The line join to use to stroke the underline.</param>
        /// <param name="lineDash">The line dash to use to stroke the underline.</param>
        /// <param name="textBaseline">The text baseline (determines what <paramref name="originY"/> represents).</param>
        /// <param name="tag">A tag to identify the stroked underline.</param>
        public void StrokeTextUnderline(double originX, double originY, IEnumerable<FormattedText> text, Brush strokeColour, TextBaselines textBaseline = TextBaselines.Top, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            StrokeTextUnderline(new Point(originX, originY), text, strokeColour, textBaseline, lineWidth, lineCap, lineJoin, lineDash, tag);
        }

        /// <summary>
        /// Stroke the underline of the specified formatted text string.
        /// </summary>
        /// <param name="origin">The text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The <see cref="FormattedText"/> to draw.</param>
        /// <param name="strokeColour">The default <see cref="Brush"/> with which to stroke the underline.</param>
        /// <param name="lineWidth">The width of the line with which the underline is stroked.</param>
        /// <param name="lineCap">The line cap to use to stroke the underline.</param>
        /// <param name="lineJoin">The line join to use to stroke the underline.</param>
        /// <param name="lineDash">The line dash to use to stroke the underline.</param>
        /// <param name="textBaseline">The text baseline (determines what the vertical component of <paramref name="origin"/> represents).</param>
        /// <param name="tag">A tag to identify the stroked underline.</param>
        public void StrokeTextUnderline(Point origin, IEnumerable<FormattedText> text, Brush strokeColour, TextBaselines textBaseline = TextBaselines.Top, double lineWidth = 1, LineCaps lineCap = LineCaps.Butt, LineJoins lineJoin = LineJoins.Miter, LineDash? lineDash = null, string tag = null)
        {
            List<FormattedText> enumeratedText = new List<FormattedText>();
            List<Font.DetailedFontMetrics> allMetrics = new List<Font.DetailedFontMetrics>();

```

```
            Font.DetailedFontMetrics fullMetrics = text.Measure(enumeratedText, allMetrics);

            Point baselineOrigin = origin;

            switch (textBaseline)
            {
                case TextBaselines.Baseline:
                    baselineOrigin = origin;
                    break;
                case TextBaselines.Top:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Top);
                    break;
                case TextBaselines.Bottom:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Bottom);
                    break;
                case TextBaselines.Middle:
                    baselineOrigin = new Point(origin.X, origin.Y + fullMetrics.Top * 0.5 + fullMetrics.Bottom * 0.5);
                    break;
            }

            for (int i = 0; i < enumeratedText.Count; i++)
            {
                FormattedText txt = enumeratedText[i];

                if (txt.Script == Script.Normal)
                {
                    Font.DetailedFontMetrics metrics = allMetrics[i];

                    if (i > 0)
                    {
                        StrokeTextUnderline(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y, txt.Text, txt.Font, txt.Brush ?? strokeColour, TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                    }
                    else
                    {
                        StrokeTextUnderline(baselineOrigin, txt.Text, txt.Font, txt.Brush ?? strokeColour, TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                    }

                    if (i > 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing, baselineOrigin.Y);
                    }
                    else
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing, baselineOrigin.Y);
                    }
                }
                else
                {
                    Font newFont = new Font(txt.Font.FontFamily, txt.Font.FontSize * 0.7, txt.Font.Underline);

                    Font.DetailedFontMetrics metrics = allMetrics[i];

                    if (i == 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X - metrics.LeftSideBearing, baselineOrigin.Y);
                    }

                    if (txt.Script == Script.Subscript)
                    {
                        StrokeTextUnderline(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + txt.Font.FontSize * 0.14, txt.Text, newFont, txt.Brush ?? strokeColour, TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                    }
                    else if (txt.Script == Script.Superscript)
                    {
                        StrokeTextUnderline(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y - txt.Font.FontSize * 0.33, txt.Text, newFont, txt.Brush ?? strokeColour, TextBaselines.Baseline, lineWidth, lineCap, lineJoin, lineDash, tag + "@" + i.ToString());
                    }


                    if (i > 0)
                    {
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing, baselineOrigin.Y);
                    }
                    else
                    {
```

```
                        baselineOrigin = new Point(baselineOrigin.X + metrics.Width + metrics.RightSideBearing, baselineOrigin.Y);
                    }
                }
            }
        }
    }
}
```

8.46 GraphicsAction.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using VectSharp.Filters;

namespace VectSharp
{
    internal interface IGraphicsAction
    {
        IGraphicsAction ShallowClone();
        string Tag { get; set; }
    }

    internal interface IPrintableAction
    {
        Brush Fill { get; }
        Brush Stroke { get; }
        double LineWidth { get; }
        LineCaps LineCap { get; }
        LineJoins LineJoin { get; }
        LineDash LineDash { get; }
        string Tag { get; set; }
        Rectangle GetBounds();
    }

    internal class TransformAction : IGraphicsAction
    {
        public Point? Delta { get; } = null;

        public double? Angle { get; } = null;

        public Size? Scale { get; } = null;

        public double[,] Matrix { get; } = null;
        public string Tag { get; set; }

        public IGraphicsAction ShallowClone() => (IGraphicsAction)MemberwiseClone();

        public TransformAction(Point delta, string tag)
        {
            this.Delta = delta;
            this.Tag = tag;
        }

        public TransformAction(double angle, string tag)
        {
            this.Angle = angle;
            this.Tag = tag;
        }

        public TransformAction(Size scale, string tag)
        {
            this.Scale = scale;
            this.Tag = tag;
        }

        public TransformAction(double[,] matrix, string tag)
        {
```

```
            this.Matrix = matrix;
            this.Tag = tag;
        }

        public double[,] GetMatrix()
        {
            if (this.Matrix != null)
            {
                return this.Matrix;
            }
            else if (this.Delta != null)
            {
                return Graphics.TranslationMatrix(this.Delta.Value.X, this.Delta.Value.Y);
            }
            else if (this.Angle != null)
            {
                return Graphics.RotationMatrix(this.Angle.Value);
            }
            else if (this.Scale != null)
            {
                return Graphics.ScaleMatrix(this.Scale.Value.Width, this.Scale.Value.Height);
            }
            else
            {
                return null;
            }
        }
    }

    internal class StateAction : IGraphicsAction
    {
        public string Tag { get; set; } = null;

        public IGraphicsAction ShallowClone() => (IGraphicsAction)MemberwiseClone();

        public enum StateActionTypes
        {
            Save, Restore
        }

        public StateActionTypes StateActionType { get; }

        public StateAction(StateActionTypes type)
        {
            this.StateActionType = type;
        }
    }

    internal class TextAction : IGraphicsAction, IPrintableAction
    {
        public IGraphicsAction ShallowClone() => (IGraphicsAction)MemberwiseClone();

        public Brush Fill { get; }
        public Brush Stroke { get; }
        public double LineWidth { get; }
        public LineCaps LineCap { get; }
        public LineJoins LineJoin { get; }
        public LineDash LineDash { get; }
        public string Tag { get; set; }
        public string Text { get; }
        public Point Origin { get; }
        public TextBaselines TextBaseline { get; }
        public Font Font { get; }

        public TextAction(Point origin, string text, Font font, TextBaselines textBaseLine, Brush
    fill, Brush stroke, double lineWidth, LineCaps lineCap, LineJoins lineJoin, LineDash lineDash, string
    tag)
        {
            this.Origin = origin;
            this.Text = text;
            this.Font = font;
            this.TextBaseline = textBaseLine;
            this.Fill = fill;
            this.Stroke = stroke;
            this.LineCap = lineCap;
            this.LineJoin = lineJoin;
            this.LineWidth = lineWidth;
            this.Tag = tag;
            this.LineDash = lineDash;
        }

        public Rectangle GetBounds()
        {
            Font.DetailedFontMetrics metrics = this.Font.MeasureTextAdvanced(this.Text);

            switch (this.TextBaseline)
            {
```

```
                case TextBaselines.Top:
                    return new Rectangle(this.Origin, new Size(metrics.Width, metrics.Height));
                case TextBaselines.Bottom:
                    return new Rectangle(this.Origin.X, this.Origin.Y - metrics.Height, metrics.Width, metrics.Height);
                case TextBaselines.Middle:
                    return new Rectangle(this.Origin.X, this.Origin.Y - metrics.Height * 0.5, metrics.Width, metrics.Height);
                case TextBaselines.Baseline:
                    return new Rectangle(this.Origin.X, this.Origin.Y - metrics.Top, metrics.Width, metrics.Height);
                default:
                    throw new System.ArgumentOutOfRangeException(nameof(TextBaseline), this.TextBaseline, "Invalid text baseline!");
            }
        }
    }

    internal class RectangleAction : IGraphicsAction, IPrintableAction
    {
        public IGraphicsAction ShallowClone() => (IGraphicsAction)MemberwiseClone();
        public Brush Fill { get; }
        public Brush Stroke { get; }
        public double LineWidth { get; }
        public LineCaps LineCap { get; }
        public LineJoins LineJoin { get; }
        public LineDash LineDash { get; }
        public string Tag { get; set; }
        public Point TopLeft { get; }
        public Size Size { get; }

        public RectangleAction(Point topLeft, Size size, Brush fill, Brush stroke, double lineWidth, LineCaps lineCap, LineJoins lineJoin, LineDash lineDash, string tag)
        {
            this.TopLeft = topLeft;
            this.Size = size;
            this.Fill = fill;
            this.Stroke = stroke;
            this.LineCap = lineCap;
            this.LineJoin = lineJoin;
            this.LineWidth = lineWidth;
            this.LineDash = lineDash;
            this.Tag = tag;
        }

        public Rectangle GetBounds()
        {
            return new Rectangle(this.TopLeft, this.Size);
        }
    }

    internal class PathAction : IGraphicsAction, IPrintableAction
    {
        public IGraphicsAction ShallowClone() => (IGraphicsAction)MemberwiseClone();
        public GraphicsPath Path { get; }
        public Brush Fill { get; }
        public Brush Stroke { get; }
        public string Tag { get; set; }
        public double LineWidth { get; }
        public LineCaps LineCap { get; }
        public LineJoins LineJoin { get; }
        public LineDash LineDash { get; }
        public bool IsClipping { get; }
        public Rectangle GetBounds()
        {
            return this.Path.GetBounds();
        }
        public PathAction(GraphicsPath path, Brush fill, Brush stroke, double lineWidth, LineCaps lineCap, LineJoins lineJoin, LineDash lineDash, string tag, bool isClipping)
        {
            this.Path = path;
            this.Fill = fill;
            this.Stroke = stroke;
            this.LineCap = lineCap;
            this.LineJoin = lineJoin;
            this.LineWidth = lineWidth;
            this.LineDash = lineDash;
            this.Tag = tag;
            this.IsClipping = isClipping;
        }
    }

    internal class RasterImageAction : IGraphicsAction, IPrintableAction
    {
        public IGraphicsAction ShallowClone() => (IGraphicsAction)MemberwiseClone();
        public Brush Fill { get; }
        public Brush Stroke { get; }
```

```
        public string Tag { get; set; }
        public double LineWidth { get; }
        public LineCaps LineCap { get; }
        public LineJoins LineJoin { get; }
        public LineDash LineDash { get; }
        public int SourceX { get; }
        public int SourceY { get; }
        public int SourceWidth { get; }
        public int SourceHeight { get; }
        public double DestinationX { get; }
        public double DestinationY { get; }
        public double DestinationWidth { get; }
        public double DestinationHeight { get; }
        public RasterImage Image { get; }

        public RasterImageAction(int sourceX, int sourceY, int sourceWidth, int sourceHeight, double destinationX, double destinationY, double destinationWidth, double destinationHeight, RasterImage image, string tag)
        {
            this.SourceX = sourceX;
            this.SourceY = sourceY;
            this.SourceWidth = sourceWidth;
            this.SourceHeight = sourceHeight;

            this.DestinationX = destinationX;
            this.DestinationY = destinationY;
            this.DestinationWidth = destinationWidth;
            this.DestinationHeight = destinationHeight;

            this.Image = image;
            this.Tag = tag;
        }

        public Rectangle GetBounds()
        {
            return new Rectangle(this.DestinationX, this.DestinationY, this.DestinationWidth, this.DestinationHeight);
        }
    }

    internal class FilteredGraphicsAction : IGraphicsAction, IPrintableAction
    {
        public IGraphicsAction ShallowClone() => (IGraphicsAction)MemberwiseClone();
        public Brush Fill { get; }
        public Brush Stroke { get; }
        public string Tag { get; set; }
        public double LineWidth { get; }
        public LineCaps LineCap { get; }
        public LineJoins LineJoin { get; }
        public LineDash LineDash { get; }
        public int SourceX { get; }
        public int SourceY { get; }
        public int SourceWidth { get; }
        public int SourceHeight { get; }
        public double DestinationX { get; }
        public double DestinationY { get; }
        public double DestinationWidth { get; }
        public double DestinationHeight { get; }
        public RasterImage Image { get; }
        public Graphics Content { get; }
        public IFilter Filter { get; }

        public FilteredGraphicsAction(Graphics graphics, IFilter filter)
        {
            this.Content = graphics;
            this.Filter = filter;
        }

        public Rectangle GetBounds()
        {
            Rectangle bounds = this.Content.GetBounds();

            return new Rectangle(bounds.Location.X - this.Filter.TopLeftMargin.X, bounds.Location.Y - this.Filter.TopLeftMargin.Y, bounds.Size.Width + this.Filter.TopLeftMargin.X + this.Filter.BottomRightMargin.X, bounds.Size.Height + this.Filter.TopLeftMargin.Y + this.Filter.BottomRightMargin.Y);
        }
    }
}
```

8.47 GraphicsPath.cs

```
/*
```

```
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace VectSharp
{
    /// <summary>
    /// Represents a graphics path that can be filled or stroked.
    /// </summary>
    public class GraphicsPath
    {
        /// <summary>
        /// The segments that make up the path.
        /// </summary>
        public List<Segment> Segments { get; set; } = new List<Segment>();


        /// <summary>
        /// Move the current point without tracing a segment from the previous point.
        /// </summary>
        /// <param name="p">The new point.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath MoveTo(Point p)
        {
            cachedLength = double.NaN;
            cachedBounds = Rectangle.NaN;
            Segments.Add(new MoveSegment(p));
            return this;
        }

        /// <summary>
        /// Move the current point without tracing a segment from the previous point.
        /// </summary>
        /// <param name="x">The horizontal coordinate of the new point.</param>
        /// <param name="y">The vertical coordinate of the new point.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath MoveTo(double x, double y)
        {
            MoveTo(new Point(x, y));
            return this;
        }

        /// <summary>
        /// Move the current point and trace a segment from the previous point.
        /// </summary>
        /// <param name="p">The new point.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath LineTo(Point p)
        {
            cachedLength = double.NaN;
            cachedBounds = Rectangle.NaN;

            if (Segments.Count == 0)
            {
                Segments.Add(new MoveSegment(p));
            }
            else
            {
                Segments.Add(new LineSegment(p));
            }
            return this;
        }

        /// <summary>
        /// Move the current point and trace a segment from the previous point.
        /// </summary>
        /// <param name="x">The horizontal coordinate of the new point.</param>
        /// <param name="y">The vertical coordinate of the new point.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath LineTo(double x, double y)
```

```
        {
            LineTo(new Point(x, y));
            return this;
        }

        /// <summary>
        /// Trace an arc segment from a circle with the specified <paramref name="center"/> and <paramref name="radius"/>, starting at <paramref name="startAngle"/> and ending at <paramref name="endAngle"/>.
        /// The current point is updated to the end point of the arc.
        /// </summary>
        /// <param name="center">The center of the arc.</param>
        /// <param name="radius">The radius of the arc.</param>
        /// <param name="startAngle">The start angle (in radians) of the arc.</param>
        /// <param name="endAngle">The end angle (in radians) of the arc.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath Arc(Point center, double radius, double startAngle, double endAngle)
        {
            cachedLength = double.NaN;
            cachedBounds = Rectangle.NaN;

            if (Segments.Count == 0)
            {
                Segments.Add(new MoveSegment(center.X + radius * Math.Cos(startAngle), center.Y + radius * Math.Sin(startAngle)));
            }
            Segments.Add(new ArcSegment(center, radius, startAngle, endAngle));
            return this;
        }

        /// <summary>
        /// Trace an arc segment from a circle with the specified center and <paramref name="radius"/>, starting at <paramref name="startAngle"/> and ending at <paramref name="endAngle"/>.
        /// The current point is updated to the end point of the arc.
        /// </summary>
        /// <param name="centerX">The horizontal coordinate of the center of the arc.</param>
        /// <param name="centerY">The vertical coordinate of the center of the arc.</param>
        /// <param name="radius">The radius of the arc.</param>
        /// <param name="startAngle">The start angle (in radians) of the arc.</param>
        /// <param name="endAngle">The end angle (in radians) of the arc.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath Arc(double centerX, double centerY, double radius, double startAngle, double endAngle)
        {
            Arc(new Point(centerX, centerY), radius, startAngle, endAngle);
            return this;
        }

        /// <summary>
        /// Trace an arc from an ellipse with the specified radii, rotated by <paramref name="axisAngle"/> with respect to the x-axis, starting at the current point and ending at the <paramref name="endPoint"/>.
        /// </summary>
        /// <param name="radiusX">The horizontal radius of the ellipse.</param>
        /// <param name="radiusY">The vertical radius of the ellipse.</param>
        /// <param name="axisAngle">The angle of the horizontal axis of the ellipse with respect to the horizontal axis.</param>
        /// <param name="largeArc">Determines whether the large or the small arc is drawn.</param>
        /// <param name="sweepClockwise">Determines whether the clockwise or anticlockwise arc is drawn.</param>
        /// <param name="endPoint">The end point of the arc.</param>
        /// <returns></returns>
        public GraphicsPath EllipticalArc(double radiusX, double radiusY, double axisAngle, bool largeArc, bool sweepClockwise, Point endPoint)
        {
            if (radiusX == 0 || radiusY == 0)
            {
                return this.LineTo(endPoint);
            }
            else
            {
                radiusX = Math.Abs(radiusX);
                radiusY = Math.Abs(radiusY);
            }

            double x1 = 0;
            double y1 = 0;

            if (this.Segments.Count > 0)
            {
                for (int i = this.Segments.Count - 1; i >= 0; i--)
                {
                    if (this.Segments[i].Type != SegmentType.Close)
                    {
                        x1 = this.Segments[i].Point.X;
                        y1 = this.Segments[i].Point.Y;
                        break;
```

```
                    }
                }
            }

            double x2 = endPoint.X;
            double y2 = endPoint.Y;

            double x1P = Math.Cos(axisAngle) * (x1 - x2) * 0.5 + Math.Sin(axisAngle) * (y1 - y2) * 0.5;

            if (Math.Abs(x1P) < 1e-7)
            {
                x1P = 0;
            }

            double y1P = -Math.Sin(axisAngle) * (x1 - x2) * 0.5 + Math.Cos(axisAngle) * (y1 - y2) * 0.5;

            if (Math.Abs(y1P) < 1e-7)
            {
                y1P = 0;
            }

            double lambda = x1P * x1P / (radiusX * radiusX) + y1P * y1P / (radiusY * radiusY);

            if (lambda > 1)
            {
                double sqrtLambda = Math.Sqrt(lambda);
                radiusX *= sqrtLambda;
                radiusY *= sqrtLambda;
            }

            double sqrtTerm = (largeArc != sweepClockwise ? 1 : -1) * Math.Sqrt(Math.Max(0, (radiusX * radiusX * radiusY * radiusY - radiusX * radiusX * y1P * y1P - radiusY * radiusY * x1P * x1P) / (radiusX * radiusX * y1P * y1P + radiusY * radiusY * x1P * x1P)));

            double cXP = sqrtTerm * radiusX * y1P / radiusY;
            double cYP = -sqrtTerm * radiusY * x1P / radiusX;

            double cX = Math.Cos(axisAngle) * cXP - Math.Sin(axisAngle) * cYP + (x1 + x2) * 0.5;
            double cY = Math.Sin(axisAngle) * cXP + Math.Cos(axisAngle) * cYP + (y1 + y2) * 0.5;

            double theta1 = AngleVectors(1, 0, (x1P - cXP) / radiusX, (y1P - cYP) / radiusY);
            double deltaTheta = AngleVectors((x1P - cXP) / radiusX, (y1P - cYP) / radiusY, (-x1P - cXP) / radiusX, (-y1P - cYP) / radiusY) % (2 * Math.PI);

            if (!sweepClockwise && deltaTheta > 0)
            {
                deltaTheta -= 2 * Math.PI;
            }
            else if (sweepClockwise && deltaTheta < 0)
            {
                deltaTheta += 2 * Math.PI;
            }

            double r = Math.Min(radiusX, radiusY);

            ArcSegment arc = new ArcSegment(0, 0, r, theta1, theta1 + deltaTheta);

            Segment[] segments = arc.ToBezierSegments();

            for (int i = 0; i < segments.Length; i++)
            {
                for (int j = 0; j < segments[i].Points.Length; j++)
                {
                    double newX = segments[i].Points[j].X * radiusX / r;
                    double newY = segments[i].Points[j].Y * radiusY / r;

                    segments[i].Points[j] = new Point(newX * Math.Cos(axisAngle) - newY * Math.Sin(axisAngle) + cX, newX * Math.Sin(axisAngle) + newY * Math.Cos(axisAngle) + cY);
                }
            }

            cachedLength = double.NaN;
            cachedBounds = Rectangle.NaN;

            this.Segments.AddRange(segments);

            return this;
        }

        private static double AngleVectors(double uX, double uY, double vX, double vY)
        {
            double tbr = Math.Acos((uX * vX + uY * vY) / Math.Sqrt((uX * uX + uY * uY) * (vX * vX + vY * vY)));
            double sign = Math.Sign(uX * vY - uY * vX);
            if (sign != 0)
```

```
            {
                tbr *= sign;
            }
            return tbr;
        }


        /// <summary>
        /// Trace a cubic Bezier curve from the current point to a destination point, with two control points.
        /// The current point is updated to the end point of the Bezier curve.
        /// </summary>
        /// <param name="control1">The first control point.</param>
        /// <param name="control2">The second control point.</param>
        /// <param name="endPoint">The destination point.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath CubicBezierTo(Point control1, Point control2, Point endPoint)
        {
            cachedLength = double.NaN;
            cachedBounds = Rectangle.NaN;

            if (Segments.Count == 0)
            {
                Segments.Add(new MoveSegment(control1));
            }
            Segments.Add(new CubicBezierSegment(control1, control2, endPoint));
            return this;
        }

        /// <summary>
        /// Trace a cubic Bezier curve from the current point to a destination point, with two control points.
        /// The current point is updated to the end point of the Bezier curve.
        /// </summary>
        /// <param name="control1X">The horizontal coordinate of the first control point.</param>
        /// <param name="control1Y">The vertical coordinate of the first control point.</param>
        /// <param name="control2X">The horizontal coordinate of the second control point.</param>
        /// <param name="control2Y">The vertical coordinate of the second control point.</param>
        /// <param name="endPointX">The horizontal coordinate of the destination point.</param>
        /// <param name="endPointY">The vertical coordinate of the destination point.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath CubicBezierTo(double control1X, double control1Y, double control2X, double control2Y, double endPointX, double endPointY)
        {
            CubicBezierTo(new Point(control1X, control1Y), new Point(control2X, control2Y), new Point(endPointX, endPointY));
            return this;
        }

        /// <summary>
        /// Trace a quadratic Bezier curve from the current point to a destination point, with a single control point.
        /// The current point is updated to the end point of the Bezier curve.
        /// </summary>
        /// <param name="control">The control point.</param>
        /// <param name="endPoint">The destination point.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath QuadraticBezierTo(Point control, Point endPoint)
        {
            Point currentPoint = control;

            if (Segments.Count > 0)
            {
                for (int i = Segments.Count - 1; i >= 0; i--)
                {
                    if (Segments[i].Type != SegmentType.Close)
                    {
                        currentPoint = Segments[i].Point;
                        break;
                    }
                }
            }

            Point control1 = new Point((currentPoint.X + 2 * control.X) / 3, (currentPoint.Y + 2 * control.Y) / 3);
            Point control2 = new Point((endPoint.X + 2 * control.X) / 3, (endPoint.Y + 2 * control.Y) / 3);

            CubicBezierTo(control1, control2, endPoint);

            return this;
        }

        /// <summary>
        /// Trace a quadratic Bezier curve from the current point to a destination point, with a single control point.
        /// The current point is updated to the end point of the Bezier curve.
```

```
        /// </summary>
        /// <param name="controlX">The horizontal coordinate of the control point.</param>
        /// <param name="controlY">The vertical coordinate of the control point.</param>
        /// <param name="endPointX">The horizontal coordinate of the destination point.</param>
        /// <param name="endPointY">The vertical coordinate of the destination point.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath QuadraticBezierTo(double controlX, double controlY, double endPointX, double endPointY)
        {
            QuadraticBezierTo(new Point(controlX, controlY), new Point(endPointX, endPointY));
            return this;
        }

        /// <summary>
        /// Trace a segment from the current point to the start point of the figure and flag the figure as closed.
        /// </summary>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath Close()
        {
            cachedLength = double.NaN;
            cachedBounds = Rectangle.NaN;
            Segments.Add(new CloseSegment());
            return this;
        }

        /// <summary>
        /// Add the contour of a text string to the current path.
        /// </summary>
        /// <param name="originX">The horizontal coordinate of the text origin.</param>
        /// <param name="originY">The vertical coordinate of the text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string to draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="textBaseline">The text baseline (determines what <paramref name="originY"/> represents).</param>
        /// /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath AddText(double originX, double originY, string text, Font font, TextBaselines textBaseline = TextBaselines.Top)
        {
            return AddText(new Point(originX, originY), text, font, textBaseline);
        }

        /// <summary>
        /// Add the contour of a text string to the current path.
        /// </summary>
        /// <param name="origin">The text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string to draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="textBaseline">The text baseline (determines what the vertical component of <paramref name="origin"/> represents).</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath AddText(Point origin, string text, Font font, TextBaselines textBaseline = TextBaselines.Top)
        {
            Font.DetailedFontMetrics metrics = font.MeasureTextAdvanced(text);

            Point baselineOrigin = origin;

            switch (textBaseline)
            {
                case TextBaselines.Baseline:
                    baselineOrigin = new Point(origin.X - metrics.LeftSideBearing, origin.Y);
                    break;
                case TextBaselines.Top:
                    baselineOrigin = new Point(origin.X - metrics.LeftSideBearing, origin.Y + metrics.Top);
                    break;
                case TextBaselines.Bottom:
                    baselineOrigin = new Point(origin.X - metrics.LeftSideBearing, origin.Y + metrics.Bottom);
                    break;
                case TextBaselines.Middle:
                    baselineOrigin = new Point(origin.X - metrics.LeftSideBearing, origin.Y + (metrics.Top - metrics.Bottom) * 0.5 + metrics.Bottom);
                    break;
            }

            Point currentGlyphPlacementDelta = new Point();
            Point currentGlyphAdvanceDelta = new Point();
            Point nextGlyphPlacementDelta = new Point();
            Point nextGlyphAdvanceDelta = new Point();

            for (int i = 0; i < text.Length; i++)
            {
                char c = text[i];

```

```
                if (Font.EnableKerning && i < text.Length - 1)
                {
                    currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                    currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                    nextGlyphAdvanceDelta = new Point();
                    nextGlyphPlacementDelta = new Point();

                    TrueTypeFile.PairKerning kerning = font.FontFamily.TrueTypeFile.Get1000EmKerning(c, text[i + 1]);

                    if (kerning != null)
                    {
                        currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                        currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                        nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                        nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                    }
                }

                TrueTypeFile.TrueTypePoint[][] glyphPaths = font.FontFamily.TrueTypeFile.GetGlyphPath(c, font.FontSize);

                for (int j = 0; j < glyphPaths.Length; j++)
                {
                    for (int k = 0; k < glyphPaths[j].Length; k++)
                    {
                        if (k == 0)
                        {
                            this.MoveTo(glyphPaths[j][k].X + baselineOrigin.X + currentGlyphPlacementDelta.X, -glyphPaths[j][k].Y + baselineOrigin.Y + currentGlyphPlacementDelta.Y);
                        }
                        else
                        {
                            if (glyphPaths[j][k].IsOnCurve)
                            {
                                this.LineTo(glyphPaths[j][k].X + baselineOrigin.X + currentGlyphPlacementDelta.X, -glyphPaths[j][k].Y + baselineOrigin.Y + currentGlyphPlacementDelta.Y);
                            }
                            else
                            {
                                Point startPoint = this.Segments.Last().Point;
                                Point quadCtrl = new Point(glyphPaths[j][k].X + baselineOrigin.X + currentGlyphPlacementDelta.X, -glyphPaths[j][k].Y + baselineOrigin.Y + currentGlyphPlacementDelta.Y);
                                Point endPoint = new Point(glyphPaths[j][k + 1].X + baselineOrigin.X + currentGlyphPlacementDelta.X, -glyphPaths[j][k + 1].Y + baselineOrigin.Y + currentGlyphPlacementDelta.Y);


                                Point ctrl1 = new Point(startPoint.X / 3 + 2 * quadCtrl.X / 3, startPoint.Y / 3 + 2 * quadCtrl.Y / 3);
                                Point ctrl2 = new Point(endPoint.X / 3 + 2 * quadCtrl.X / 3, endPoint.Y / 3 + 2 * quadCtrl.Y / 3);

                                this.CubicBezierTo(ctrl1, ctrl2, endPoint);

                                k++;
                            }
                        }
                    }

                    this.Close();
                }

                baselineOrigin.X += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                baselineOrigin.Y += (currentGlyphAdvanceDelta.Y) * font.FontSize / 1000;
            }
            return this;
        }

        /// <summary>
        /// Add the contour of a text string flowing along a <see cref="GraphicsPath"/> to the current path.
        /// </summary>
        /// <param name="path">The <see cref="GraphicsPath"/> along which the text will flow.</param>
        /// <param name="text">The string to draw.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="reference">The (relative) starting point on the path starting from which the text should be drawn (0 is the start of the path, 1 is the end of the path).</param>
        /// <param name="anchor">The anchor in the text string that will correspond to the point specified by the <paramref name="reference"/>.</param>
        /// <param name="textBaseline">The text baseline (determines which the position of the text in
```

```
relation to the <paramref name="path"/>.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath AddTextOnPath(GraphicsPath path, string text, Font font, double reference = 0, TextAnchors anchor = TextAnchors.Left, TextBaselines textBaseline = TextBaselines.Top)
        {
            cachedLength = double.NaN;
            cachedBounds = Rectangle.NaN;

            double currDelta = 0;
            double pathLength = path.MeasureLength();

            Font.DetailedFontMetrics fullMetrics = font.MeasureTextAdvanced(text);

            switch (anchor)
            {
                case TextAnchors.Left:
                    break;
                case TextAnchors.Center:
                    currDelta = -fullMetrics.Width * 0.5 / pathLength;
                    break;
                case TextAnchors.Right:
                    currDelta = -fullMetrics.Width / pathLength;
                    break;
            }

            Point currentGlyphPlacementDelta = new Point();
            Point currentGlyphAdvanceDelta = new Point();
            Point nextGlyphPlacementDelta = new Point();
            Point nextGlyphAdvanceDelta = new Point();

            for (int i = 0; i < text.Length; i++)
            {
                string c = text.Substring(i, 1);

                if (Font.EnableKerning && i < text.Length - 1)
                {
                    currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                    currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                    nextGlyphAdvanceDelta = new Point();
                    nextGlyphPlacementDelta = new Point();

                    TrueTypeFile.PairKerning kerning = font.FontFamily.TrueTypeFile.Get1000EmKerning(text[i], text[i + 1]);

                    if (kerning != null)
                    {
                        currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                        currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                        nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                        nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                    }
                }

                Font.DetailedFontMetrics metrics = font.MeasureTextAdvanced(c);

                Point origin = path.GetPointAtRelative(reference + currDelta + currentGlyphPlacementDelta.X * font.FontSize / 1000);

                Point tangent = path.GetTangentAtRelative(reference + currDelta + currentGlyphPlacementDelta.X * font.FontSize / 1000 + (metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing) / pathLength * 0.5);

                origin = new Point(origin.X - tangent.Y * currentGlyphPlacementDelta.Y * font.FontSize / 1000, origin.Y + tangent.X * currentGlyphPlacementDelta.Y * font.FontSize / 1000);

                GraphicsPath glyphPath = new GraphicsPath();

                switch (textBaseline)
                {
                    case TextBaselines.Top:
                        if (i > 0)
                        {
                            glyphPath.AddText(new Point(metrics.LeftSideBearing, fullMetrics.Top), c, font, textBaseline: TextBaselines.Baseline);
                        }
                        else
                        {
                            glyphPath.AddText(new Point(0, fullMetrics.Top), c, font, textBaseline: TextBaselines.Baseline);
                        }
                        break;
                    case TextBaselines.Baseline:
```

```
                        if (i > 0)
                        {
                            glyphPath.AddText(new Point(metrics.LeftSideBearing, 0), c, font, textBaseline: TextBaselines.Baseline);
                        }
                        else
                        {
                            glyphPath.AddText(new Point(0, 0), c, font, textBaseline: TextBaselines.Baseline);
                        }
                        break;
                    case TextBaselines.Bottom:
                        if (i > 0)
                        {
                            glyphPath.AddText(new Point(metrics.LeftSideBearing, fullMetrics.Bottom), c, font, textBaseline: TextBaselines.Baseline);
                        }
                        else
                        {
                            glyphPath.AddText(new Point(0, fullMetrics.Bottom), c, font, textBaseline: TextBaselines.Baseline);
                        }
                        break;
                    case TextBaselines.Middle:
                        if (i > 0)
                        {
                            glyphPath.AddText(new Point(metrics.LeftSideBearing, fullMetrics.Bottom + fullMetrics.Height / 2), c, font, textBaseline: TextBaselines.Baseline);
                        }
                        else
                        {
                            glyphPath.AddText(new Point(0, fullMetrics.Bottom + fullMetrics.Height / 2), c, font, textBaseline: TextBaselines.Baseline);
                        }
                        break;
                }

                double angle = Math.Atan2(tangent.Y, tangent.X);

                for (int j = 0; j < glyphPath.Segments.Count; j++)
                {
                    if (glyphPath.Segments[j].Points != null)
                    {
                        for (int k = 0; k < glyphPath.Segments[j].Points.Length; k++)
                        {
                            double newX = glyphPath.Segments[j].Points[k].X * Math.Cos(angle) - glyphPath.Segments[j].Points[k].Y * Math.Sin(angle) + origin.X;
                            double newY = glyphPath.Segments[j].Points[k].X * Math.Sin(angle) + glyphPath.Segments[j].Points[k].Y * Math.Cos(angle) + origin.Y;

                            glyphPath.Segments[j].Points[k] = new Point(newX, newY);
                        }
                    }

                    this.Segments.Add(glyphPath.Segments[j]);
                }

                if (i > 0)
                {
                    currDelta += (metrics.Width + metrics.RightSideBearing + metrics.LeftSideBearing + currentGlyphAdvanceDelta.X * font.FontSize / 1000) / pathLength;
                }
                else
                {
                    currDelta += (metrics.Width + metrics.RightSideBearing + currentGlyphAdvanceDelta.X * font.FontSize / 1000) / pathLength;
                }
            }

            return this;
        }



        /// <summary>
        /// Add the contour of the underline of the specified text string to the current path.
        /// </summary>
        /// <param name="origin">The text origin. See <paramref name="textBaseline"/>.</param>
        /// <param name="text">The string whose underline will be drawn.</param>
        /// <param name="font">The font with which to draw the text.</param>
        /// <param name="textBaseline">The text baseline (determines what the vertical component of <paramref name="origin"/> represents).</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath AddTextUnderline(Point origin, string text, Font font, TextBaselines textBaseline = TextBaselines.Top)
        {
            if (font.Underline == null)
```

```
            {
                return this;
            }

            Font.DetailedFontMetrics metrics = font.MeasureTextAdvanced(text);

            double italicAngle = font.FontFamily.TrueTypeFile?.GetItalicAngle() ?? 0;

            if (double.IsNaN(italicAngle))
            {
                italicAngle = 0;
            }

            Point baselineOrigin = origin;

            switch (textBaseline)
            {
                case TextBaselines.Baseline:
                    baselineOrigin = new Point(origin.X - metrics.LeftSideBearing, origin.Y);
                    break;
                case TextBaselines.Top:
                    baselineOrigin = new Point(origin.X - metrics.LeftSideBearing, origin.Y + metrics.Top);
                    break;
                case TextBaselines.Bottom:
                    baselineOrigin = new Point(origin.X - metrics.LeftSideBearing, origin.Y + metrics.Bottom);
                    break;
                case TextBaselines.Middle:
                    baselineOrigin = new Point(origin.X - metrics.LeftSideBearing, origin.Y + (metrics.Top - metrics.Bottom) * 0.5 + metrics.Bottom);
                    break;
            }

            if (!font.Underline.SkipDescenders)
            {
                double italicShift;

                if (!font.Underline.FollowItalicAngle || italicAngle == 0)
                {
                    italicShift = 0;
                }
                else
                {
                    italicShift = font.Underline.Thickness * font.FontSize * Math.Tan(italicAngle / 180.0 * Math.PI);
                }

                if (font.Underline.LineCap == LineCaps.Butt)
                {
                    if (!font.Underline.FollowItalicAngle || italicAngle == 0)
                    {
                        this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.Close();
                    }
                    else
                    {
                        this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.Close();
                    }
                }
                else if (font.Underline.LineCap == LineCaps.Square)
                {
                    if (!font.Underline.FollowItalicAngle || italicAngle == 0)
                    {
                        this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing - font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width + font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position *
```

```
font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width + font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing - font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.Close();
                    }
                    else
                    {
                        this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing - font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width + font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width + font.Underline.Thickness * font.FontSize * 0.5 + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing - font.Underline.Thickness * font.FontSize * 0.5 + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.Close();
                    }
                }
                else if (font.Underline.LineCap == LineCaps.Round)
                {
                    if (!font.Underline.FollowItalicAngle || italicAngle == 0)
                    {
                        this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.Arc(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize * 0.5, font.Underline.Thickness * font.FontSize * 0.5, -Math.PI / 2, Math.PI / 2);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.Arc(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize * 0.5, font.Underline.Thickness * font.FontSize * 0.5, Math.PI / 2, 3 * Math.PI / 2);
                        this.Close();
                    }
                    else
                    {
                        this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing - italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        this.CubicBezierTo(baselineOrigin.X + metrics.LeftSideBearing + metrics.Width + font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize,
                            baselineOrigin.X + metrics.LeftSideBearing + metrics.Width + italicShift + font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize,
                            baselineOrigin.X + metrics.LeftSideBearing + metrics.Width + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);

                        this.LineTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.CubicBezierTo(baselineOrigin.X + metrics.LeftSideBearing - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize,
                            baselineOrigin.X + metrics.LeftSideBearing - italicShift - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize,
                            baselineOrigin.X + metrics.LeftSideBearing - italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize);

                        this.Close();
                    }
                }

                return this;
            }
            else
            {
                if (font.Underline.LineCap == LineCaps.Butt)
                {
                    double italicShift;

                    if (!font.Underline.FollowItalicAngle || italicAngle == 0)
                    {
                        italicShift = 0;
                    }
                    else
```

```
                    {
                        italicShift = font.Underline.Thickness * font.FontSize * Math.Tan(italicAngle / 180.0 * Math.PI);
                    }

                    bool started = false;

                    double currX = baselineOrigin.X;
                    double underlineStartX = baselineOrigin.X + metrics.LeftSideBearing;
                    double currUnderlineX = underlineStartX - metrics.LeftSideBearing;

                    Point currentGlyphPlacementDelta = new Point();
                    Point currentGlyphAdvanceDelta = new Point();
                    Point nextGlyphPlacementDelta = new Point();
                    Point nextGlyphAdvanceDelta = new Point();

                    for (int i = 0; i < text.Length; i++)
                    {
                        char c = text[i];

                        if (Font.EnableKerning && i < text.Length - 1)
                        {
                            currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                            currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                            nextGlyphAdvanceDelta = new Point();
                            nextGlyphPlacementDelta = new Point();

                            TrueTypeFile.PairKerning kerning = font.FontFamily.TrueTypeFile.Get1000EmKerning(c, text[i + 1]);

                            if (kerning != null)
                            {
                                currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                                currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                                nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                                nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                            }
                        }

                        double[] intersections = font.FontFamily.TrueTypeFile.Get1000EmUnderlineIntersections(c, font.Underline.Position * 1000, font.Underline.Thickness * 1000);

                        if (intersections != null)
                        {
                            intersections[0] = intersections[0] * font.FontSize / 1000;
                            intersections[1] = intersections[1] * font.FontSize / 1000;

                            if (currX + intersections[0] - font.Underline.Thickness * font.FontSize >= underlineStartX)
                            {
                                if (!started)
                                {
                                    started = true;
                                    this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize).LineTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                                }

                                this.LineTo(currX + intersections[0] - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                                this.LineTo(currX + intersections[0] - font.Underline.Thickness * font.FontSize + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                                this.Close();
                            }

                            started = true;

                            this.MoveTo(currX + intersections[1] + font.Underline.Thickness * font.FontSize + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                            this.LineTo(currX + intersections[1] + font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize);

                            underlineStartX = currX + intersections[1] + font.Underline.Thickness * font.FontSize;
                            currUnderlineX = Math.Max(currX + intersections[1] + font.Underline.Thickness * font.FontSize, currX + (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000 - font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(c).RightSideBearing * font.FontSize /
```

```
1000);
                        }
                        else if (i == text.Length - 1)
                        {
                            if (c != ' ')
                            {
                                currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000 - font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(c).RightSideBearing * font.FontSize / 1000;
                            }
                            else
                            {
                                currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                            }
                        }
                        else
                        {
                            currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                        }

                        currX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                    }

                    if (!started)
                    {
                        started = true;
                        this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize).LineTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                    }

                    this.LineTo(currUnderlineX, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                    this.LineTo(currUnderlineX + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                    this.LineTo(underlineStartX + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                    this.Close();
                }
                else if (font.Underline.LineCap == LineCaps.Square)
                {
                    double italicShift;

                    if (!font.Underline.FollowItalicAngle || italicAngle == 0)
                    {
                        italicShift = 0;
                    }
                    else
                    {
                        italicShift = font.Underline.Thickness * font.FontSize * Math.Tan(italicAngle / 180.0 * Math.PI);
                    }

                    bool started = false;

                    double currX = baselineOrigin.X;
                    double underlineStartX = baselineOrigin.X + metrics.LeftSideBearing;
                    double currUnderlineX = underlineStartX - metrics.LeftSideBearing;

                    Point currentGlyphPlacementDelta = new Point();
                    Point currentGlyphAdvanceDelta = new Point();
                    Point nextGlyphPlacementDelta = new Point();
                    Point nextGlyphAdvanceDelta = new Point();

                    for (int i = 0; i < text.Length; i++)
                    {
                        char c = text[i];

                        if (Font.EnableKerning && i < text.Length - 1)
                        {
                            currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                            currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                            nextGlyphAdvanceDelta = new Point();
                            nextGlyphPlacementDelta = new Point();

                            TrueTypeFile.PairKerning kerning = font.FontFamily.TrueTypeFile.Get1000EmKerning(c, text[i + 1]);

                            if (kerning != null)
                            {
                                currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                                currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X +
```

```
kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                                nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                                nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                            }
                        }

                        double[] intersections = font.FontFamily.TrueTypeFile.Get1000EmUnderlineIntersections(c, font.Underline.Position * 1000, font.Underline.Thickness * 1000);

                        if (intersections != null)
                        {
                            intersections[0] = intersections[0] * font.FontSize / 1000;
                            intersections[1] = intersections[1] * font.FontSize / 1000;

                            if (currX + intersections[0] - font.Underline.Thickness * font.FontSize >= underlineStartX)
                            {
                                if (!started)
                                {
                                    started = true;
                                    this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing + italicShift - font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize).LineTo(baselineOrigin.X + metrics.LeftSideBearing - font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                                }
                                this.LineTo(currX + intersections[0] - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                                this.LineTo(currX + intersections[0] - font.Underline.Thickness * font.FontSize + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);

                                this.Close();
                            }

                            started = true;

                            this.MoveTo(currX + intersections[1] + font.Underline.Thickness * font.FontSize + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                            this.LineTo(currX + intersections[1] + font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize);

                            underlineStartX = currX + intersections[1] + font.Underline.Thickness * font.FontSize;
                            currUnderlineX = Math.Max(currX + intersections[1] + font.Underline.Thickness * font.FontSize, currX + (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000 - font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(c).RightSideBearing * font.FontSize / 1000);
                        }
                        else if (i == text.Length - 1)
                        {
                            if (c != ' ')
                            {
                                currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000 - font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(c).RightSideBearing * font.FontSize / 1000;
                            }
                            else
                            {
                                currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                            }
                        }
                        else
                        {
                            currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                        }

                        currX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                    }

                    if (!started)
                    {
                        started = true;
                        this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing + italicShift - font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize).LineTo(baselineOrigin.X + metrics.LeftSideBearing - font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize);
```

```
                    }

                    this.LineTo(currUnderlineX + font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                    this.LineTo(currUnderlineX + italicShift + font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                    this.LineTo(underlineStartX + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                    this.Close();
                }
                else if (font.Underline.LineCap == LineCaps.Round)
                {
                    if (!font.Underline.FollowItalicAngle || italicAngle == 0)
                    {
                        bool started = false;

                        double currX = baselineOrigin.X;
                        double underlineStartX = baselineOrigin.X + metrics.LeftSideBearing;
                        double currUnderlineX = underlineStartX - metrics.LeftSideBearing;

                        Point currentGlyphPlacementDelta = new Point();
                        Point currentGlyphAdvanceDelta = new Point();
                        Point nextGlyphPlacementDelta = new Point();
                        Point nextGlyphAdvanceDelta = new Point();

                        for (int i = 0; i < text.Length; i++)
                        {
                            char c = text[i];

                            if (Font.EnableKerning && i < text.Length - 1)
                            {
                                currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                                currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
                                nextGlyphAdvanceDelta = new Point();
                                nextGlyphPlacementDelta = new Point();

                                TrueTypeFile.PairKerning kerning = font.FontFamily.TrueTypeFile.Get1000EmKerning(c, text[i + 1]);

                                if (kerning != null)
                                {
                                    currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                                    currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                                    nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                                    nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                                }
                            }

                            double[] intersections = font.FontFamily.TrueTypeFile.Get1000EmUnderlineIntersections(c, font.Underline.Position * 1000, font.Underline.Thickness * 1000);

                            if (intersections != null)
                            {
                                intersections[0] = intersections[0] * font.FontSize / 1000;
                                intersections[1] = intersections[1] * font.FontSize / 1000;

                                if (currX + intersections[0] - font.Underline.Thickness * font.FontSize >= underlineStartX)
                                {
                                    if (!started)
                                    {
                                        started = true;
                                        this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                                        this.Arc(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize * 0.5, font.Underline.Thickness * font.FontSize * 0.5, Math.PI / 2, 3 * Math.PI / 2);
                                    }

                                    this.LineTo(currX + intersections[0] - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                                    this.LineTo(currX + intersections[0] - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);

                                    this.Close();
                                }

```

```
                                started = true;

                                this.MoveTo(currX + intersections[1] + font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                                this.LineTo(currX + intersections[1] + font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize);

                                underlineStartX = currX + intersections[1] + font.Underline.Thickness * font.FontSize;
                                currUnderlineX = Math.Max(currX + intersections[1] + font.Underline.Thickness * font.FontSize, currX + (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000 - font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(c).RightSideBearing * font.FontSize / 1000);
                            }
                            else if (i == text.Length - 1)
                            {
                                if (c != ' ')
                                {
                                    currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000 - font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(c).RightSideBearing * font.FontSize / 1000;
                                }
                                else
                                {
                                    currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                                }
                            }
                            else
                            {
                                currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                            }

                            currX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                        }

                        if (!started)
                        {
                            started = true;
                            this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                            this.Arc(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize * 0.5, font.Underline.Thickness * font.FontSize * 0.5, Math.PI / 2, 3 * Math.PI / 2);
                        }

                        this.LineTo(currUnderlineX, baselineOrigin.Y + font.Underline.Position * font.FontSize);

                        this.Arc(currUnderlineX, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize * 0.5, font.Underline.Thickness * font.FontSize * 0.5, -Math.PI / 2, Math.PI / 2);

                        this.LineTo(underlineStartX, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.Close();
                    }
                    else
                    {
                        double italicShift = font.Underline.Thickness * font.FontSize * Math.Tan(italicAngle / 180.0 * Math.PI);

                        bool started = false;

                        double currX = baselineOrigin.X;
                        double underlineStartX = baselineOrigin.X + metrics.LeftSideBearing;
                        double currUnderlineX = underlineStartX - metrics.LeftSideBearing;

                        Point currentGlyphPlacementDelta = new Point();
                        Point currentGlyphAdvanceDelta = new Point();
                        Point nextGlyphPlacementDelta = new Point();
                        Point nextGlyphAdvanceDelta = new Point();

                        for (int i = 0; i < text.Length; i++)
                        {
                            char c = text[i];

                            if (Font.EnableKerning && i < text.Length - 1)
                            {
                                currentGlyphPlacementDelta = nextGlyphPlacementDelta;
                                currentGlyphAdvanceDelta = nextGlyphAdvanceDelta;
```

```
                                nextGlyphAdvanceDelta = new Point();
                                nextGlyphPlacementDelta = new Point();

                                TrueTypeFile.PairKerning kerning = font.FontFamily.TrueTypeFile.Get1000EmKerning(c, text[i + 1]);

                                if (kerning != null)
                                {
                                    currentGlyphPlacementDelta = new Point(currentGlyphPlacementDelta.X + kerning.Glyph1Placement.X, currentGlyphPlacementDelta.Y + kerning.Glyph1Placement.Y);
                                    currentGlyphAdvanceDelta = new Point(currentGlyphAdvanceDelta.X + kerning.Glyph1Advance.X, currentGlyphAdvanceDelta.Y + kerning.Glyph1Advance.Y);

                                    nextGlyphPlacementDelta = new Point(nextGlyphPlacementDelta.X + kerning.Glyph2Placement.X, nextGlyphPlacementDelta.Y + kerning.Glyph2Placement.Y);
                                    nextGlyphAdvanceDelta = new Point(nextGlyphAdvanceDelta.X + kerning.Glyph2Advance.X, nextGlyphAdvanceDelta.Y + kerning.Glyph2Advance.Y);
                                }
                            }

                            double[] intersections = font.FontFamily.TrueTypeFile.Get1000EmUnderlineIntersections(c, font.Underline.Position * 1000, font.Underline.Thickness * 1000);

                            if (intersections != null)
                            {
                                intersections[0] = intersections[0] * font.FontSize / 1000;
                                intersections[1] = intersections[1] * font.FontSize / 1000;

                                if (currX + intersections[0] - font.Underline.Thickness * font.FontSize >= underlineStartX)
                                {
                                    if (!started)
                                    {
                                        started = true;

                                        this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);

                                        this.CubicBezierTo(baselineOrigin.X + metrics.LeftSideBearing - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize,
                                                           baselineOrigin.X + metrics.LeftSideBearing - italicShift - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize,
                                                           baselineOrigin.X + metrics.LeftSideBearing - italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                                    }

                                    this.LineTo(currX + intersections[0] - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                                    this.LineTo(currX + intersections[0] - font.Underline.Thickness * font.FontSize + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);

                                    this.Close();
                                }

                                started = true;

                                this.MoveTo(currX + intersections[1] + font.Underline.Thickness * font.FontSize + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                                this.LineTo(currX + intersections[1] + font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize);

                                underlineStartX = currX + intersections[1] + font.Underline.Thickness * font.FontSize;
                                currUnderlineX = Math.Max(currX + intersections[1] + font.Underline.Thickness * font.FontSize, currX + (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000 - font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(c).RightSideBearing * font.FontSize / 1000);
                            }
                            else if (i == text.Length - 1)
                            {
                                if (c != ' ')
                                {
                                    currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000 - font.FontFamily.TrueTypeFile.Get1000EmGlyphBearings(c).RightSideBearing * font.FontSize / 1000;
                                }
                                else
                                {
                                    currUnderlineX +=
```

```
(font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                                }
                            }
                            else
                            {
                                currUnderlineX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                            }

                            currX += (font.FontFamily.TrueTypeFile.Get1000EmGlyphWidth(c) + currentGlyphAdvanceDelta.X) * font.FontSize / 1000;
                        }

                        if (!started)
                        {
                            started = true;

                            this.MoveTo(baselineOrigin.X + metrics.LeftSideBearing, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);

                            this.CubicBezierTo(baselineOrigin.X + metrics.LeftSideBearing - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize,
                                baselineOrigin.X + metrics.LeftSideBearing - italicShift - font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize,
                                baselineOrigin.X + metrics.LeftSideBearing - italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize);
                        }

                        this.LineTo(currUnderlineX, baselineOrigin.Y + font.Underline.Position * font.FontSize);

                        //this.LineTo(currUnderlineX + italicShift + font.Underline.Thickness * font.FontSize * 0.5, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.CubicBezierTo(currUnderlineX + font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize,
                            currUnderlineX + italicShift + font.Underline.Thickness * font.FontSize, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize,
                            currUnderlineX + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);

                        this.LineTo(underlineStartX + italicShift, baselineOrigin.Y + font.Underline.Position * font.FontSize + font.Underline.Thickness * font.FontSize);
                        this.Close();
                    }
                }
            }
            return this;
        }


        /// <summary>
        /// Adds a smooth spline composed of cubic bezier segments that pass through the specified points.
        /// </summary>
        /// <param name="points">The points through which the spline should pass.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath AddSmoothSpline(params Point[] points)
        {
            if (points.Length == 0)
            {
                return this;
            }
            else if (points.Length == 1)
            {
                return this.LineTo(points[0]);
            }
            else if (points.Length == 2)
            {
                return this.LineTo(points[0]).LineTo(points[1]);
            }

            Point[] smoothedSpline = SmoothSpline.SmoothSplines(points);

            this.LineTo(smoothedSpline[0]);

            for (int i = 1; i < smoothedSpline.Length; i += 3)
            {
                this.CubicBezierTo(smoothedSpline[i], smoothedSpline[i + 1], smoothedSpline[i + 2]);
            }

            return this;
        }

```

```
        /// <summary>
        /// Adds another <see cref="GraphicsPath"/> to the current <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="path">The existing <see cref="GraphicsPath"/> that should be added to the current <see cref="GraphicsPath"/>.</param>
        /// <returns>The <see cref="GraphicsPath"/>, to allow for chained calls.</returns>
        public GraphicsPath AddPath(GraphicsPath path)
        {
            this.Segments.AddRange(path.Segments);
            return this;
        }

        private double cachedLength = double.NaN;

        /// <summary>
        /// Measures the length of the <see cref="GraphicsPath"/>.
        /// </summary>
        /// <returns>The length of the <see cref="GraphicsPath"/></returns>
        public double MeasureLength()
        {
            if (double.IsNaN(cachedLength))
            {
                cachedLength = 0;
                Point currPoint = new Point();
                Point figureStartPoint = new Point();

                for (int i = 0; i < this.Segments.Count; i++)
                {
                    switch (this.Segments[i].Type)
                    {
                        case SegmentType.Move:
                            currPoint = this.Segments[i].Point;
                            figureStartPoint = this.Segments[i].Point;
                            break;
                        case SegmentType.Line:
                            if (i > 0)
                            {
                                cachedLength += this.Segments[i].Measure(currPoint);
                                currPoint = this.Segments[i].Point;
                            }
                            else
                            {
                                currPoint = this.Segments[i].Point;
                                figureStartPoint = this.Segments[i].Point;
                            }
                            break;
                        case SegmentType.Arc:
                            if (i > 0)
                            {
                                cachedLength += this.Segments[i].Measure(currPoint);
                                currPoint = this.Segments[i].Point;
                            }
                            else
                            {
                                ArcSegment seg = (ArcSegment)this.Segments[i];
                                figureStartPoint = new Point(seg.Points[0].X + Math.Cos(seg.StartAngle) * seg.Radius, seg.Points[0].Y + Math.Sin(seg.StartAngle) * seg.Radius);
                                cachedLength += this.Segments[i].Measure(figureStartPoint);
                                currPoint = this.Segments[i].Point;
                            }
                            break;
                        case SegmentType.Close:
                            cachedLength += Math.Sqrt((currPoint.X - figureStartPoint.X) * (currPoint.X - figureStartPoint.X) + (currPoint.Y - figureStartPoint.Y) * (currPoint.Y - figureStartPoint.Y));
                            currPoint = figureStartPoint;
                            break;
                        case SegmentType.CubicBezier:
                            if (i > 0)
                            {
                                cachedLength += this.Segments[i].Measure(currPoint);
                                currPoint = this.Segments[i].Point;
                            }
                            else
                            {
                                currPoint = this.Segments[i].Points[0];
                                figureStartPoint = this.Segments[i].Points[0];
                                cachedLength += this.Segments[i].Measure(currPoint);
                                currPoint = this.Segments[i].Point;
                            }
                            break;
                    }
                }
            }

            return cachedLength;
        }
```

```
        /// <summary>
        /// Gets the point at the relative position specified on the <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="position">The position on the <see cref="GraphicsPath"/> (0 is the start of the path, 1 is the end of the path).</param>
        /// <returns>The point at the specified position.</returns>
        public Point GetPointAtRelative(double position)
        {
            return GetPointAtAbsolute(position * this.MeasureLength());
        }

        /// <summary>
        /// Gets the point at the absolute position specified on the <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="length">The distance to the point from the start of the <see cref="GraphicsPath"/>.</param>
        /// <returns>The point at the specified position.</returns>
        public Point GetPointAtAbsolute(double length)
        {
            double pathLength = this.MeasureLength();

            if (length >= 0 && length <= pathLength)
            {
                double currLen = 0;

                Point currPoint = new Point();
                Point figureStartPoint = new Point();

                for (int i = 0; i < this.Segments.Count; i++)
                {
                    switch (this.Segments[i].Type)
                    {
                        case SegmentType.Move:
                            currPoint = this.Segments[i].Point;
                            figureStartPoint = this.Segments[i].Point;
                            break;
                        case SegmentType.Line:
                            if (i > 0)
                            {
                                double segLength = this.Segments[i].Measure(currPoint);

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return this.Segments[i].GetPointAt(currPoint, pos);
                                }
                            }
                            else
                            {
                                currPoint = this.Segments[i].Point;
                                figureStartPoint = this.Segments[i].Point;
                            }
                            break;
                        case SegmentType.Arc:
                            if (i > 0)
                            {
                                double segLength = this.Segments[i].Measure(currPoint);

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return this.Segments[i].GetPointAt(currPoint, pos);
                                }
                            }
                            else
                            {
                                ArcSegment seg = (ArcSegment)this.Segments[i];
                                figureStartPoint = new Point(seg.Points[0].X + Math.Cos(seg.StartAngle) * seg.Radius, seg.Points[0].Y + Math.Sin(seg.StartAngle) * seg.Radius);
                                currPoint = figureStartPoint;

                                double segLength = this.Segments[i].Measure(currPoint);

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
```

```
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return this.Segments[i].GetPointAt(currPoint, pos);
                                }
                            }
                            break;
                        case SegmentType.Close:
                            {
                                double segLength = Math.Sqrt((currPoint.X - figureStartPoint.X) * (currPoint.X - figureStartPoint.X) + (currPoint.Y - figureStartPoint.Y) * (currPoint.Y - figureStartPoint.Y));

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
                                    currPoint = figureStartPoint;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return new Point(currPoint.X * (1 - pos) + figureStartPoint.X * pos, currPoint.Y * (1 - pos) + figureStartPoint.Y * pos);
                                }
                            }
                            break;
                        case SegmentType.CubicBezier:
                            if (i > 0)
                            {
                                double segLength = this.Segments[i].Measure(currPoint);

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return this.Segments[i].GetPointAt(currPoint, pos);
                                }
                            }
                            else
                            {
                                currPoint = this.Segments[i].Points[0];
                                figureStartPoint = this.Segments[i].Points[0];
                                double segLength = this.Segments[i].Measure(currPoint);

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return this.Segments[i].GetPointAt(currPoint, pos);
                                }
                            }
                            break;
                    }
                }

                throw new InvalidOperationException("Unexpected code path!");
            }
            else if (length > pathLength)
            {
                double currLength = 0;

                Point currPoint = new Point();
                Point figureStartPoint = new Point();

                for (int i = 0; i < this.Segments.Count - 1; i++)
                {
                    switch (this.Segments[i].Type)
                    {
                        case SegmentType.Move:
                            currPoint = this.Segments[i].Point;
                            figureStartPoint = this.Segments[i].Point;
                            break;
                        case SegmentType.Line:
                            if (i > 0)
                            {
                                currLength += this.Segments[i].Measure(currPoint);
                                currPoint = this.Segments[i].Point;
```

```
                            }
                            else
                            {
                                currPoint = this.Segments[i].Point;
                                figureStartPoint = this.Segments[i].Point;
                            }
                            break;
                        case SegmentType.Arc:
                            if (i > 0)
                            {
                                currLength += this.Segments[i].Measure(currPoint);
                                currPoint = this.Segments[i].Point;
                            }
                            else
                            {
                                ArcSegment seg = (ArcSegment)this.Segments[i];
                                figureStartPoint = new Point(seg.Points[0].X + Math.Cos(seg.StartAngle) * seg.Radius, seg.Points[0].Y + Math.Sin(seg.StartAngle) * seg.Radius);
                                currLength += this.Segments[i].Measure(figureStartPoint);
                                currPoint = this.Segments[i].Point;
                            }
                            break;
                        case SegmentType.Close:
                            currLength += Math.Sqrt((currPoint.X - figureStartPoint.X) * (currPoint.X - figureStartPoint.X) + (currPoint.Y - figureStartPoint.Y) * (currPoint.Y - figureStartPoint.Y));
                            currPoint = figureStartPoint;
                            break;
                        case SegmentType.CubicBezier:
                            if (i > 0)
                            {
                                currLength += this.Segments[i].Measure(currPoint);
                                currPoint = this.Segments[i].Point;
                            }
                            else
                            {
                                currPoint = this.Segments[i].Points[0];
                                figureStartPoint = this.Segments[i].Points[0];
                                currLength += this.Segments[i].Measure(currPoint);
                                currPoint = this.Segments[i].Point;
                            }
                            break;
                    }
                }

                switch (this.Segments[this.Segments.Count - 1].Type)
                {
                    case SegmentType.Arc:
                    case SegmentType.CubicBezier:
                    case SegmentType.Line:
                        {
                            double pos = 1 + (length - pathLength) / this.Segments[this.Segments.Count - 1].Measure(currPoint);
                            return this.Segments[this.Segments.Count - 1].GetPointAt(currPoint, pos);
                        }
                    case SegmentType.Move:
                        return currPoint;
                    case SegmentType.Close:
                        return this.GetPointAtAbsolute(length - pathLength);
                }

                throw new InvalidOperationException("Unexpected code path!");
            }
            else
            {
                Point currPoint = new Point();
                Point figureStartPoint = new Point();

                for (int i = 0; i < this.Segments.Count; i++)
                {
                    switch (this.Segments[i].Type)
                    {
                        case SegmentType.Move:
                            currPoint = this.Segments[i].Point;
                            figureStartPoint = this.Segments[i].Point;
                            break;
                        case SegmentType.Line:
                            if (i > 0)
                            {
                                double segLength = this.Segments[i].Measure(currPoint);
                                double pos = length / segLength;
                                return this.Segments[i].GetPointAt(currPoint, pos);
                            }
                            else
                            {
                                currPoint = this.Segments[i].Point;
                                figureStartPoint = this.Segments[i].Point;
                            }
```

```
                            break;
                        case SegmentType.Arc:
                            if (i > 0)
                            {
                                double segLength = this.Segments[i].Measure(currPoint);
                                double pos = length / segLength;
                                return this.Segments[i].GetPointAt(currPoint, pos);
                            }
                            else
                            {
                                ArcSegment seg = (ArcSegment)this.Segments[i];
                                figureStartPoint = new Point(seg.Points[0].X + Math.Cos(seg.StartAngle) * seg.Radius, seg.Points[0].Y + Math.Sin(seg.StartAngle) * seg.Radius);
                                currPoint = figureStartPoint;

                                double segLength = this.Segments[i].Measure(currPoint);
                                double pos = length / segLength;
                                return this.Segments[i].GetPointAt(currPoint, pos);
                            }
                        case SegmentType.Close:
                            {
                                double segLength = Math.Sqrt((currPoint.X - figureStartPoint.X) * (currPoint.X - figureStartPoint.X) + (currPoint.Y - figureStartPoint.Y) * (currPoint.Y - figureStartPoint.Y));
                                double pos = length / segLength;
                                return new Point(currPoint.X * (1 - pos) + figureStartPoint.X * pos, currPoint.Y * (1 - pos) + figureStartPoint.Y * pos);
                            }
                        case SegmentType.CubicBezier:
                            if (i > 0)
                            {
                                double segLength = this.Segments[i].Measure(currPoint);
                                double pos = length / segLength;
                                return this.Segments[i].GetPointAt(currPoint, pos);
                            }
                            else
                            {
                                currPoint = this.Segments[i].Points[0];
                                figureStartPoint = this.Segments[i].Points[0];
                                double segLength = this.Segments[i].Measure(currPoint);
                                double pos = length / segLength;
                                return this.Segments[i].GetPointAt(currPoint, pos);
                            }
                    }
                }

                throw new InvalidOperationException("Unexpected code path!");
            }
        }

        /// <summary>
        /// Gets the tangent to the point at the relative position specified on the <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="position">The position on the <see cref="GraphicsPath"/> (0 is the start of the path, 1 is the end of the path).</param>
        /// <returns>The tangent to the point at the specified position.</returns>
        public Point GetTangentAtRelative(double position)
        {
            return GetTangentAtAbsolute(position * this.MeasureLength());
        }

        /// <summary>
        /// Gets the tangent to the point at the absolute position specified on the <see cref="GraphicsPath"/>.
        /// </summary>
        /// <param name="length">The distance to the point from the start of the <see cref="GraphicsPath"/>.</param>
        /// <returns>The tangent to the point at the specified position.</returns>
        public Point GetTangentAtAbsolute(double length)
        {
            double pathLength = this.MeasureLength();

            if (length >= 0 && length <= pathLength)
            {
                double currLen = 0;

                Point currPoint = new Point();
                Point figureStartPoint = new Point();

                for (int i = 0; i < this.Segments.Count; i++)
                {
                    switch (this.Segments[i].Type)
                    {
                        case SegmentType.Move:
                            currPoint = this.Segments[i].Point;
                            figureStartPoint = this.Segments[i].Point;
```

```
                            break;
                        case SegmentType.Line:
                            if (i > 0)
                            {
                                double segLength = this.Segments[i].Measure(currPoint);

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return this.Segments[i].GetTangentAt(currPoint, pos);
                                }
                            }
                            else
                            {
                                currPoint = this.Segments[i].Point;
                                figureStartPoint = this.Segments[i].Point;
                            }
                            break;
                        case SegmentType.Arc:
                            if (i > 0)
                            {
                                double segLength = this.Segments[i].Measure(currPoint);

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return this.Segments[i].GetTangentAt(currPoint, pos);
                                }
                            }
                            else
                            {
                                ArcSegment seg = (ArcSegment)this.Segments[i];
                                figureStartPoint = new Point(seg.Points[0].X +
    Math.Cos(seg.StartAngle) * seg.Radius, seg.Points[0].Y + Math.Sin(seg.StartAngle) * seg.Radius);
                                currPoint = figureStartPoint;

                                double segLength = this.Segments[i].Measure(currPoint);

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return this.Segments[i].GetTangentAt(currPoint, pos);
                                }
                            }
                            break;
                        case SegmentType.Close:
                            {
                                double segLength = Math.Sqrt((currPoint.X - figureStartPoint.X) *
    (currPoint.X - figureStartPoint.X) + (currPoint.Y - figureStartPoint.Y) * (currPoint.Y -
    figureStartPoint.Y));

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
                                    currPoint = figureStartPoint;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return new Point(figureStartPoint.X - currPoint.X,
    figureStartPoint.Y - currPoint.Y).Normalize();
                                }
                            }
                            break;
                        case SegmentType.CubicBezier:
                            if (i > 0)
                            {
                                double segLength = this.Segments[i].Measure(currPoint);

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
```

```
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return this.Segments[i].GetTangentAt(currPoint, pos);
                                }
                            }
                            else
                            {
                                currPoint = this.Segments[i].Points[0];
                                figureStartPoint = this.Segments[i].Points[0];
                                double segLength = this.Segments[i].Measure(currPoint);

                                if (currLen + segLength < length)
                                {
                                    currLen += segLength;
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    double pos = (length - currLen) / segLength;
                                    return this.Segments[i].GetTangentAt(currPoint, pos);
                                }
                            }
                            break;
                        }
                    }

                    throw new InvalidOperationException("Unexpected code path!");
                }
                else if (length > pathLength)
                {
                    double currLength = 0;

                    Point currPoint = new Point();
                    Point figureStartPoint = new Point();

                    for (int i = 0; i < this.Segments.Count - 1; i++)
                    {
                        switch (this.Segments[i].Type)
                        {
                            case SegmentType.Move:
                                currPoint = this.Segments[i].Point;
                                figureStartPoint = this.Segments[i].Point;
                                break;
                            case SegmentType.Line:
                                if (i > 0)
                                {
                                    currLength += this.Segments[i].Measure(currPoint);
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    currPoint = this.Segments[i].Point;
                                    figureStartPoint = this.Segments[i].Point;
                                }
                                break;
                            case SegmentType.Arc:
                                if (i > 0)
                                {
                                    currLength += this.Segments[i].Measure(currPoint);
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    ArcSegment seg = (ArcSegment)this.Segments[i];
                                    figureStartPoint = new Point(seg.Points[0].X +
Math.Cos(seg.StartAngle) * seg.Radius, seg.Points[0].Y + Math.Sin(seg.StartAngle) * seg.Radius);
                                    currLength += this.Segments[i].Measure(figureStartPoint);
                                    currPoint = this.Segments[i].Point;
                                }
                                break;
                            case SegmentType.Close:
                                currLength += Math.Sqrt((currPoint.X - figureStartPoint.X) * (currPoint.X
- figureStartPoint.X) + (currPoint.Y - figureStartPoint.Y) * (currPoint.Y - figureStartPoint.Y));
                                currPoint = figureStartPoint;
                                break;
                            case SegmentType.CubicBezier:
                                if (i > 0)
                                {
                                    currLength += this.Segments[i].Measure(currPoint);
                                    currPoint = this.Segments[i].Point;
                                }
                                else
                                {
                                    currPoint = this.Segments[i].Points[0];
```

```
                                figureStartPoint = this.Segments[i].Points[0];
                                currLength += this.Segments[i].Measure(currPoint);
                                currPoint = this.Segments[i].Point;
                            }
                            break;
                        }
                    }

                    switch (this.Segments[this.Segments.Count - 1].Type)
                    {
                        case SegmentType.Arc:
                        case SegmentType.CubicBezier:
                        case SegmentType.Line:
                            {
                                double pos = 1 + (length - pathLength) / this.Segments[this.Segments.Count - 1].Measure(currPoint);
                                return this.Segments[this.Segments.Count - 1].GetTangentAt(currPoint, pos);
                            }
                        case SegmentType.Move:
                            return new Point();
                        case SegmentType.Close:
                            return this.GetTangentAtAbsolute(length - pathLength);
                    }

                    throw new InvalidOperationException("Unexpected code path!");
                }
                else
                {
                    Point currPoint = new Point();
                    Point figureStartPoint = new Point();

                    for (int i = 0; i < this.Segments.Count; i++)
                    {
                        switch (this.Segments[i].Type)
                        {
                            case SegmentType.Move:
                                currPoint = this.Segments[i].Point;
                                figureStartPoint = this.Segments[i].Point;
                                break;
                            case SegmentType.Line:
                                if (i > 0)
                                {
                                    double segLength = this.Segments[i].Measure(currPoint);
                                    double pos = length / segLength;
                                    return this.Segments[i].GetTangentAt(currPoint, pos);
                                }
                                else
                                {
                                    currPoint = this.Segments[i].Point;
                                    figureStartPoint = this.Segments[i].Point;
                                }
                                break;
                            case SegmentType.Arc:
                                if (i > 0)
                                {
                                    double segLength = this.Segments[i].Measure(currPoint);
                                    double pos = length / segLength;
                                    return this.Segments[i].GetTangentAt(currPoint, pos);
                                }
                                else
                                {
                                    ArcSegment seg = (ArcSegment)this.Segments[i];
                                    figureStartPoint = new Point(seg.Points[0].X + Math.Cos(seg.StartAngle) * seg.Radius, seg.Points[0].Y + Math.Sin(seg.StartAngle) * seg.Radius);
                                    currPoint = figureStartPoint;

                                    double segLength = this.Segments[i].Measure(currPoint);
                                    double pos = length / segLength;
                                    return this.Segments[i].GetTangentAt(currPoint, pos);
                                }
                            case SegmentType.Close:
                                {
                                    double segLength = Math.Sqrt((currPoint.X - figureStartPoint.X) * (currPoint.X - figureStartPoint.X) + (currPoint.Y - figureStartPoint.Y) * (currPoint.Y - figureStartPoint.Y));
                                    double pos = length / segLength;
                                    return new Point(figureStartPoint.X - currPoint.X, figureStartPoint.Y - currPoint.Y).Normalize();
                                }
                            case SegmentType.CubicBezier:
                                if (i > 0)
                                {
                                    double segLength = this.Segments[i].Measure(currPoint);
                                    double pos = length / segLength;
                                    return this.Segments[i].GetTangentAt(currPoint, pos);
                                }
```

```
                            else
                            {
                                currPoint = this.Segments[i].Points[0];
                                figureStartPoint = this.Segments[i].Points[0];
                                double segLength = this.Segments[i].Measure(currPoint);
                                double pos = length / segLength;
                                return this.Segments[i].GetTangentAt(currPoint, pos);
                            }
                        }
                    }

                    throw new InvalidOperationException("Unexpected code path!");
                }
            }

            /// <summary>
            /// Gets the normal to the point at the absolute position specified on the <see cref="GraphicsPath"/>.
            /// </summary>
            /// <param name="length">The distance to the point from the start of the <see cref="GraphicsPath"/>.</param>
            /// <returns>The normal to the point at the specified position.</returns>
            public Point GetNormalAtAbsolute(double length)
            {
                Point tangent = this.GetTangentAtAbsolute(length);
                return new Point(-tangent.Y, tangent.X);
            }

            /// <summary>
            /// Gets the normal to the point at the relative position specified on the <see cref="GraphicsPath"/>.
            /// </summary>
            /// <param name="position">The position on the <see cref="GraphicsPath"/> (0 is the start of the path, 1 is the end of the path).</param>
            /// <returns>The normal to the point at the specified position.</returns>
            public Point GetNormalAtRelative(double position)
            {
                Point tangent = this.GetTangentAtRelative(position);
                return new Point(-tangent.Y, tangent.X);
            }

            /// <summary>
            /// Linearises a <see cref="GraphicsPath"/>, replacing curve segments with series of line segments that approximate them.
            /// </summary>
            /// <param name="resolution">The absolute length between successive samples in curve segments.</param>
            /// <returns>A <see cref="GraphicsPath"/> composed only of linear segments that approximates the current <see cref="GraphicsPath"/>.</returns>
            public GraphicsPath Linearise(double resolution)
            {
                if (!(resolution > 0))
                {
                    throw new ArgumentOutOfRangeException(nameof(resolution), resolution, "The resolution must be greater than 0!");
                }

                GraphicsPath tbr = new GraphicsPath();

                Point?[] previousPoints = new Point?[this.Segments.Count];

                previousPoints[0] = null;

                for (int i = 0; i < this.Segments.Count - 1; i++)
                {
                    if (this.Segments[i].Type != SegmentType.Close)
                    {
                        previousPoints[i + 1] = this.Segments[i].Point;
                    }
                    else
                    {
                        previousPoints[i + 1] = previousPoints[i];
                    }
                }

                Segment[][] linearisedSegments = new Segment[this.Segments.Count][];

                System.Threading.Tasks.Parallel.For(0, this.Segments.Count, i =>
                {
                    linearisedSegments[i] = this.Segments[i].Linearise(previousPoints[i], resolution).ToArray();
                });

                int total = 0;

                for (int i = 0; i < linearisedSegments.Length; i++)
                {
```

```
                total += linearisedSegments[i].Length;
            }

            tbr.Segments.Capacity = total;

            for (int i = 0; i < linearisedSegments.Length; i++)
            {
                tbr.Segments.AddRange(linearisedSegments[i]);
            }

            return tbr;
        }

        /// <summary>
        /// Discretises a <see cref="GraphicsPath"/>, replacing curve segments with series of line segments that approximate them and ensuring that all line segments are shorter than the specified <paramref name="resolution"/>.
        /// </summary>
        /// <param name="resolution">The maximum length (in absolute units) of line segments in the resulting <see cref="GraphicsPath"/>.</param>
        /// <returns>A <see cref="GraphicsPath"/> composed only of linear segments that are shorter than <paramref name="resolution"/> and approximate the current <see cref="GraphicsPath"/>.</returns>
        public GraphicsPath Discretise(double resolution)
        {
            if (!(resolution > 0))
            {
                throw new ArgumentOutOfRangeException(nameof(resolution), resolution, "The resolution must be greater than 0!");
            }

            GraphicsPath tbr = new GraphicsPath();

            Point? previousPoint = null;

            foreach (Segment seg in this.Segments)
            {
                tbr.Segments.AddRange(seg.Linearise(previousPoint, resolution));

                if (seg.Type != SegmentType.Close)
                {
                    previousPoint = seg.Point;
                }
            }

            return Graphics.ReduceMaximumLength(tbr, resolution);
        }

        /// <summary>
        /// Gets a collection of the end points of all the segments in the <see cref="GraphicsPath"/>, divided by figure.
        /// </summary>
        /// <returns>A collection of the end points of all the segments in the <see cref="GraphicsPath"/>, divided by figure.</returns>
        public IEnumerable<List<Point» GetPoints()
        {
            Point startPoint = new Point();

            List<Point> currFigure = null;
            bool returned = true;

            foreach (Segment seg in this.Segments)
            {
                if (seg.Type != SegmentType.Close)
                {
                    Point currPoint = seg.Point;
                    if (seg.Type == SegmentType.Move)
                    {
                        if (!returned)
                        {
                            yield return currFigure;
                        }

                        startPoint = currPoint;
                        currFigure = new List<Point>();
                        returned = false;
                    }
                    currFigure.Add(currPoint);
                }
                else
                {
                    currFigure.Add(startPoint);
                    yield return currFigure;
                    returned = true;
                }
            }

            if (!returned)
```

```
            {
                yield return currFigure;
            }
        }

        /// <summary>
        /// Gets a collection of all the figures in the <see cref="GraphicsPath"/>, returned as individual <see cref="GraphicsPath"/>s.
        /// </summary>
        /// <returns>A collection of all the figures in the <see cref="GraphicsPath"/>, returned as individual <see cref="GraphicsPath"/>s.</returns>
        public IEnumerable<GraphicsPath> GetFigures()
        {
            GraphicsPath currFigure = null;
            bool returned = true;

            foreach (Segment seg in this.Segments)
            {
                if (seg.Type != SegmentType.Close)
                {
                    Point currPoint = seg.Point;
                    if (seg.Type == SegmentType.Move)
                    {
                        if (!returned)
                        {
                            yield return currFigure;
                        }

                        currFigure = new GraphicsPath();
                        returned = false;
                    }

                    currFigure.Segments.Add(seg);
                }
                else
                {
                    currFigure.Close();
                    yield return currFigure;
                    returned = true;
                }
            }

            if (!returned)
            {
                yield return currFigure;
            }
        }


        /// <summary>
        /// Gets a collection of the tangents at the end point of the segments in which the <see cref="GraphicsPath"/> would be linearised, divided by figure.
        /// </summary>
        /// <param name="resolution">The absolute length between successive samples in curve segments.</param>
        /// <returns>A collection of the tangents at the end point of the segments in which the <see cref="GraphicsPath"/> would be linearised, divided by figure.</returns>
        public IEnumerable<List<Point>> GetLinearisationPointsNormals(double resolution)
        {
            if (!(resolution > 0))
            {
                throw new ArgumentOutOfRangeException(nameof(resolution), resolution, "The resolution must be greater than 0!");
            }

            Point previousPoint = new Point();
            Point startPoint = new Point();

            List<Point> currFigure = null;
            bool returned = true;

            for (int i = 0; i < this.Segments.Count; i++)
            {
                Segment seg = this.Segments[i];

                if (seg.Type != SegmentType.Close)
                {
                    Point currPoint = seg.Point;
                    if (seg.Type == SegmentType.Move)
                    {
                        if (!returned)
                        {
                            yield return currFigure;
                        }

                        startPoint = currPoint;
                        currFigure = new List<Point>();
```

```
                        returned = false;

                        if (i < this.Segments.Count - 1 && this.Segments[i + 1].Type != SegmentType.Move)
                        {
                            Point tangent = this.Segments[i + 1].GetTangentAt(seg.Point, 0);

                            currFigure.Add(new Point(-tangent.Y, tangent.X));
                        }
                        else
                        {
                            currFigure.Add(new Point());
                        }
                    }
                    else
                    {
                        foreach (Point tangent in seg.GetLinearisationTangents(previousPoint, resolution))
                        {
                            currFigure.Add(new Point(-tangent.Y, tangent.X));
                        }
                    }

                    previousPoint = currPoint;
                }
                else
                {
                    Point normal;

                    if (!startPoint.IsEqual(previousPoint, 1e-4))
                    {
                        Point tangent = new Point(startPoint.X - previousPoint.X, startPoint.Y - previousPoint.Y).Normalize();
                        normal = new Point(-tangent.Y, tangent.X);
                    }
                    else
                    {
                        normal = currFigure[currFigure.Count - 1];
                    }

                    currFigure.Add(normal);
                    currFigure[0] = new Point((currFigure[1].X + normal.X) * 0.5, (currFigure[1].Y + normal.Y) * 0.5).Normalize();

                    yield return currFigure;
                    returned = true;
                }
            }

            if (!returned)
            {
                yield return currFigure;
            }

        }


        private enum VertexType
        {
            Start, End, Regular, Split, Merge
        };

        /// <summary>
        /// Divides a <see cref="GraphicsPath"/> into triangles.
        /// </summary>
        /// <param name="resolution">The resolution that will be used to linearise curve segments in the <see cref="GraphicsPath"/>.</param>
        /// <param name="clockwise">If this is <see langword="true"/>, the triangles will have their vertices in a clockwise order, otherwise they will be in anticlockwise order.</param>
        /// <returns>A collection of distinct <see cref="GraphicsPath"/>s, each representing one triangle.</returns>
        public IEnumerable<GraphicsPath> Triangulate(double resolution, bool clockwise)
        {
            double shiftAmount = 0.01 * resolution;

            if (!(resolution > 0))
            {
                throw new ArgumentOutOfRangeException(nameof(resolution), resolution, "The resolution must be greater than 0!");
            }

            GraphicsPath linearisedPath = this.Linearise(resolution);

            List<Point> vertices = new List<Point>();
            List<List<int»> vertexEdges = new List<List<int»();
            List<(int, int)> edges = new List<(int, int)>();
            int lastStartingPoint = -1;
```

```
            int lastSegmentEnd = -1;
            double area = 0;

            foreach (Segment seg in linearisedPath.Segments)
            {
                if (seg is MoveSegment)
                {
                    vertices.Add(seg.Point);
                    vertexEdges.Add(new List<int>(2));
                    lastStartingPoint = vertices.Count - 1;
                    lastSegmentEnd = vertices.Count - 1;
                }
                else if (seg is LineSegment)
                {
                    if (!vertices[lastSegmentEnd].IsEqual(seg.Point, 1e-4))
                    {
                        vertices.Add(seg.Point);
                        vertexEdges.Add(new List<int>(2));
                        edges.Add((lastSegmentEnd, vertices.Count - 1));
                        area += (seg.Point.X - vertices[lastSegmentEnd].X) * (seg.Point.Y + vertices[lastSegmentEnd].Y);
                        vertexEdges[lastSegmentEnd].Add(edges.Count - 1);
                        vertexEdges[vertices.Count - 1].Add(edges.Count - 1);
                        lastSegmentEnd = vertices.Count - 1;
                    }
                }
                else if (seg is CloseSegment)
                {
                    if (!vertices[lastSegmentEnd].IsEqual(vertices[lastStartingPoint], 1e-4))
                    {
                        edges.Add((lastSegmentEnd, lastStartingPoint));
                        area += (vertices[lastStartingPoint].X - vertices[lastSegmentEnd].X) * (vertices[lastStartingPoint].Y + vertices[lastSegmentEnd].Y);
                        vertexEdges[lastSegmentEnd].Add(edges.Count - 1);
                        vertexEdges[lastStartingPoint].Add(edges.Count - 1);
                    }
                    else
                    {
                        vertices.RemoveAt(lastSegmentEnd);
                        vertexEdges.RemoveAt(lastSegmentEnd);

                        for (int i = 0; i < edges.Count; i++)
                        {
                            if (edges[i].Item1 == lastSegmentEnd)
                            {
                                edges[i] = (lastStartingPoint, edges[i].Item2);
                                vertexEdges[lastStartingPoint].Add(i);
                            }
                            else if (edges[i].Item2 == lastSegmentEnd)
                            {
                                edges[i] = (edges[i].Item1, lastStartingPoint);
                                vertexEdges[lastStartingPoint].Add(i);
                            }
                        }
                    }

                    lastStartingPoint = -1;
                    lastSegmentEnd = -1;
                }
            }

            if (vertices.Count < 3)
            {
                yield break;
            }

            bool isAntiClockwise = area > 0;

            int compareVertices(Point a, Point b)
            {
                if (a.Y - b.Y != 0)
                {
                    return Math.Sign(a.Y - b.Y);
                }
                else
                {
                    return Math.Sign(a.X - b.X);
                }
            }

            Dictionary<double, int> yCoordinates = new Dictionary<double, int>();
            Dictionary<double, int> yShiftCount = new Dictionary<double, int>();

            foreach (Point pt in vertices)
            {
                if (yCoordinates.ContainsKey(pt.Y))
                {
```

```
                    yCoordinates[pt.Y]++;
                }
                else
                {
                    yCoordinates[pt.Y] = 1;
                    yShiftCount[pt.Y] = 0;
                }
            }

            HashSet<double> yS = new HashSet<double>(from el in yCoordinates select el.Key);

            for (int i = 0; i < vertices.Count; i++)
            {
                if (yCoordinates[vertices[i].Y] > 1)
                {
                    int shiftCount = yShiftCount[vertices[i].Y];

                    double targetCoordinate;

                    do
                    {
                        shiftCount++;

                        targetCoordinate = vertices[i].Y + (2 * (shiftCount % 2) - 1) * (1 - Math.Pow(0.5, (shiftCount - 1) / 2 + 1)) * shiftAmount;
                    }
                    while (yS.Contains(targetCoordinate));

                    yS.Add(targetCoordinate);
                    yCoordinates[vertices[i].Y]--;
                    yShiftCount[vertices[i].Y] = shiftCount;
                    vertices[i] = new Point(vertices[i].X, targetCoordinate);
                }

            }

            Queue<int> sortedVertices = new Queue<int>(Enumerable.Range(0, vertices.Count).OrderBy(i => vertices[i], Comparer<Point>.Create(compareVertices)));

            VertexType[] vertexTypes = new VertexType[vertices.Count];

            List<(int, int)> exploredEdges = new List<(int, int)>();
            List<int> helpers = new List<int>();
            List<(int, int)> diagonals = new List<(int, int)>();
            int[] nexts = new int[vertices.Count];
            int[] prevs = new int[vertices.Count];

            while (sortedVertices.Count > 0)
            {
                int vertex = sortedVertices.Dequeue();

                Point pt = vertices[vertex];

                (int, int) edge1 = edges[vertexEdges[vertex][0]];
                (int, int) edge2 = edges[vertexEdges[vertex][1]];

                int neighbour1 = edge1.Item1 != vertex ? edge1.Item1 : edge1.Item2;
                int neighbour2 = edge2.Item1 != vertex ? edge2.Item1 : edge2.Item2;

                int minNeighbour = Math.Min(neighbour1, neighbour2);
                int maxNeighbour = Math.Max(neighbour1, neighbour2);

                int prev, next;

                if (vertex - minNeighbour == 1 && maxNeighbour - vertex == 1)
                {
                    prev = minNeighbour;
                    next = maxNeighbour;
                }
                else if ((minNeighbour - vertex == 1 && maxNeighbour - vertex > 1) || vertex - maxNeighbour == 1 && vertex - minNeighbour > 1)
                {
                    prev = maxNeighbour;
                    next = minNeighbour;
                }
                else
                {
                    throw new InvalidOperationException("Could not make sense of the ordering of the vertices!");
                }

                nexts[vertex] = next;
                prevs[vertex] = prev;

                Point prevPoint = vertices[prev];
                Point nextPoint = vertices[next];

```

```
                double angle = Math.Atan2(prevPoint.Y - pt.Y, prevPoint.X - pt.X) - Math.Atan2(nextPoint.Y - pt.Y, nextPoint.X - pt.X);

                if (angle < 0)
                {
                    angle += 2 * Math.PI;
                }

                VertexType vertexType;

                if (prevPoint.Y >= pt.Y && nextPoint.Y >= pt.Y && angle < Math.PI)
                {
                    vertexType = VertexType.Start;
                }
                else if (prevPoint.Y >= pt.Y && nextPoint.Y >= pt.Y && angle > Math.PI)
                {
                    vertexType = VertexType.Split;
                }
                else if (prevPoint.Y <= pt.Y && nextPoint.Y <= pt.Y && angle < Math.PI)
                {
                    vertexType = VertexType.End;
                }
                else if (prevPoint.Y <= pt.Y && nextPoint.Y <= pt.Y && angle > Math.PI)
                {
                    vertexType = VertexType.Merge;
                }
                else
                {
                    vertexType = VertexType.Regular;
                }

                vertexTypes[vertex] = vertexType;

                //gpr.FillText(vertices[vertex], vertex.ToString(), new Font(new FontFamily(FontFamily.StandardFontFamilies.Helvetica), 4), Colours.Orange);

                if (vertexType == VertexType.Start)
                {
                    exploredEdges.Add((prev, vertex));
                    helpers.Add(vertex);

                    //gpr.StrokeRectangle(pt.X - 1, pt.Y - 1, 2, 2, Colours.Green, 0.25);
                }
                else if (vertexType == VertexType.End)
                {
                    int eiM1 = -1;

                    for (int i = exploredEdges.Count - 1; i >= 0; i--)
                    {
                        if (exploredEdges[i].Item1 == vertex && exploredEdges[i].Item2 == next)
                        {
                            eiM1 = i;
                            break;
                        }
                    }

                    if (eiM1 >= 0)
                    {
                        if (vertexTypes[helpers[eiM1]] == VertexType.Merge)
                        {
                            diagonals.Add((helpers[eiM1], vertex));
                        }

                        exploredEdges.RemoveAt(eiM1);
                        helpers.RemoveAt(eiM1);
                    }
                }
                else if (vertexType == VertexType.Split)
                {
                    (int, int) ej = (-1, -1);
                    int ejIndex = -1;

                    double xJ = double.MinValue;

                    for (int i = 0; i < exploredEdges.Count; i++)
                    {
                        if ((vertices[exploredEdges[i].Item1].Y <= pt.Y && vertices[exploredEdges[i].Item2].Y >= pt.Y) || (vertices[exploredEdges[i].Item1].Y >= pt.Y && vertices[exploredEdges[i].Item2].Y <= pt.Y))
                        {
                            double dy = pt.Y - vertices[exploredEdges[i].Item1].Y;
                            double dx = dy * (vertices[exploredEdges[i].Item2].X - vertices[exploredEdges[i].Item1].X) / (vertices[exploredEdges[i].Item2].Y - vertices[exploredEdges[i].Item1].Y);

                            double x = dx + vertices[exploredEdges[i].Item1].X;

```

```
                            if (x < pt.X && x >= xJ)
                            {
                                xJ = x;
                                ej = exploredEdges[i];
                                ejIndex = i;
                            }
                        }
                    }

                    if (ejIndex >= 0)
                    {
                        diagonals.Add((helpers[ejIndex], vertex));

                        helpers[ejIndex] = vertex;

                        exploredEdges.Add((prev, vertex));
                        helpers.Add(vertex);
                    }


                    /*(int, int) ej = (-1, -1);
                    (int, int) ek = (-1, -1);

                    double xJ = double.MinValue;
                    double xK = double.MaxValue;

                    for (int i = 0; i < edges.Count; i++)
                    {
                        if ((vertices[edges[i].Item1].Y < pt.Y && vertices[edges[i].Item2].Y > pt.Y) || (vertices[edges[i].Item1].Y > pt.Y && vertices[edges[i].Item2].Y < pt.Y))
                        {
                            double dy = pt.Y - vertices[edges[i].Item1].Y;
                            double dx = dy * (vertices[edges[i].Item2].X - vertices[edges[i].Item1].X) / (vertices[edges[i].Item2].Y - vertices[edges[i].Item1].Y);

                            double x = dx + vertices[edges[i].Item1].X;

                            if (x < pt.X && x >= xJ)
                            {
                                xJ = x;
                                ej = edges[i];
                            }

                            if (x > pt.X && x <= xK)
                            {
                                xK = x;
                                ek = edges[i];
                            }
                        }
                    }

                    Point helper = new Point(double.NaN, double.MinValue);
                    int helperIndex = -1;

                    for (int i = 0; i < vertices.Count; i++)
                    {
                        Point h = vertices[i];
                        if (h.Y < pt.Y && h.Y > helper.Y)
                        {
                            double dyJ = h.Y - vertices[ej.Item1].Y;
                            double dxJ = dyJ * (vertices[ej.Item2].X - vertices[ej.Item1].X) / (vertices[ej.Item2].Y - vertices[ej.Item1].Y);
                            double hxJ = dxJ + vertices[ej.Item1].X;

                            double dyK = h.Y - vertices[ek.Item1].Y;
                            double dxK = dyJ * (vertices[ek.Item2].X - vertices[ek.Item1].X) / (vertices[ek.Item2].Y - vertices[ek.Item1].Y);
                            double hxK = dxJ + vertices[ek.Item1].X;

                            if (h.X >= hxJ && h.X <= hxK)
                            {
                                helper = h;
                                helperIndex = i;
                            }
                        }
                    }

                    diagonals.Add((vertex, helperIndex));*/
                }
                else if (vertexType == VertexType.Merge)
                {
                    int eiM1 = -1;

                    for (int i = exploredEdges.Count - 1; i >= 0; i--)
                    {
                        if (exploredEdges[i].Item1 == vertex && exploredEdges[i].Item2 == next)
                        {
```

```
                            eiM1 = i;
                            break;
                        }
                    }

                    if (eiM1 >= 0)
                    {
                        if (vertexTypes[helpers[eiM1]] == VertexType.Merge)
                        {
                            diagonals.Add((helpers[eiM1], vertex));
                        }

                        exploredEdges.RemoveAt(eiM1);
                        helpers.RemoveAt(eiM1);
                    }

                    (int, int) ej = (-1, -1);
                    int ejIndex = -1;

                    double xJ = double.MinValue;

                    for (int i = 0; i < exploredEdges.Count; i++)
                    {
                        if ((vertices[exploredEdges[i].Item1].Y <= pt.Y && vertices[exploredEdges[i].Item2].Y >= pt.Y) || (vertices[exploredEdges[i].Item1].Y >= pt.Y && vertices[exploredEdges[i].Item2].Y <= pt.Y))
                        {
                            double dy = pt.Y - vertices[exploredEdges[i].Item1].Y;
                            double dx = dy * (vertices[exploredEdges[i].Item2].X - vertices[exploredEdges[i].Item1].X) / (vertices[exploredEdges[i].Item2].Y - vertices[exploredEdges[i].Item1].Y);

                            double x = dx + vertices[exploredEdges[i].Item1].X;

                            if (x < pt.X && x >= xJ)
                            {
                                xJ = x;
                                ej = exploredEdges[i];
                                ejIndex = i;
                            }
                        }
                    }

                    if (ejIndex >= 0)
                    {
                        if (vertexTypes[helpers[ejIndex]] == VertexType.Merge)
                        {
                            diagonals.Add((helpers[ejIndex], vertex));
                        }

                        helpers[ejIndex] = vertex;
                    }

                    //gpr.FillPath(new GraphicsPath().MoveTo(pt.X - 1, pt.Y - 1).LineTo(pt.X, pt.Y + 1).LineTo(pt.X + 1, pt.Y - 1).Close(), Colours.Green);

                    /*(int, int) ej = (-1, -1);
                    (int, int) ek = (-1, -1);

                    double xJ = double.MinValue;
                    double xK = double.MaxValue;

                    for (int i = 0; i < edges.Count; i++)
                    {
                        if ((vertices[edges[i].Item1].Y < pt.Y && vertices[edges[i].Item2].Y > pt.Y) || (vertices[edges[i].Item1].Y > pt.Y && vertices[edges[i].Item2].Y < pt.Y))
                        {
                            double dy = pt.Y - vertices[edges[i].Item1].Y;
                            double dx = dy * (vertices[edges[i].Item2].X - vertices[edges[i].Item1].X) / (vertices[edges[i].Item2].Y - vertices[edges[i].Item1].Y);

                            double x = dx + vertices[edges[i].Item1].X;

                            if (x < pt.X && x >= xJ)
                            {
                                xJ = x;
                                ej = edges[i];
                            }

                            if (x > pt.X && x <= xK)
                            {
                                xK = x;
                                ek = edges[i];
                            }
                        }
                    }

```

```
                    Point helper = new Point(double.NaN, double.MaxValue);
                    int helperIndex = -1;

                    for (int i = 0; i < vertices.Count; i++)
                    {
                        Point h = vertices[i];
                        if (h.Y > pt.Y && h.Y < helper.Y)
                        {
                            double dyJ = h.Y - vertices[ej.Item1].Y;
                            double dxJ = dyJ * (vertices[ej.Item2].X - vertices[ej.Item1].X) / (vertices[ej.Item2].Y - vertices[ej.Item1].Y);
                            double hxJ = dxJ + vertices[ej.Item1].X;

                            double dyK = h.Y - vertices[ek.Item1].Y;
                            double dxK = dyJ * (vertices[ek.Item2].X - vertices[ek.Item1].X) / (vertices[ek.Item2].Y - vertices[ek.Item1].Y);
                            double hxK = dxJ + vertices[ek.Item1].X;

                            if (h.X >= hxJ && h.X <= hxK)
                            {
                                helper = h;
                                helperIndex = i;
                            }
                        }
                    }

                    diagonals.Add((vertex, helperIndex));*/
                }
                else if (vertexType == VertexType.Regular)
                {
                    if ((isAntiClockwise && (prevPoint.Y < pt.Y || pt.Y < nextPoint.Y)) || (!isAntiClockwise && (prevPoint.Y > pt.Y || pt.Y > nextPoint.Y)))
                    {
                        int eiM1 = -1;

                        for (int i = exploredEdges.Count - 1; i >= 0; i--)
                        {
                            if (exploredEdges[i].Item1 == vertex && exploredEdges[i].Item2 == next)
                            {
                                eiM1 = i;
                                break;
                            }
                        }

                        if (eiM1 >= 0)
                        {
                            if (vertexTypes[helpers[eiM1]] == VertexType.Merge)
                            {
                                diagonals.Add((helpers[eiM1], vertex));
                            }

                            exploredEdges.RemoveAt(eiM1);
                            helpers.RemoveAt(eiM1);
                        }

                        exploredEdges.Add((prev, vertex));
                        helpers.Add(vertex);
                    }
                    else
                    {
                        (int, int) ej = (-1, -1);
                        int ejIndex = -1;

                        double xJ = double.MinValue;

                        for (int i = 0; i < exploredEdges.Count; i++)
                        {
                            if ((vertices[exploredEdges[i].Item1].Y <= pt.Y && vertices[exploredEdges[i].Item2].Y >= pt.Y) || (vertices[exploredEdges[i].Item1].Y >= pt.Y && vertices[exploredEdges[i].Item2].Y <= pt.Y))
                            {
                                double dy = pt.Y - vertices[exploredEdges[i].Item1].Y;
                                double dx = dy * (vertices[exploredEdges[i].Item2].X - vertices[exploredEdges[i].Item1].X) / (vertices[exploredEdges[i].Item2].Y - vertices[exploredEdges[i].Item1].Y);

                                double x = dx + vertices[exploredEdges[i].Item1].X;

                                if (x < pt.X && x >= xJ)
                                {
                                    xJ = x;
                                    ej = exploredEdges[i];
                                    ejIndex = i;
                                }
                            }
                        }

```

```
                        if (ejIndex >= 0)
                        {
                            if (vertexTypes[helpers[ejIndex]] == VertexType.Merge)
                            {
                                diagonals.Add((helpers[ejIndex], vertex));
                            }

                            helpers[ejIndex] = vertex;
                        }
                    }
                }
            }

            for (int i = diagonals.Count - 1; i >= 0; i--)
            {
                for (int j = 0; j < edges.Count; j++)
                {
                    if (CompareEdges(diagonals[i], edges[j]))
                    {
                        diagonals.RemoveAt(i);
                        break;
                    }
                }
            }

            List<List<(int, int)» polygons = SplitPolygons(edges, diagonals, vertices, isAntiClockwise ? prevs : nexts);

            int[] directions = new int[vertices.Count];

            int ind = 0;

            foreach (List<(int, int)> polygon in polygons)
            {
                foreach (GraphicsPath pth in TriangulateMonotone(vertices, polygon, directions, clockwise ? -1 : 1))
                {
                    yield return pth;
                }

                ind++;
            }
        }

        private static bool CompareEdges((int, int) edge1, (int, int) edge2)
        {
            return (edge1.Item1 == edge2.Item1 && edge1.Item2 == edge2.Item2) || (edge1.Item1 == edge2.Item2 && edge1.Item2 == edge2.Item1);
        }

        private static List<List<(int, int)» SplitPolygons(List<(int, int)> edges, List<(int, int)> diagonals, List<Point> vertices, int[] nexts)
        {
            List<List<(int, int)» polygons = new List<List<(int, int)»();

            List<int>[] outPaths = new List<int>[vertices.Count];

            for (int i = 0; i < edges.Count; i++)
            {
                if (outPaths[edges[i].Item1] == null)
                {
                    outPaths[edges[i].Item1] = new List<int>();
                }

                if (outPaths[edges[i].Item2] == null)
                {
                    outPaths[edges[i].Item2] = new List<int>();
                }

                outPaths[edges[i].Item1].Add(edges[i].Item2);
                outPaths[edges[i].Item2].Add(edges[i].Item1);
            }

            for (int i = 0; i < diagonals.Count; i++)
            {
                if (outPaths[diagonals[i].Item1] == null)
                {
                    outPaths[diagonals[i].Item1] = new List<int>();
                }

                if (outPaths[diagonals[i].Item2] == null)
                {
                    outPaths[diagonals[i].Item2] = new List<int>();
                }

                outPaths[diagonals[i].Item1].Add(diagonals[i].Item2);
                outPaths[diagonals[i].Item1].Add(diagonals[i].Item2);
```

```
                outPaths[diagonals[i].Item2].Add(diagonals[i].Item1);
                outPaths[diagonals[i].Item2].Add(diagonals[i].Item1);
            }

            List<int> activeVertices = (from el in Enumerable.Range(0, outPaths.Length) where outPaths[el] != null && outPaths[el].Count > 0 select el).ToList();

            int[] newNexts = new int[nexts.Length];

            for (int i = 0; i < nexts.Length; i++)
            {
                if (activeVertices.Contains(i))
                {
                    int currNext = nexts[i];
                    while (!outPaths[i].Contains(currNext))
                    {
                        currNext = nexts[currNext];
                    }
                    newNexts[i] = currNext;
                }
                else
                {
                    newNexts[i] = -1;
                }
            }

            while (activeVertices.Count > 0)
            {
                List<(int, int)> polygon = new List<(int, int)>();

                int startPoint = activeVertices.Min();

                if (!outPaths[startPoint].Contains(newNexts[startPoint]))
                {
                    throw new InvalidOperationException("Missing edge!");
                }

                int prevVertex = startPoint;
                int currVertex = newNexts[startPoint];

                outPaths[startPoint].Remove(currVertex);
                outPaths[currVertex].Remove(startPoint);
                polygon.Add((startPoint, currVertex));

                while (currVertex != startPoint)
                {
                    Point currPoint = vertices[currVertex];
                    Point prevPoint = vertices[prevVertex];

                    double angleIncoming = Math.Atan2(prevPoint.Y - currPoint.Y, prevPoint.X - currPoint.X);
                    if (angleIncoming < 0)
                    {
                        angleIncoming += 2 * Math.PI;
                    }

                    double maxAngle = double.MinValue;
                    int candidateVertex = -1;

                    for (int i = 0; i < outPaths[currVertex].Count; i++)
                    {
                        double angleI = Math.Atan2(vertices[outPaths[currVertex][i]].Y - currPoint.Y, vertices[outPaths[currVertex][i]].X - currPoint.X);
                        if (angleI < 0)
                        {
                            angleI += 2 * Math.PI;
                        }
                        angleI -= angleIncoming;
                        if (angleI < 0)
                        {
                            angleI += 2 * Math.PI;
                        }

                        if (angleI > maxAngle)
                        {
                            candidateVertex = outPaths[currVertex][i];
                            maxAngle = angleI;
                        }
                    }

                    outPaths[currVertex].Remove(candidateVertex);
                    outPaths[candidateVertex].Remove(currVertex);
                    polygon.Add((currVertex, candidateVertex));

                    prevVertex = currVertex;
                    currVertex = candidateVertex;
                }
```

```

                polygons.Add(polygon);
                activeVertices = (from el in Enumerable.Range(0, outPaths.Length) where outPaths[el] != null && outPaths[el].Count > 0 select el).ToList();

                if (activeVertices.Contains(currVertex))
                {
                    int currNext = newNexts[currVertex];
                    while (!outPaths[currVertex].Contains(currNext))
                    {
                        currNext = newNexts[currNext];
                    }
                    newNexts[currVertex] = currNext;
                }

                if (activeVertices.Contains(prevVertex))
                {
                    int currNext = newNexts[prevVertex];
                    while (!outPaths[prevVertex].Contains(currNext))
                    {
                        currNext = newNexts[currNext];
                    }
                    newNexts[prevVertex] = currNext;
                }
            }

            return polygons;
        }

        private static IEnumerable<GraphicsPath> TriangulateMonotone(List<Point> vertices, List<(int, int)> edges, int[] directions, int targetSign)
        {
            int getDirection((int, int) edge)
            {
                if (vertices[edge.Item2].Y != vertices[edge.Item1].Y)
                {
                    return Math.Sign(vertices[edge.Item2].Y - vertices[edge.Item1].Y);
                }
                else
                {
                    for (int i = 0; i < edges.Count; i++)
                    {
                        if (edges[i].Item2 == edge.Item1)
                        {
                            return getDirection(edges[i]);
                        }
                    }
                }

                throw new InvalidOperationException("Unknown edge direction!");
            }


            for (int i = 0; i < edges.Count; i++)
            {
                directions[edges[i].Item1] = getDirection(edges[i]);
            }

            int[] sortedVertices = (from el in edges select el.Item1).OrderBy(a => a, Comparer<int>.Create((a, b) =>
            {
                if (vertices[a].Y != vertices[b].Y)
                {
                    return Math.Sign(vertices[a].Y - vertices[b].Y);
                }
                else
                {
                    return Math.Sign(vertices[a].X - vertices[b].X);
                }
            })).ToArray();

            List<(int, int)> diagonals = new List<(int, int)>();

            Stack<int> stack = new Stack<int>();

            stack.Push(sortedVertices[0]);
            stack.Push(sortedVertices[1]);

            for (int i = 2; i < sortedVertices.Length - 1; i++)
            {
                int onTop = stack.Peek();

                if (directions[sortedVertices[i]] != directions[onTop])
                {
                    while (stack.Count > 1)
                    {
                        int v = stack.Pop();
```

```
                        diagonals.Add((v, sortedVertices[i]));
                    }

                    stack.Pop();

                    stack.Push(sortedVertices[i - 1]);
                    stack.Push(sortedVertices[i]);
                }
                else
                {
                    int lastPopped = stack.Pop();

                    bool shouldContinue = true;

                    while (shouldContinue && stack.Count > 0)
                    {
                        int currVert = stack.Peek();

                        double areaSign = Math.Sign((vertices[currVert].X - vertices[lastPopped].X) * (vertices[sortedVertices[i]].Y - vertices[lastPopped].Y) - (vertices[currVert].Y - vertices[lastPopped].Y) * (vertices[sortedVertices[i]].X - vertices[lastPopped].X));

                        double dirSign = Math.Sign(vertices[currVert].X - vertices[lastPopped].X);

                        if (areaSign * dirSign > 0)
                        {
                            lastPopped = stack.Pop();

                            diagonals.Add((currVert, sortedVertices[i]));
                        }
                        else
                        {
                            shouldContinue = false;
                        }
                    }

                    stack.Push(lastPopped);
                    stack.Push(sortedVertices[i]);
                }
            }

            stack.Pop();

            while (stack.Count > 1)
            {
                int v = stack.Pop();

                diagonals.Add((v, sortedVertices[sortedVertices.Length - 1]));
            }

            List<int>[] connections = new List<int>[vertices.Count];

            for (int i = 0; i < edges.Count; i++)
            {
                connections[edges[i].Item1] = new List<int>();
            }

            for (int i = 0; i < edges.Count; i++)
            {
                connections[edges[i].Item1].Add(edges[i].Item2);
                connections[edges[i].Item2].Add(edges[i].Item1);
            }

            for (int i = 0; i < diagonals.Count; i++)
            {
                connections[diagonals[i].Item1].Add(diagonals[i].Item2);
                connections[diagonals[i].Item1].Add(diagonals[i].Item2);

                connections[diagonals[i].Item2].Add(diagonals[i].Item1);
                connections[diagonals[i].Item2].Add(diagonals[i].Item1);
            }

            int totalTriangles = (edges.Count + diagonals.Count * 2) / 3;

            List<List<(int, int)» polygons = new List<List<(int, int)»();

            while (polygons.Count < totalTriangles)
            {
                int p1 = -1;
                int p2 = -1;

                for (int i = 0; i < connections.Length; i++)
                {
                    if (connections[i] != null && connections[i].Count > 0)
                    {
                        p1 = i;
```

```
                        p2 = connections[i][0];

                        connections[i].Remove(p2);
                        connections[p2].Remove(i);
                        break;
                    }
                }

                int p3 = -1;

                for (int i = 0; i < connections[p1].Count; i++)
                {
                    if (connections[connections[p1][i]].Contains(p2))
                    {
                        p3 = connections[p1][i];
                        connections[p1].Remove(p3);
                        connections[p2].Remove(p3);
                        connections[p3].Remove(p1);
                        connections[p3].Remove(p2);
                        break;
                    }
                }

                int sign = Math.Sign((vertices[p1].X - vertices[p2].X) * (vertices[p3].Y - vertices[p2].Y) - (vertices[p1].Y - vertices[p2].Y) * (vertices[p3].X - vertices[p2].X));

                if (sign == targetSign)
                {
                    polygons.Add(new List<(int, int)>() { (p1, p2), (p2, p3), (p3, p1) });
                }
                else
                {
                    polygons.Add(new List<(int, int)>() { (p1, p3), (p3, p2), (p2, p1) });
                }
            }

            foreach (List<(int, int)> polygon in polygons)
            {
                GraphicsPath polygonPath = new GraphicsPath();
                foreach ((int, int) edge in polygon)
                {
                    if (polygonPath.Segments.Count == 0)
                    {
                        polygonPath.MoveTo(vertices[edge.Item1]);
                    }

                    polygonPath.LineTo(vertices[edge.Item2]);
                }

                yield return polygonPath;
            }
        }

        /// <summary>
        /// Transforms all of the <see cref="Point"/>s in the <see cref="GraphicsPath"/> with an arbitrary transformation function.
        /// </summary>
        /// <param name="transformationFunction">An arbitrary transformation function.</param>
        /// <returns>A new <see cref="GraphicsPath"/> in which all points have been replaced using the <paramref name="transformationFunction"/>.</returns>
        public GraphicsPath Transform(Func<Point, Point> transformationFunction)
        {
            GraphicsPath tbr = new GraphicsPath();

            foreach (Segment seg in this.Segments)
            {
                tbr.Segments.AddRange(seg.Transform(transformationFunction));
            }

            return tbr;
        }


        private Rectangle cachedBounds = Rectangle.NaN;

        /// <summary>
        /// Compute the rectangular bounds of the path.
        /// </summary>
        /// <returns>The smallest <see cref="Rectangle"/> that contains the path.</returns>
        public Rectangle GetBounds()
        {
            if (double.IsNaN(cachedBounds.Location.X) || double.IsNaN(cachedBounds.Location.Y) || double.IsNaN(cachedBounds.Size.Width) || double.IsNaN(cachedBounds.Size.Height))
            {
                Point min = new Point(double.MaxValue, double.MaxValue);
                Point max = new Point(double.MinValue, double.MinValue);

```

```
                Point currPoint = new Point();
                Point figureStartPoint = new Point();

                for (int i = 0; i < this.Segments.Count; i++)
                {
                    switch (this.Segments[i].Type)
                    {
                        case SegmentType.Move:
                            currPoint = this.Segments[i].Point;
                            figureStartPoint = this.Segments[i].Point;
                            min = Point.Min(min, this.Segments[i].Point);
                            max = Point.Max(max, this.Segments[i].Point);
                            break;
                        case SegmentType.Line:
                            if (i > 0)
                            {
                                currPoint = this.Segments[i].Point;
                            }
                            else
                            {
                                currPoint = this.Segments[i].Point;
                                figureStartPoint = this.Segments[i].Point;
                            }
                            min = Point.Min(min, this.Segments[i].Point);
                            max = Point.Max(max, this.Segments[i].Point);
                            break;
                        case SegmentType.Arc:
                            ArcSegment seg = (ArcSegment)this.Segments[i];

                            if (i == 0)
                            {
                                figureStartPoint = new Point(seg.Points[0].X + Math.Cos(seg.StartAngle) * seg.Radius, seg.Points[0].Y + Math.Sin(seg.StartAngle) * seg.Radius);
                                currPoint = figureStartPoint;

                                min = Point.Min(min, currPoint);
                                max = Point.Max(max, currPoint);
                            }

                            double theta1 = seg.StartAngle;
                            double theta2 = seg.EndAngle;

                            if (theta1 > theta2)
                            {
                                double temp = theta1;
                                theta1 = theta2;
                                theta2 = temp;
                            }

                            while (theta1 < 0)
                            {
                                theta1 += 2 * Math.PI;
                                theta2 += 2 * Math.PI;
                            }

                            theta1 /= 2 * Math.PI;
                            theta2 /= 2 * Math.PI;

                            int minAngle = (int)Math.Min(0, Math.Floor(theta1)) * 4;
                            int maxAngle = (int)Math.Max(1, Math.Ceiling(theta2)) * 4;

                            theta1 *= 4;
                            theta2 *= 4;

                            bool right = false;
                            bool bottom = false;
                            bool left = false;
                            bool top = false;

                            for (int j = minAngle; j < maxAngle; j++)
                            {
                                if (theta1 <= j && theta2 >= j)
                                {
                                    switch (j % 4)
                                    {
                                        case 0:
                                            right = true;
                                            break;
                                        case 1:
                                            bottom = true;
                                            break;
                                        case 2:
                                            left = true;
                                            break;
                                        case 3:
                                            top = true;
                                            break;
```

```
                                    }
                                }
                            }

                            if (right)
                            {
                                Point p = new Point(seg.Points[0].X + seg.Radius, seg.Points[0].Y);
                                min = Point.Min(min, p);
                                max = Point.Max(max, p);
                            }

                            if (bottom)
                            {
                                Point p = new Point(seg.Points[0].X, seg.Points[0].Y + seg.Radius);
                                min = Point.Min(min, p);
                                max = Point.Max(max, p);
                            }

                            if (left)
                            {
                                Point p = new Point(seg.Points[0].X - seg.Radius, seg.Points[0].Y);
                                min = Point.Min(min, p);
                                max = Point.Max(max, p);
                            }

                            if (top)
                            {
                                Point p = new Point(seg.Points[0].X, seg.Points[0].Y - seg.Radius);
                                min = Point.Min(min, p);
                                max = Point.Max(max, p);
                            }

                            min = Point.Min(min, this.Segments[i].Point);
                            max = Point.Max(max, this.Segments[i].Point);

                            currPoint = this.Segments[i].Point;
                            break;
                        case SegmentType.Close:
                            currPoint = figureStartPoint;
                            break;
                        case SegmentType.CubicBezier:
                            if (i == 0)
                            {
                                currPoint = this.Segments[i].Points[0];
                                figureStartPoint = this.Segments[i].Points[0];
                            }

                            min = Point.Min(min, this.Segments[i].Point);
                            max = Point.Max(max, this.Segments[i].Point);

                            {
                                double aX = -currPoint.X + 3 * this.Segments[i].Points[0].X - 3 *
        this.Segments[i].Points[1].X + this.Segments[i].Point.X;
                                double bX = 2 * (currPoint.X - 2 * this.Segments[i].Points[0].X +
        this.Segments[i].Points[1].X);
                                double cX = this.Segments[i].Points[0].X - currPoint.X;

                                double t1X;
                                double t2X;

                                if (Math.Abs(aX) < 1e-5)
                                {
                                    if (Math.Abs(bX) >= 1e-5)
                                    {
                                        t1X = -cX / bX;
                                        t2X = t1X;
                                    }
                                    else
                                    {
                                        t1X = double.NaN;
                                        t2X = double.NaN;
                                    }
                                }
                                else
                                {
                                    double delta = bX * bX - 4 * aX * cX;

                                    if (delta >= -1e-5)
                                    {
                                        delta = Math.Max(delta, 0);

                                        delta = Math.Sqrt(delta);
                                        t1X = (-bX + delta) / (2 * aX);
                                        t2X = (-bX - delta) / (2 * aX);
                                    }
                                    else
                                    {
```

```
                                        t1X = double.NaN;
                                        t2X = double.NaN;
                                    }
                                }

                                if (t1X >= 0 && t1X <= 1)
                                {
                                    Point p1 = ((CubicBezierSegment)this.Segments[i]).GetBezierPointAt(currPoint, t1X);
                                    min = Point.Min(min, p1);
                                    max = Point.Max(max, p1);
                                }

                                if (t2X >= 0 && t2X <= 1)
                                {
                                    Point p2 = ((CubicBezierSegment)this.Segments[i]).GetBezierPointAt(currPoint, t2X);
                                    min = Point.Min(min, p2);
                                    max = Point.Max(max, p2);
                                }
                            }

                            {
                                double aY = -currPoint.Y + 3 * this.Segments[i].Points[0].Y - 3 * this.Segments[i].Points[1].Y + this.Segments[i].Point.Y;
                                double bY = 2 * (currPoint.Y - 2 * this.Segments[i].Points[0].Y + this.Segments[i].Points[1].Y);
                                double cY = this.Segments[i].Points[0].Y - currPoint.Y;

                                double t1Y;
                                double t2Y;

                                if (Math.Abs(aY) < 1e-5)
                                {
                                    if (Math.Abs(bY) >= 1e-5)
                                    {
                                        t1Y = -cY / bY;
                                        t2Y = t1Y;
                                    }
                                    else
                                    {
                                        t1Y = double.NaN;
                                        t2Y = double.NaN;
                                    }
                                }
                                else
                                {
                                    double delta = bY * bY - 4 * aY * cY;

                                    if (delta >= -1e-5)
                                    {
                                        delta = Math.Max(delta, 0);

                                        delta = Math.Sqrt(delta);
                                        t1Y = (-bY + delta) / (2 * aY);
                                        t2Y = (-bY - delta) / (2 * aY);
                                    }
                                    else
                                    {
                                        t1Y = double.NaN;
                                        t2Y = double.NaN;
                                    }
                                }

                                if (t1Y >= 0 && t1Y <= 1)
                                {
                                    Point p1 = ((CubicBezierSegment)this.Segments[i]).GetBezierPointAt(currPoint, t1Y);
                                    min = Point.Min(min, p1);
                                    max = Point.Max(max, p1);
                                }

                                if (t2Y >= 0 && t2Y <= 1)
                                {
                                    Point p2 = ((CubicBezierSegment)this.Segments[i]).GetBezierPointAt(currPoint, t2Y);
                                    min = Point.Min(min, p2);
                                    max = Point.Max(max, p2);
                                }
                            }

                            currPoint = this.Segments[i].Point;
                            break;
                        }
                    }

                    cachedBounds = new Rectangle(min, max);
```

```
            }

            return cachedBounds;
        }

        internal GraphicsPath ConvertArcsToBeziers()
        {
            GraphicsPath tbr = new GraphicsPath();

            foreach (Segment seg in this.Segments)
            {
                if (seg is ArcSegment arc)
                {
                    tbr.Segments.AddRange(arc.ToBezierSegments());
                }
                else
                {
                    tbr.Segments.Add(seg);
                }
            }

            return tbr;
        }
    }
}
```

8.48 ImageFormats.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.IO;
using System.IO.Compression;
using System.Runtime.InteropServices;
using System.Threading;

namespace VectSharp
{
    /// <summary>
    /// Contains methods to create animated PNG image files.
    /// </summary>
    public static class AnimatedPNG
    {
        /// <summary>
        /// Types of inter-frame compression.
        /// </summary>
        public enum InterframeCompression
        {
            /// <summary>
            /// No inter-frame compression is performed (fastest, but produces the largest files).
            /// </summary>
            None,

            /// <summary>
            /// Inter-frame compression is performed by encoding the difference between each frame and the first frame in the animation.
            /// </summary>
            First,

            /// <summary>
            /// Inter-frame compression is performed by encoding the difference between each frame and the previous frame (produces the smallest files, but it is the slowest and requires large amounts of memory).
            /// </summary>
            Previous
        }

```

```
        /// <summary>
        /// Represents an individual frame of a PNG animation.
        /// </summary>
        public class CompressedFrame : IDisposable
        {
            internal MemoryStream compressedStream;
            private bool disposedValue;

            /// <summary>
            /// The duration of the frame in milliseconds.
            /// </summary>
            public double Duration { get; }

            internal InterframeCompression InterframeCompression { get; } = InterframeCompression.None;

            /// <summary>
            /// Creates a new <see cref="CompressedFrame"/> from raw pixel data in RGB or RGBA format.
            /// </summary>
            /// <param name="rawFrameData">A pointer to the raw pixel data for the frame.</param>
            /// <param name="width">The width of the image in pixels.</param>
            /// <param name="height">The height of the image in pixels.</param>
            /// <param name="hasAlpha">If the image data is in RGBA format, set this to <see langword="true"/>; if it is in RGB format, set it to <see langword="false"/>.</param>
            /// <param name="duration">The duration of the frame in milliseconds.</param>
            public unsafe CompressedFrame(DisposableIntPtr rawFrameData, int width, int height, bool hasAlpha, double duration)
            {
                this.compressedStream = PNGEncoder.CompressAPNGFrame((byte*)rawFrameData.InternalPointer, width, height, hasAlpha, PNGEncoder.FilterModes.Adaptive, 1);
                this.Duration = duration;
                this.InterframeCompression = InterframeCompression.None;
            }

            /// <summary>
            /// Creates a new <see cref="CompressedFrame"/> from raw pixel data in RGB or RGBA format, applying inter-frame compression based on another frame.
            /// </summary>
            /// <param name="rawFrameData">A pointer to the raw pixel data for the new frame.</param>
            /// <param name="otherFrameData">A pointer to the raw pixel data for the frame to use as a reference. This can either be the first frame in the animation, or the frame immediately preceeding the current frame.</param>
            /// <param name="isFirstFrame">If <paramref name="otherFrameData"/> is the raw pixel data for the first frame in the animation, set this to <see langword="true"/>; if it is the immediately preceeding frame, set this to <see langword="false"/>.</param>
            /// <param name="width">The width of the image in pixels.</param>
            /// <param name="height">The height of the image in pixels.</param>
            /// <param name="hasAlpha">If the image data is in RGBA format, set this to <see langword="true"/>; if it is in RGB format, set it to <see langword="false"/>.</param>
            /// <param name="duration">The duration of the frame in milliseconds.</param>
            public unsafe CompressedFrame(DisposableIntPtr rawFrameData, DisposableIntPtr otherFrameData, bool isFirstFrame, int width, int height, bool hasAlpha, double duration)
            {
                int pixelSize = hasAlpha ? 4 : 3;

                IntPtr frameDiff = Marshal.AllocHGlobal(width * height * 4);

                byte* frameDiffData = (byte*)frameDiff;
                byte* previousFrame = (byte*)otherFrameData.InternalPointer;
                byte* currentFrame = (byte*)rawFrameData.InternalPointer;

                for (int y = 0; y < height; y++)
                {
                    for (int x = 0; x < width; x++)
                    {
                        int pixelIndex = (y * width + x) * pixelSize;

                        if (previousFrame[pixelIndex] != currentFrame[pixelIndex] || previousFrame[pixelIndex + 1] != currentFrame[pixelIndex + 1] ||
                            previousFrame[pixelIndex + 2] != currentFrame[pixelIndex + 2] || (hasAlpha && previousFrame[pixelIndex + 3] != currentFrame[pixelIndex + 3]))
                        {
                            frameDiffData[pixelIndex] = currentFrame[pixelIndex];
                            frameDiffData[pixelIndex + 1] = currentFrame[pixelIndex + 1];
                            frameDiffData[pixelIndex + 2] = currentFrame[pixelIndex + 2];

                            if (hasAlpha)
                            {
                                frameDiffData[pixelIndex + 3] = currentFrame[pixelIndex + 3];
                            }
                            else
                            {
                                frameDiffData[pixelIndex + 3] = 255;
                            }
                        }
                        else
```

```
                        {
                            frameDiffData[pixelIndex] = 0;
                            frameDiffData[pixelIndex + 1] = 0;
                            frameDiffData[pixelIndex + 2] = 0;
                            frameDiffData[pixelIndex + 3] = 0;
                        }
                    }
                }

                this.compressedStream = PNGEncoder.CompressAPNGFrame((byte*)frameDiff, width, height, true, PNGEncoder.FilterModes.Adaptive, 1);
                this.Duration = duration;

                if (isFirstFrame)
                {
                    this.InterframeCompression = InterframeCompression.First;
                }
                else
                {
                    this.InterframeCompression = InterframeCompression.Previous;
                }

                Marshal.FreeHGlobal(frameDiff);
            }

            /// <inheritdoc/>
            protected virtual void Dispose(bool disposing)
            {
                if (!disposedValue)
                {
                    if (disposing)
                    {
                        this.compressedStream.Dispose();
                    }

                    disposedValue = true;
                }
            }

            /// <inheritdoc/>
            public void Dispose()
            {
                Dispose(disposing: true);
                GC.SuppressFinalize(this);
            }
        }

        /// <summary>
        /// Create a new animated PNG image, outputting it to the specified stream.
        /// </summary>
        /// <param name="outputStream">The stream to which the animated PNG image will be written.</param>
        /// <param name="width">The width of the image in pixels.</param>
        /// <param name="height">The height of the image in pixels.</param>
        /// <param name="hasAlpha">If the frames of the image have an alpha channel, set this to <see langword="true"/>; otherwise, set it to <see langword="false"/>.</param>
        /// <param name="compressedFrames">The frames that will be used to create the animated PNG image.</param>
        /// <param name="repeatCount">The number of times that the animation should loop. Set this to 0 for an infinitely repeating animation.</param>
        public static unsafe void Create(Stream outputStream, int width, int height, bool hasAlpha, IReadOnlyList<CompressedFrame> compressedFrames, int repeatCount)
        {
            PNGEncoder.StartAPNG(width, height, hasAlpha, outputStream, compressedFrames.Count, repeatCount);

            for (int i = 0; i < compressedFrames.Count; i++)
            {
                ushort frameDurationNumerator = (ushort)Math.Round(compressedFrames[i].Duration * 60);
                ushort frameDurationDenominator = 60000;
                PNGEncoder.AddPreCompressedAPNGFrame(compressedFrames[i], width, height, hasAlpha, outputStream, i, frameDurationNumerator, frameDurationDenominator);
            }

            PNGEncoder.FinishAPNG(outputStream);
        }
    }



    internal static class PNGEncoder
    {
        internal unsafe static void SavePNG(byte* image, int width, int height, bool hasAlpha, Stream fs, FilterModes filter, int threadCount = 0)
        {
            if (threadCount == 0)
            {
```

```
                threadCount = filter == FilterModes.Adaptive ? Math.Max(1, Math.Min(width / 600, Environment.ProcessorCount - 2)) : 1;
            }

            //Header
            fs.Write(new byte[] { 0x89, 0x50, 0x4E, 0x47, 0x0D, 0x0A, 0x1A, 0x0A }, 0, 8);

            //IHDR chunk
            fs.WriteInt(13);
            using (MemoryStream ihdr = new MemoryStream(13))
            {
                ihdr.WriteASCIIString("IHDR");
                ihdr.WriteInt(width);
                ihdr.WriteInt(height);
                ihdr.WriteByte(8); //Bit depth

                if (hasAlpha)
                {
                    ihdr.WriteByte(6); //Colour type
                }
                else
                {
                    ihdr.WriteByte(2); //Colour type
                }

                ihdr.WriteByte(0); //Compression method
                ihdr.WriteByte(0); //Filter method
                ihdr.WriteByte(0); //Interlace

                ihdr.Seek(0, SeekOrigin.Begin);
                ihdr.CopyTo(fs);

                fs.WriteUInt(CRC32.ComputeCRC(ihdr));
            }

            //IDAT chunk
            IntPtr filteredImage;

            if (threadCount > 1)
            {
                filteredImage = FilterImageData(image, width, height, hasAlpha ? 4 : 3, filter, threadCount);
            }
            else
            {
                filteredImage = FilterImageData(image, width, height, hasAlpha ? 4 : 3, filter);
            }

            using (MemoryStream compressedImage = StreamUtils.ZLibCompress(filteredImage, height * (width * (hasAlpha ? 4 : 3) + 1)))
            {
                compressedImage.Seek(0, SeekOrigin.Begin);
                fs.WriteUInt((uint)compressedImage.Length);
                fs.WriteASCIIString("IDAT");
                compressedImage.Seek(0, SeekOrigin.Begin);
                compressedImage.CopyTo(fs);

                fs.WriteUInt(CRC32.ComputeCRC(compressedImage.GetBuffer(), (int)compressedImage.Length, new byte[] { 73, 68, 65, 84 }));
            }

            Marshal.FreeHGlobal(filteredImage);

            //IEND chunk
            fs.WriteInt(0);
            fs.WriteASCIIString("IEND");
            fs.Write(new byte[] { 0xAE, 0x42, 0x60, 0x82 }, 0, 4);

        }

        public unsafe static void StartAPNG(int width, int height, bool hasAlpha, Stream fs, int frameCount, int repeatCount)
        {
            //Header
            fs.Write(new byte[] { 0x89, 0x50, 0x4E, 0x47, 0x0D, 0x0A, 0x1A, 0x0A }, 0, 8);

            //IHDR chunk
            fs.WriteInt(13);
            using (MemoryStream ihdr = new MemoryStream(17))
            {
                ihdr.WriteASCIIString("IHDR");
                ihdr.WriteInt(width);
                ihdr.WriteInt(height);
                ihdr.WriteByte(8); //Bit depth

                if (hasAlpha)
                {
```

```
                    ihdr.WriteByte(6); //Colour type
                }
                else
                {
                    ihdr.WriteByte(2); //Colour type
                }

                ihdr.WriteByte(0); //Compression method
                ihdr.WriteByte(0); //Filter method
                ihdr.WriteByte(0); //Interlace

                ihdr.Seek(0, SeekOrigin.Begin);
                ihdr.CopyTo(fs);

                fs.WriteUInt(CRC32.ComputeCRC(ihdr));
            }

            //acTL chunk
            fs.WriteInt(8);
            using (MemoryStream actl = new MemoryStream(12))
            {
                actl.WriteASCIIString("acTL");
                actl.WriteInt(frameCount);
                actl.WriteInt(repeatCount);

                actl.Seek(0, SeekOrigin.Begin);
                actl.CopyTo(fs);

                fs.WriteUInt(CRC32.ComputeCRC(actl));
            }
        }

        internal unsafe static void AddAPNGFrame(byte* image, int width, int height, bool hasAlpha, Stream fs, FilterModes filter, int frameNumber, ushort frameDelayNumerator, ushort frameDelayDenominator, int threadCount = 0)
        {
            if (threadCount == 0)
            {
                threadCount = filter == FilterModes.Adaptive ? Math.Max(1, Math.Min(width / 600, Environment.ProcessorCount - 2)) : 1;
            }

            int fcTLIndex = frameNumber == 0 ? 0 : ((frameNumber - 1) * 2 + 1);
            int fDATIndex = frameNumber == 0 ? 0 : (frameNumber * 2);

            //fcTL chunk
            fs.WriteInt(26);
            using (MemoryStream fctl = new MemoryStream(30))
            {
                fctl.WriteASCIIString("fcTL");
                fctl.WriteInt(fcTLIndex);
                fctl.WriteInt(width);
                fctl.WriteInt(height);
                fctl.WriteInt(0);
                fctl.WriteInt(0);
                fctl.WriteUShort(frameDelayNumerator);
                fctl.WriteUShort(frameDelayDenominator);
                fctl.WriteByte(1);
                fctl.WriteByte(0);

                fctl.Seek(0, SeekOrigin.Begin);
                fctl.CopyTo(fs);

                fs.WriteUInt(CRC32.ComputeCRC(fctl));
            }

            //fDAT chunk
            IntPtr filteredImage;

            if (threadCount > 1)
            {
                filteredImage = FilterImageData(image, width, height, hasAlpha ? 4 : 3, filter, threadCount);
            }
            else
            {
                filteredImage = FilterImageData(image, width, height, hasAlpha ? 4 : 3, filter);
            }

            if (frameNumber > 0)
            {
                using (MemoryStream compressedImage = StreamUtils.ZLibCompress(filteredImage, height * (width * (hasAlpha ? 4 : 3) + 1)))
                {
                    fs.WriteUInt((uint)compressedImage.Length + 4);

                    fs.WriteASCIIString("fdAT");
```

```
                    fs.WriteInt(fDATIndex);

                    compressedImage.Seek(0, SeekOrigin.Begin);
                    compressedImage.CopyTo(fs);

                    fs.WriteUInt(CRC32.ComputeCRC(compressedImage.GetBuffer(), (int)compressedImage.Length, new byte[] { 102, 100, 65, 84, (byte)(fDATIndex » 24), (byte)((fDATIndex » 16) & 255), (byte)((fDATIndex » 8) & 255), (byte)(fDATIndex & 255) }));
                    //fs.WriteUInt(CRC32.ComputeCRC(chunkStream.GetBuffer(), (int)chunkStream.Length, new byte[] { 73, 68, 65, 84 }));
                    //fs.WriteUInt(CRC32.ComputeCRC(chunkStream));
                }
            }
            else
            {
                using (MemoryStream compressedImage = StreamUtils.ZLibCompress(filteredImage, height * (width * (hasAlpha ? 4 : 3) + 1)))
                {
                    compressedImage.Seek(0, SeekOrigin.Begin);
                    fs.WriteUInt((uint)compressedImage.Length);
                    fs.WriteASCIIString("IDAT");
                    compressedImage.Seek(0, SeekOrigin.Begin);
                    compressedImage.CopyTo(fs);

                    fs.WriteUInt(CRC32.ComputeCRC(compressedImage.GetBuffer(), (int)compressedImage.Length, new byte[] { 73, 68, 65, 84 }));
                }
            }

            Marshal.FreeHGlobal(filteredImage);
        }

        internal static unsafe MemoryStream CompressAPNGFrame(byte* image, int width, int height, bool hasAlpha, FilterModes filter, int threadCount)
        {
            //fDAT chunk
            IntPtr filteredImage;

            if (threadCount > 1)
            {
                filteredImage = FilterImageData(image, width, height, hasAlpha ? 4 : 3, filter, threadCount);
            }
            else
            {
                filteredImage = FilterImageData(image, width, height, hasAlpha ? 4 : 3, filter);
            }

            MemoryStream compressedImage = StreamUtils.ZLibCompress(filteredImage, height * (width * (hasAlpha ? 4 : 3) + 1));

            Marshal.FreeHGlobal(filteredImage);

            return compressedImage;
        }

        internal static void AddPreCompressedAPNGFrame(AnimatedPNG.CompressedFrame compressedFrame, int width, int height, bool hasAlpha, Stream fs, int frameNumber, ushort frameDelayNumerator, ushort frameDelayDenominator)
        {
            int fcTLIndex = frameNumber == 0 ? 0 : ((frameNumber - 1) * 2 + 1);
            int fDATIndex = frameNumber == 0 ? 0 : (frameNumber * 2);

            //fcTL chunk
            fs.WriteInt(26);
            using (MemoryStream fctl = new MemoryStream(30))
            {
                fctl.WriteASCIIString("fcTL");
                fctl.WriteInt(fcTLIndex);
                fctl.WriteInt(width);
                fctl.WriteInt(height);
                fctl.WriteInt(0);
                fctl.WriteInt(0);
                fctl.WriteUShort(frameDelayNumerator);
                fctl.WriteUShort(frameDelayDenominator);

                if (compressedFrame.InterframeCompression == AnimatedPNG.InterframeCompression.None || frameNumber == 0)
                {
                    fctl.WriteByte(0);
                    fctl.WriteByte(0);
                }
                else if (compressedFrame.InterframeCompression == AnimatedPNG.InterframeCompression.First)
                {
                    fctl.WriteByte(2);
                    fctl.WriteByte(1);
```

```
                }
                else if (compressedFrame.InterframeCompression == AnimatedPNG.InterframeCompression.Previous)
                {
                    fctl.WriteByte(0);
                    fctl.WriteByte(1);
                }

                fctl.Seek(0, SeekOrigin.Begin);
                fctl.CopyTo(fs);

                fs.WriteUInt(CRC32.ComputeCRC(fctl));
            }

            MemoryStream compressedImage = compressedFrame.compressedStream;

            if (frameNumber > 0)
            {
                fs.WriteUInt((uint)compressedImage.Length + 4);

                fs.WriteASCIIString("fdAT");
                fs.WriteInt(fDATIndex);

                compressedImage.Seek(0, SeekOrigin.Begin);
                compressedImage.CopyTo(fs);

                fs.WriteUInt(CRC32.ComputeCRC(compressedImage.GetBuffer(), (int)compressedImage.Length, new byte[] { 102, 100, 65, 84, (byte)(fDATIndex >> 24), (byte)((fDATIndex >> 16) & 255), (byte)((fDATIndex >> 8) & 255), (byte)(fDATIndex & 255) }));
            }
            else
            {
                compressedImage.Seek(0, SeekOrigin.Begin);
                fs.WriteUInt((uint)compressedImage.Length);
                fs.WriteASCIIString("IDAT");
                compressedImage.Seek(0, SeekOrigin.Begin);
                compressedImage.CopyTo(fs);

                fs.WriteUInt(CRC32.ComputeCRC(compressedImage.GetBuffer(), (int)compressedImage.Length, new byte[] { 73, 68, 65, 84 }));
            }
        }

        internal unsafe static void FinishAPNG(Stream fs)
        {
            //IEND chunk
            fs.WriteInt(0);
            fs.WriteASCIIString("IEND");
            fs.Write(new byte[] { 0xAE, 0x42, 0x60, 0x82 }, 0, 4);
        }

        internal enum FilterModes
        {
            None = 0,
            Sub = 1,
            Up = 2,
            Average = 3,
            Paeth = 4,
            Adaptive = -1
        }

        internal unsafe static void FilterRow(byte* image, int width, int pixelSize, int stride, int y, FilterModes filter, byte* destinationImage, byte* tempBuffer)
        {
            int startIndex = y * stride;
            int destinationIndex = y * (stride + 1);

            if (filter == FilterModes.None)
            {
                destinationImage[destinationIndex++] = 0;
                for (int x = 0; x < width; x++)
                {
                    destinationImage[destinationIndex++] = image[startIndex++];
                    destinationImage[destinationIndex++] = image[startIndex++];
                    destinationImage[destinationIndex++] = image[startIndex++];

                    if (pixelSize == 4)
                    {
                        destinationImage[destinationIndex++] = image[startIndex++];
                    }

                }
            }
            else if (filter == FilterModes.Sub)
            {
                destinationImage[destinationIndex++] = 1;
                for (int x = 0; x < width; x++)
```

```
                {
                    if (x > 0)
                    {
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize]);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize]);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize]);
                        if (pixelSize == 4)
                        {
                            destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize]);
                        }
                    }
                    else
                    {
                        destinationImage[destinationIndex++] = image[startIndex++];
                        destinationImage[destinationIndex++] = image[startIndex++];
                        destinationImage[destinationIndex++] = image[startIndex++];
                        if (pixelSize == 4)
                        {
                            destinationImage[destinationIndex++] = image[startIndex++];
                        }
                    }
                }
            }
            else if (filter == FilterModes.Up)
            {
                destinationImage[destinationIndex++] = 2;
                for (int x = 0; x < width; x++)
                {
                    if (y > 0)
                    {
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride]);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride]);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride]);
                        if (pixelSize == 4)
                        {
                            destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride]);
                        }
                    }
                    else
                    {
                        destinationImage[destinationIndex++] = image[startIndex++];
                        destinationImage[destinationIndex++] = image[startIndex++];
                        destinationImage[destinationIndex++] = image[startIndex++];
                        if (pixelSize == 4)
                        {
                            destinationImage[destinationIndex++] = image[startIndex++];
                        }
                    }
                }
            }
            else if (filter == FilterModes.Average)
            {
                destinationImage[destinationIndex++] = 3;
                for (int x = 0; x < width; x++)
                {
                    if (x > 0 && y > 0)
                    {
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - (image[startIndex - pixelSize] + image[startIndex++ - stride]) / 2);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - (image[startIndex - pixelSize] + image[startIndex++ - stride]) / 2);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - (image[startIndex - pixelSize] + image[startIndex++ - stride]) / 2);
                        if (pixelSize == 4)
                        {
                            destinationImage[destinationIndex++] = (byte)(image[startIndex] - (image[startIndex - pixelSize] + image[startIndex++ - stride]) / 2);
                        }
                    }
                    else if (x > 0 && y == 0)
                    {
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize] / 2);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize] / 2);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize] / 2);
                        if (pixelSize == 4)
                        {
```

```
                            destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize] / 2);
                        }
                    }
                    else if (x == 0 && y > 0)
                    {
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride] / 2);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride] / 2);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride] / 2);
                        if (pixelSize == 4)
                        {
                            destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride] / 2);
                        }
                    }
                    else
                    {
                        destinationImage[destinationIndex++] = image[startIndex++];
                        destinationImage[destinationIndex++] = image[startIndex++];
                        destinationImage[destinationIndex++] = image[startIndex++];
                        if (pixelSize == 4)
                        {
                            destinationImage[destinationIndex++] = image[startIndex++];
                        }
                    }
                }
            }
            else if (filter == FilterModes.Paeth)
            {
                destinationImage[destinationIndex++] = 4;
                for (int x = 0; x < width; x++)
                {
                    if (x > 0 && y > 0)
                    {
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - PaethPredictor(image[startIndex - pixelSize], image[startIndex - stride], image[startIndex++ - pixelSize - stride]));
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - PaethPredictor(image[startIndex - pixelSize], image[startIndex - stride], image[startIndex++ - pixelSize - stride]));
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - PaethPredictor(image[startIndex - pixelSize], image[startIndex - stride], image[startIndex++ - pixelSize - stride]));
                        if (pixelSize == 4)
                        {
                            destinationImage[destinationIndex++] = (byte)(image[startIndex] - PaethPredictor(image[startIndex - pixelSize], image[startIndex - stride], image[startIndex++ - pixelSize - stride]));
                        }
                    }
                    else if (x > 0 && y == 0)
                    {
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize]);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize]);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize]);
                        if (pixelSize == 4)
                        {
                            destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - pixelSize]);
                        }
                    }
                    else if (x == 0 && y > 0)
                    {
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride]);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride]);
                        destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride]);
                        if (pixelSize == 4)
                        {
                            destinationImage[destinationIndex++] = (byte)(image[startIndex] - image[startIndex++ - stride]);
                        }
                    }
                    else
                    {
                        destinationImage[destinationIndex++] = image[startIndex++];
                        destinationImage[destinationIndex++] = image[startIndex++];
                        destinationImage[destinationIndex++] = image[startIndex++];
                        if (pixelSize == 4)
```

```
                        {
                            destinationImage[destinationIndex++] = image[startIndex++];
                        }
                    }
                }
            }
            else if (filter == FilterModes.Adaptive)
            {
                int tempIndex = 0;

                int none = 0;
                int sub = 0;
                int up = 0;
                int average = 0;
                int paeth = 0;

                for (int x = 0; x < width; x++)
                {
                    for (int i = 0; i < pixelSize; i++)
                    {
                        //none
                        tempBuffer[tempIndex] = image[startIndex];

                        //sub
                        if (x > 0)
                        {
                            tempBuffer[tempIndex + stride] = (byte)(image[startIndex] - image[startIndex - pixelSize]);
                        }
                        else
                        {
                            tempBuffer[tempIndex + stride] = image[startIndex];
                        }

                        //up
                        if (y > 0)
                        {
                            tempBuffer[tempIndex + stride * 2] = (byte)(image[startIndex] - image[startIndex - stride]);
                        }
                        else
                        {
                            tempBuffer[tempIndex + stride * 2] = image[startIndex];
                        }

                        //average
                        if (x > 0 && y > 0)
                        {
                            tempBuffer[tempIndex + stride * 3] = (byte)(image[startIndex] - (image[startIndex - pixelSize] + image[startIndex - stride]) / 2);
                        }
                        else if (x > 0 && y == 0)
                        {
                            tempBuffer[tempIndex + stride * 3] = (byte)(image[startIndex] - image[startIndex - pixelSize] / 2);
                        }
                        else if (x == 0 && y > 0)
                        {
                            tempBuffer[tempIndex + stride * 3] = (byte)(image[startIndex] - image[startIndex - stride] / 2);
                        }
                        else
                        {
                            tempBuffer[tempIndex + stride * 3] = image[startIndex];
                        }

                        //paeth
                        if (x > 0 && y > 0)
                        {
                            tempBuffer[tempIndex + stride * 4] = (byte)(image[startIndex] - PaethPredictor(image[startIndex - pixelSize], image[startIndex - stride], image[startIndex - pixelSize - stride]));
                        }
                        else if (x > 0 && y == 0)
                        {
                            tempBuffer[tempIndex + stride * 4] = (byte)(image[startIndex] - image[startIndex - pixelSize]);
                        }
                        else if (x == 0 && y > 0)
                        {
                            tempBuffer[tempIndex + stride * 4] = (byte)(image[startIndex] - image[startIndex - stride]);
                        }
                        else
                        {
                            tempBuffer[tempIndex + stride * 4] = image[startIndex];
                        }
```

```

                        none += tempBuffer[tempIndex];
                        sub += tempBuffer[tempIndex + stride];
                        up += tempBuffer[tempIndex + stride * 2];
                        average += tempBuffer[tempIndex + stride * 3];
                        paeth += tempBuffer[tempIndex + stride * 4];
                        tempIndex++;
                        startIndex++;
                    }
                }

                if (paeth <= average && paeth <= up && paeth <= sub && paeth <= none)
                {
                    destinationImage[destinationIndex++] = 4;
                    tempIndex = stride * 4;
                    for (int i = 0; i < stride; i++)
                    {
                        destinationImage[destinationIndex++] = tempBuffer[tempIndex++];
                    }
                }
                else if (average <= up && average <= sub && average <= none)
                {
                    destinationImage[destinationIndex++] = 3;
                    tempIndex = stride * 3;
                    for (int i = 0; i < stride; i++)
                    {
                        destinationImage[destinationIndex++] = tempBuffer[tempIndex++];
                    }
                }
                else if (up <= sub && up <= none)
                {
                    destinationImage[destinationIndex++] = 2;
                    tempIndex = stride * 2;
                    for (int i = 0; i < stride; i++)
                    {
                        destinationImage[destinationIndex++] = tempBuffer[tempIndex++];
                    }
                }
                else if (sub <= none)
                {
                    destinationImage[destinationIndex++] = 1;
                    tempIndex = stride;
                    for (int i = 0; i < stride; i++)
                    {
                        destinationImage[destinationIndex++] = tempBuffer[tempIndex++];
                    }
                }
                else
                {
                    destinationImage[destinationIndex++] = 0;
                    tempIndex = 0;
                    for (int i = 0; i < stride; i++)
                    {
                        destinationImage[destinationIndex++] = tempBuffer[tempIndex++];
                    }
                }
            }
        }

        internal unsafe static IntPtr FilterImageData(byte* image, int width, int height, int pixelSize, FilterModes filter)
        {
            IntPtr filteredMemory = Marshal.AllocHGlobal(height * (1 + width * pixelSize));

            IntPtr tempBuffer;

            if (filter == FilterModes.Adaptive)
            {
                tempBuffer = Marshal.AllocHGlobal(5 * width * pixelSize);
            }
            else
            {
                tempBuffer = IntPtr.Zero;
            }

            int stride = width * pixelSize;

            byte* filteredPointer = (byte*)filteredMemory;

            byte* tempPointer = (byte*)tempBuffer;

            for (int y = 0; y < height; y++)
            {
                FilterRow(image, width, pixelSize, stride, y, filter, filteredPointer, tempPointer);
            }

            Marshal.FreeHGlobal(tempBuffer);
```

```

            return filteredMemory;
        }

        internal unsafe static IntPtr FilterImageData(byte* image, int width, int height, int pixelSize, FilterModes filter, int threadCount)
        {
            threadCount = Math.Min(threadCount, height);

            IntPtr filteredMemory = Marshal.AllocHGlobal(height * (1 + width * pixelSize));

            int stride = width * pixelSize;

            byte* filteredPointer = (byte*)filteredMemory;

            IntPtr[] tempBuffers = new IntPtr[threadCount];

            Thread[] threads = new Thread[threadCount];
            int[] rowsByThread = new int[threadCount];
            EventWaitHandle[] signalsFromThreads = new EventWaitHandle[threadCount];
            EventWaitHandle[] signalsToThreads = new EventWaitHandle[threadCount];

            int firstNeededRow = threadCount;

            for (int i = 0; i < threadCount; i++)
            {
                rowsByThread[i] = i;
                signalsFromThreads[i] = new EventWaitHandle(false, EventResetMode.ManualReset);
                signalsToThreads[i] = new EventWaitHandle(false, EventResetMode.ManualReset);

                if (filter == FilterModes.Adaptive)
                {
                    tempBuffers[i] = Marshal.AllocHGlobal(5 * width * pixelSize);
                }
                else
                {
                    tempBuffers[i] = IntPtr.Zero;
                }

                byte* tempPointer = (byte*)tempBuffers[i];

                int threadIndex = i;

                threads[i] = new Thread(() =>
                {
                    while (rowsByThread[threadIndex] >= 0)
                    {
                        FilterRow(image, width, pixelSize, stride, rowsByThread[threadIndex], filter, filteredPointer, tempPointer);
                        EventWaitHandle.SignalAndWait(signalsFromThreads[threadIndex], signalsToThreads[threadIndex]);
                        signalsToThreads[threadIndex].Reset();
                    }
                });
            }

            int finished = 0;

            for (int i = 0; i < threadCount; i++)
            {
                threads[i].Start();
            }

            while (finished < threadCount)
            {
                int threadInd = EventWaitHandle.WaitAny(signalsFromThreads);

                signalsFromThreads[threadInd].Reset();

                if (firstNeededRow < height)
                {
                    rowsByThread[threadInd] = firstNeededRow;
                    firstNeededRow++;
                    signalsToThreads[threadInd].Set();
                }
                else
                {
                    rowsByThread[threadInd] = -1;
                    signalsToThreads[threadInd].Set();
                    finished++;
                }
            }

            for (int i = 0; i < threadCount; i++)
            {
                if (filter == FilterModes.Adaptive)
                {
```

```
                    Marshal.FreeHGlobal(tempBuffers[i]);
                }
            }

            return filteredMemory;
        }

        //Based on http://www.libpng.org/pub/png/spec/1.2/PNG-Filters.html
        internal static byte PaethPredictor(byte a, byte b, byte c)
        {
            int p = (int)a + (int)b - (int)c;
            int pa = Math.Abs(p - (int)a);
            int pb = Math.Abs(p - (int)b);
            int pc = Math.Abs(p - (int)c);

            if (pa <= pb && pa <= pc)
            {
                return a;
            }
            else if (pb <= pc)
            {
                return b;
            }
            else
            {
                return c;
            }
        }
    }

    //Derived from http://www.libpng.org/pub/png/spec/1.2/PNG-CRCAppendix.html
    internal static class CRC32
    {
        /* Table of CRCs of all 8-bit messages. */
        private static readonly uint[] crc_table = new uint[256];

        /* Flag: has the table been computed? Initially false. */
        private static bool crc_table_computed = false;

        /* Make the table for a fast CRC. */
        private static void MakeCRCtable()
        {
            uint c;
            int n, k;

            for (n = 0; n < 256; n++)
            {
                c = (uint)n;
                for (k = 0; k < 8; k++)
                {
                    if ((c & 1) != 0)
                    {
                        c = 0xedb88320 ^ (c » 1);
                    }
                    else
                    {
                        c »= 1;
                    }
                }
                crc_table[n] = c;
            }
            crc_table_computed = true;
        }

        /* Update a running CRC with the bytes buf[0..len-1]--the CRC
           should be initialized to all 1's, and the transmitted value
           is the 1's complement of the final running CRC (see the
           crc() routine below)). */

        private static uint UpdateCRC(uint crc, Stream buf)
        {
            uint c = crc;
            int n;

            if (!crc_table_computed)
            {
                MakeCRCtable();
            }

            buf.Seek(0, SeekOrigin.Begin);

            for (n = 0; n < buf.Length; n++)
            {
                c = crc_table[(c ^ (byte)buf.ReadByte()) & 0xff] ^ (c » 8);
            }
            return c;
        }
```

```

        private static uint UpdateCRC(uint crc, byte[] buf, int length)
        {
            uint c = crc;
            int n;

            if (!crc_table_computed)
            {
                MakeCRCtable();
            }

            for (n = 0; n < length; n++)
            {
                c = crc_table[(c ^ buf[n]) & 0xff] ^ (c >> 8);
            }
            return c;
        }

        /* Return the CRC of the bytes buf[0..len-1]. */
        public static uint ComputeCRC(Stream buf)
        {
            return UpdateCRC(0xffffffff, buf) ^ 0xffffffff;
        }

        public static uint ComputeCRC(byte[] buf, int length, byte[] prefix)
        {
            uint temp = UpdateCRC(0xffffffff, prefix, prefix.Length);
            return UpdateCRC(temp, buf, length) ^ 0xffffffff;
        }
    }

    internal static class StreamUtils
    {
        public static void WriteASCIIString(this Stream sr, string val)
        {
            foreach (char c in val)
            {
                sr.WriteByte((byte)c);
            }
        }

        internal static MemoryStream ZLibCompress(IntPtr memoryAddress, int memoryLength)
        {
            MemoryStream compressedStream = new MemoryStream();
            compressedStream.Write(new byte[] { 0x78, 0x01 }, 0, 2);

            byte[] buffer = new byte[1024];

            AdlerHolder adlerHolder = new AdlerHolder();

            using (DeflateStream deflate = new DeflateStream(compressedStream,
CompressionLevel.Optimal, true))
            {
                while (memoryLength > 0)
                {
                    int currentBytes = Math.Min(memoryLength, 1024);
                    Marshal.Copy(memoryAddress, buffer, 0, currentBytes);
                    deflate.Write(buffer, 0, currentBytes);
                    memoryLength -= currentBytes;
                    memoryAddress = IntPtr.Add(memoryAddress, currentBytes);
                    Adler32(buffer, currentBytes, adlerHolder);
                }
            }


            uint checksum = (adlerHolder.s2 << 16) + adlerHolder.s1;

            compressedStream.Write(new byte[] { (byte)((checksum >> 24) & 255), (byte)((checksum >> 16)
& 255), (byte)((checksum >> 8) & 255), (byte)(checksum & 255) }, 0, 4);

            compressedStream.Seek(0, SeekOrigin.Begin);

            return compressedStream;
        }

        internal static uint Adler32(Stream contentStream)
        {
            uint s1 = 1;
            uint s2 = 0;

            int readByte;

            while ((readByte = contentStream.ReadByte()) >= 0)
            {
                s1 = (s1 + (byte)readByte) % 65521U;
                s2 = (s2 + s1) % 65521U;
            }
```

```
01083
01084             return (s2 « 16) + s1;
01085         }
01086
01087
01088         internal class AdlerHolder
01089         {
01090             public uint s1;
01091             public uint s2;
01092
01093             public AdlerHolder()
01094             {
01095                 s1 = 1;
01096                 s2 = 0;
01097             }
01098         }
01099
01100         internal static void Adler32(byte[] buffer, int length, AdlerHolder holder)
01101         {
01102             for (int i = 0; i < length; i++)
01103             {
01104                 holder.s1 = (holder.s1 + buffer[i]) % 65521U;
01105                 holder.s2 = (holder.s2 + holder.s1) % 65521U;
01106             }
01107         }
01108     }
01109 }
```

8.49 Point.cs

```
00001 /*
00002     VectSharp - A light library for C# vector graphics.
00003     Copyright (C) 2020-2022 Giorgio Bianchini
00004
00005     This program is free software: you can redistribute it and/or modify
00006     it under the terms of the GNU Lesser General Public License as published by
00007     the Free Software Foundation, version 3.
00008
00009     This program is distributed in the hope that it will be useful,
00010     but WITHOUT ANY WARRANTY; without even the implied warranty of
00011     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
00012     GNU Lesser General Public License for more details.
00013
00014     You should have received a copy of the GNU Lesser General Public License
00015     along with this program. If not, see <https://www.gnu.org/licenses/>.
00016 */
00017
00018 using System;
00019 using System.Collections.Generic;
00020 using System.Linq;
00021
00022 namespace VectSharp
00023 {
00024     /// <summary>
00025     /// Represents a point relative to an origin in the top-left corner.
00026     /// </summary>
00027     public struct Point
00028     {
00029         /// <summary>
00030         /// Horizontal (x) coordinate, measured to the right of the origin.
00031         /// </summary>
00032         public double X;
00033
00034         /// <summary>
00035         /// Vertical (y) coordinate, measured to the bottom of the origin.
00036         /// </summary>
00037         public double Y;
00038
00039         /// <summary>
00040         /// Create a new <see cref="Point"/>.
00041         /// </summary>
00042         /// <param name="x">The horizontal (x) coordinate.</param>
00043         /// <param name="y">The vertical (y) coordinate.</param>
00044         public Point(double x, double y)
00045         {
00046             X = x;
00047             Y = y;
00048         }
00049
00050         /// <summary>
00051         /// Computes the modulus of the vector represented by the <see cref="Point"/>.
00052         /// </summary>
00053         /// <returns>The modulus of the vector represented by the <see cref="Point"/>.</returns>
```

```
        public double Modulus()
        {
            return Math.Sqrt(X * X + Y * Y);
        }

        /// <summary>
        /// Normalises a <see cref="Point"/>.
        /// </summary>
        /// <returns>The normalised <see cref="Point"/>.</returns>
        public Point Normalize()
        {
            double mod = Modulus();
            return new Point(X / mod, Y / mod);
        }

        /// <summary>
        /// Checks whether this <see cref="Point"/> is equal to another <see cref="Point"/>, up to a specified tolerance.
        /// </summary>
        /// <param name="p2">The <see cref="Point"/> to compare.</param>
        /// <param name="tolerance">The tolerance threshold.</param>
        /// <returns><see langword="true"/> if both coordinates of the <see cref="Point"/>s are closer than <paramref name="tolerance"/> or if their relative difference (i.e. <c>(a - b) / (a + b) * 2</c>) is smaller than <paramref name="tolerance"/>. <see langword="false"/> otherwise.</returns>
        public bool IsEqual(Point p2, double tolerance)
        {
            return (Math.Abs(p2.X - this.X) <= tolerance || Math.Abs((p2.X - this.X) / (p2.X + this.X)) <= tolerance * 0.5) && (Math.Abs(p2.Y - this.Y) <= tolerance || Math.Abs((p2.Y - this.Y) / (p2.Y + this.Y)) <= tolerance * 0.5);
        }

        /// <summary>
        /// Computes the top-left corner of the <see cref="Rectangle"/> identified by two <see cref="Point"/>s.
        /// </summary>
        /// <param name="p1">The first point.</param>
        /// <param name="p2">The second point.</param>
        /// <returns>A <see cref="Point"/> whose <see cref="X"/> coordinate is the smallest between the one of <paramref name="p1"/> and <paramref name="p2"/>, and likewise for the <see cref="Y"/> coordinate.</returns>
        public static Point Min(Point p1, Point p2)
        {
            return new Point(Math.Min(p1.X, p2.X), Math.Min(p1.Y, p2.Y));
        }

        /// <summary>
        /// Computes the bottom-right corner of the <see cref="Rectangle"/> identified by two <see cref="Point"/>s.
        /// </summary>
        /// <param name="p1">The first point.</param>
        /// <param name="p2">The second point.</param>
        /// <returns>A <see cref="Point"/> whose <see cref="X"/> coordinate is the largest between the one of <paramref name="p1"/> and <paramref name="p2"/>, and likewise for the <see cref="Y"/> coordinate.</returns>
        public static Point Max(Point p1, Point p2)
        {
            return new Point(Math.Max(p1.X, p2.X), Math.Max(p1.Y, p2.Y));
        }

        /// <summary>
        /// Computes the smallest <see cref="Rectangle"/> that contains all the specified points.
        /// </summary>
        /// <param name="points">The points whose bounds are being computed.</param>
        /// <returns>The smallest <see cref="Rectangle"/> that contains all the specified points.</returns>
        public static Rectangle Bounds(IEnumerable<Point> points)
        {
            bool initialised = false;
            Point min = new Point(double.NaN, double.NaN);
            Point max = new Point(double.NaN, double.NaN);

            foreach (Point pt in points)
            {
                if (!initialised)
                {
                    min = pt;
                    max = pt;
                    initialised = true;
                }
                else
                {
                    min = Point.Min(min, pt);
                    max = Point.Max(max, pt);
                }
            }

            return new Rectangle(min, max);
```

```
        }

        /// <summary>
        /// Computes the smallest <see cref="Rectangle"/> that contains all the specified points.
        /// </summary>
        /// <param name="points">The points whose bounds are being computed.</param>
        /// <returns>The smallest <see cref="Rectangle"/> that contains all the specified points.</returns>
        public static Rectangle Bounds(params Point[] points)
        {
            return Bounds((IEnumerable<Point>)points);
        }
    }

    /// <summary>
    /// Represents the size of an object.
    /// </summary>
    public struct Size
    {
        /// <summary>
        /// Width of the object.
        /// </summary>
        public double Width;

        /// <summary>
        /// Height of the object.
        /// </summary>
        public double Height;

        /// <summary>
        /// Create a new <see cref="Size"/>.
        /// </summary>
        /// <param name="width">The width of the object.</param>
        /// <param name="height">The height of the object.</param>
        public Size(double width, double height)
        {
            Width = width;
            Height = height;
        }
    }

    /// <summary>
    /// Represents a rectangle.
    /// </summary>
    public struct Rectangle
    {
        /// <summary>
        /// A rectangle whose dimensions are all <see cref="double.NaN"/>.
        /// </summary>
        public static readonly Rectangle NaN = new Rectangle(double.NaN, double.NaN, double.NaN, double.NaN);

        /// <summary>
        /// Checks whether the rectangle is equivalent to <see cref="Rectangle.NaN"/>.
        /// </summary>
        /// <returns><see langword="true"/> if all the dimensions of the rectangle are <see cref="double.NaN"/>, <see langword="false"/> otherwise.</returns>
        public bool IsNaN()
        {
            return double.IsNaN(this.Location.X) && double.IsNaN(this.Location.Y) && double.IsNaN(this.Size.Width) && double.IsNaN(this.Size.Height);
        }

        /// <summary>
        /// The top-left corner of the rectangle.
        /// </summary>
        public Point Location;

        /// <summary>
        /// The size of the rectangle.
        /// </summary>
        public Size Size;

        /// <summary>
        /// The centre of the rectangle.
        /// </summary>
        public Point Centre { get { return new Point(this.Location.X + this.Size.Width * 0.5, this.Location.Y + this.Size.Height * 0.5); } }

        /// <summary>
        /// Create a new <see cref="Rectangle"/> given its top-left corner and its size.
        /// </summary>
        /// <param name="location">The top-left corner of the rectangle.</param>
        /// <param name="size">The size of the rectangle.</param>
        public Rectangle(Point location, Size size)
        {
            this.Location = location;
```

```
            this.Size = size;
        }

        /// <summary>
        /// Create a new <see cref="Rectangle"/> given its top-left corner and its size.
        /// </summary>
        /// <param name="x">The horizontal coordinate of the top-left corner of the rectangle.</param>
        /// <param name="y">The vertical coordinate of the top-left corner of the rectangle.</param>
        /// <param name="width">The width of the rectangle.</param>
        /// <param name="height">The height of the rectangle.</param>
        public Rectangle(double x, double y, double width, double height)
        {
            Location = new Point(x, y);
            Size = new Size(width, height);
        }

        /// <summary>
        /// Create a new <see cref="Rectangle"/> given its top-left corner and its bottom-right corner.
        /// </summary>
        /// <param name="topLeft">The top-left corner of the rectangle.</param>
        /// <param name="bottomRight">The bottom-right corner of the rectangle.</param>
        public Rectangle(Point topLeft, Point bottomRight)
        {
            this.Location = topLeft;
            this.Size = new Size(bottomRight.X - topLeft.X, bottomRight.Y - topLeft.Y);
        }

        /// <summary>
        /// Computes the rectangular bounds of the union of two <see cref="Rectangle"/>s.
        /// </summary>
        /// <param name="rectangle1">The first <see cref="Rectangle"/>.</param>
        /// <param name="rectangle2">The second <see cref="Rectangle"/>.</param>
        /// <returns>The smallest <see cref="Rectangle"/> containing both <paramref name="rectangle1"/> and <paramref name="rectangle2"/>.</returns>
        public static Rectangle Union(Rectangle rectangle1, Rectangle rectangle2)
        {
            double minX = rectangle1.Location.X;
            double minY = rectangle1.Location.Y;
            double maxX = rectangle1.Location.X;
            double maxY = rectangle1.Location.Y;

            minX = Math.Min(minX, rectangle1.Location.X + rectangle1.Size.Width);
            minX = Math.Min(minX, rectangle2.Location.X);
            minX = Math.Min(minX, rectangle2.Location.X + rectangle2.Size.Width);

            minY = Math.Min(minY, rectangle1.Location.Y + rectangle1.Size.Height);
            minY = Math.Min(minY, rectangle2.Location.Y);
            minY = Math.Min(minY, rectangle2.Location.Y + rectangle2.Size.Height);


            maxX = Math.Max(maxX, rectangle1.Location.X + rectangle1.Size.Width);
            maxX = Math.Max(maxX, rectangle2.Location.X);
            maxX = Math.Max(maxX, rectangle2.Location.X + rectangle2.Size.Width);

            maxY = Math.Max(maxY, rectangle1.Location.Y + rectangle1.Size.Height);
            maxY = Math.Max(maxY, rectangle2.Location.Y);
            maxY = Math.Max(maxY, rectangle2.Location.Y + rectangle2.Size.Height);

            return new Rectangle(minX, minY, maxX - minX, maxY - minY);
        }

        /// <summary>
        /// Computes the intersection of two <see cref="Rectangle"/>s.
        /// </summary>
        /// <param name="rectangle1">The first <see cref="Rectangle"/>.</param>
        /// <param name="rectangle2">The second <see cref="Rectangle"/>.</param>
        /// <returns>The rectangle corresponding to the intersection of <paramref name="rectangle1"/> and <paramref name="rectangle2"/>, or <see cref="Rectangle.NaN"/> if the intersection is empty.</returns>
        public static Rectangle Intersection(Rectangle rectangle1, Rectangle rectangle2)
        {
            double x0 = Math.Max(rectangle1.Location.X, rectangle2.Location.X);
            double x1 = Math.Min(rectangle1.Location.X + rectangle1.Size.Width, rectangle2.Location.X + rectangle2.Size.Width);

            double y0 = Math.Max(rectangle1.Location.Y, rectangle2.Location.Y);
            double y1 = Math.Min(rectangle1.Location.Y + rectangle1.Size.Height, rectangle2.Location.Y + rectangle2.Size.Height);

            if (x1 >= x0 && y1 >= y0)
            {
                return new Rectangle(x0, y0, x1 - x0, y1 - y0);
            }
            else
            {
                return Rectangle.NaN;
```

```
            }

        }

        /// <summary>
        /// Computes the rectangular bounds of the union of multiple <see cref="Rectangle"/>s.
        /// </summary>
        /// <param name="rectangles">The <see cref="Rectangle"/>s whose union will be computed.</param>
        /// <returns>The smallest <see cref="Rectangle"/> containing all the <paramref name="rectangles"/>.</returns>
        public static Rectangle Union(IEnumerable<Rectangle> rectangles)
        {
            if (rectangles.Any())
            {
                bool initialised = false;

                Rectangle tbr = new Rectangle();

                foreach (Rectangle rect in rectangles)
                {
                    if (!initialised)
                    {
                        tbr = rect;
                    }
                    else
                    {
                        tbr = Union(rect, tbr);
                    }
                }

                return tbr;
            }
            else
            {
                return Rectangle.NaN;
            }
        }

        /// <summary>
        /// Computes the rectangular bounds of the union of multiple <see cref="Rectangle"/>s.
        /// </summary>
        /// <param name="rectangles">The <see cref="Rectangle"/>s whose union will be computed.</param>
        /// <returns>The smallest <see cref="Rectangle"/> containing all the <paramref name="rectangles"/>.</returns>
        public static Rectangle Union(params Rectangle[] rectangles)
        {
            return Union((IEnumerable<Rectangle>)rectangles);
        }
    }
}
```

8.50 RasterImage.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.IO;
using System.Runtime.InteropServices;

namespace VectSharp
{
    /// <summary>
    /// Represents the pixel format of a raster image.
    /// </summary>
    public enum PixelFormats
    {
```

```
        /// <summary>
        /// RGB 24bpp format.
        /// </summary>
        RGB,

        /// <summary>
        /// RGBA 32bpp format.
        /// </summary>
        RGBA,

        /// <summary>
        /// BGR 24bpp format.
        /// </summary>
        BGR,

        /// <summary>
        /// BGR 32bpp format.
        /// </summary>
        BGRA
    }

    /// <summary>
    /// An <see cref="IDisposable"/> wrapper around an <see cref="IntPtr"/> that frees the allocated memory when it is disposed.
    /// </summary>
    public class DisposableIntPtr : IDisposable
    {
        /// <summary>
        /// The pointer to the unmanaged memory.
        /// </summary>
        public readonly IntPtr InternalPointer;

        /// <summary>
        /// Create a new DisposableIntPtr.
        /// </summary>
        /// <param name="pointer">The pointer that should be freed upon disposing of this object.</param>
        public DisposableIntPtr(IntPtr pointer)
        {
            this.InternalPointer = pointer;
        }

        private bool disposedValue;

        ///<inheritdoc/>
        protected virtual void Dispose(bool disposing)
        {
            if (!disposedValue)
            {
                Marshal.FreeHGlobal(InternalPointer);
                disposedValue = true;
            }
        }

        ///<inheritdoc/>
        ~DisposableIntPtr()
        {
            Dispose(disposing: false);
        }

        ///<inheritdoc/>
        public void Dispose()
        {
            Dispose(disposing: true);
            GC.SuppressFinalize(this);
        }
    }

    /// <summary>
    /// Represents a raster image, created from raw pixel data. Consider using the derived classes included in the NuGet package "VectSharp.MuPDFUtils" if you need to load a raster image from a file or a <see cref="Stream"/>.
    /// </summary>
    public class RasterImage : IDisposable
    {
        /// <summary>
        /// The memory address of the image pixel data.
        /// </summary>
        public IntPtr ImageDataAddress { get; protected set; }

        /// <summary>
        /// An <see cref="IDisposable"/> that will be disposed when the image is disposed.
        /// </summary>
        public IDisposable DataHolder { get; protected set; }

        /// <summary>
        /// A univocal identifier for this image.
```

```
        /// </summary>
        public string Id { get; protected set; }

        /// <summary>
        /// Determines whether the image has an alpha channel.
        /// </summary>
        public bool HasAlpha { get; protected set; }

        /// <summary>
        /// The width in pixels of the image.
        /// </summary>
        public int Width { get; protected set; }

        /// <summary>
        /// The height in pixels of the image.
        /// </summary>
        public int Height { get; protected set; }

        /// <summary>
        /// Determines whether the image should be interpolated when it is resized.
        /// </summary>
        public bool Interpolate { get; protected set; }

        private MemoryStream _PNGStream = null;

        /// <summary>
        /// Contains a representation of the image in PNG format. Generated at the first access and cached until the image is disposed.
        /// </summary>
        public MemoryStream PNGStream
        {
            get
            {
                if (_PNGStream == null)
                {
                    _PNGStream = new MemoryStream();
                    this.EncodeAsPNG(_PNGStream);
                }
                _PNGStream.Seek(0, SeekOrigin.Begin);
                return _PNGStream;
            }
        }

        /// <summary>
        /// Default constructor, necessary for inheritance.
        /// </summary>
        protected RasterImage()
        {

        }

        /// <summary>
        /// Creates a new <see cref="RasterImage"/> instance from the specified pixel data in RGB or RGBA format.
        /// </summary>
        /// <param name="pixelData">The address of the image pixel data in RGB or RGBA format.</param>
        /// <param name="width">The width in pixels of the image.</param>
        /// <param name="height">The height in pixels of the image.</param>
        /// <param name="hasAlpha">true if the image is in RGBA format, false if it is in RGB format.</param>
        /// <param name="interpolate">Whether the image should be interpolated when it is resized.</param>
        public RasterImage(IntPtr pixelData, int width, int height, bool hasAlpha, bool interpolate)
        {
            this.Id = Guid.NewGuid().ToString();
            this.ImageDataAddress = pixelData;
            this.Width = width;
            this.Height = height;
            this.HasAlpha = hasAlpha;
            this.Interpolate = interpolate;
        }

        /// <summary>
        /// Creates a new <see cref="RasterImage"/> instance from the specified pixel data in RGB or RGBA format.
        /// </summary>
        /// <param name="pixelData">The address of the image pixel data in RGB or RGBA format wrapped in a <see cref="DisposableIntPtr"/>. The <see cref="RasterImage"/> will take ownership of this memory.</param>
        /// <param name="width">The width in pixels of the image.</param>
        /// <param name="height">The height in pixels of the image.</param>
        /// <param name="hasAlpha">true if the image is in RGBA format, false if it is in RGB format.</param>
        /// <param name="interpolate">Whether the image should be interpolated when it is resized.</param>
        public RasterImage(ref DisposableIntPtr pixelData, int width, int height, bool hasAlpha, bool interpolate)
```

```
        {
            this.Id = Guid.NewGuid().ToString();
            this.ImageDataAddress = pixelData.InternalPointer;
            this.DataHolder = pixelData;
            this.Width = width;
            this.Height = height;
            this.HasAlpha = hasAlpha;
            this.Interpolate = interpolate;
        }

        /// <summary>
        /// Creates a new <see cref="RasterImage"/> instance copying the specified pixel data.
        /// </summary>
        /// <param name="data">The image pixel data that will be copied.</param>
        /// <param name="width">The width in pixels of the image.</param>
        /// <param name="height">The height in pixels of the image.</param>
        /// <param name="pixelFormat">The format of the pixel data.</param>
        /// <param name="interpolate">Whether the image should be interpolated when it is resized.</param>
        public RasterImage(byte[] data, int width, int height, PixelFormats pixelFormat, bool interpolate)
        {
            this.ImageDataAddress = Marshal.AllocHGlobal(data.Length);
            GC.AddMemoryPressure(data.Length);
            this.DataHolder = new DisposableIntPtr(this.ImageDataAddress);
            this.Id = Guid.NewGuid().ToString();
            this.Width = width;
            this.Height = height;
            this.Interpolate = interpolate;

            switch (pixelFormat)
            {
                case PixelFormats.RGB:
                case PixelFormats.RGBA:
                    Marshal.Copy(data, 0, this.ImageDataAddress, data.Length);
                    this.HasAlpha = pixelFormat == PixelFormats.RGBA;
                    break;

                case PixelFormats.BGRA:
                case PixelFormats.BGR:
                    this.HasAlpha = pixelFormat == PixelFormats.BGRA;

                    int pixelSize = pixelFormat == PixelFormats.BGRA ? 4 : 3;
                    int pixelCount = width * height;

                    unsafe
                    {
                        byte* dataPointer = (byte*)this.ImageDataAddress;
                        for (int i = 0; i < pixelCount; i++)
                        {
                            dataPointer[i * pixelSize] = data[i * pixelSize + 2];
                            dataPointer[i * pixelSize + 1] = data[i * pixelSize + 1];
                            dataPointer[i * pixelSize + 2] = data[i * pixelSize];
                            if (pixelSize == 4)
                            {
                                dataPointer[i * pixelSize + 3] = data[i * pixelSize + 3];
                            }
                        }
                    }
                    break;
            }
        }

        private void EncodeAsPNG(Stream outputStream)
        {
            unsafe
            {
                PNGEncoder.SavePNG((byte*)this.ImageDataAddress, this.Width, this.Height, this.HasAlpha, outputStream, PNGEncoder.FilterModes.Adaptive);
            }
        }

        /// <summary>
        /// Disposes the <see cref="PNGStream"/>. Also useful if is is necessary to regenerate it, e.g. because the underlying image pixel data has changed.
        /// </summary>
        public void ClearPNGCache()
        {
            _PNGStream?.Dispose();
            _PNGStream = null;
        }

        private bool disposedValue;

        /// <inheritdoc/>
        protected virtual void Dispose(bool disposing)
        {
```

```
            if (!disposedValue)
            {
                if (disposing)
                {
                    _PNGStream?.Dispose();

                    if (DataHolder != null)
                    {
                        DataHolder.Dispose();
                        GC.RemoveMemoryPressure(Height * Width * (HasAlpha ? 4 : 3));
                    }
                }
                disposedValue = true;
            }
        }

        ///<inheritdoc/>
        ~RasterImage()
        {
            Dispose(disposing: false);
        }

        /// <inheritdoc/>
        public void Dispose()
        {
            Dispose(disposing: true);
            GC.SuppressFinalize(this);
        }
    }
}
```

8.51 Segment.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Linq;

namespace VectSharp
{

    /// <summary>
    /// Represents a segment as part of a <see cref="GraphicsPath"/>.
    /// </summary>
    public abstract class Segment
    {

        /// <summary>
        /// The type of the <see cref="Segment"/>.
        /// </summary>
        public abstract SegmentType Type { get; }

        /// <summary>
        /// The points used to define the <see cref="Segment"/>.
        /// </summary>
        public Point[] Points { get; protected set; }

        /// <summary>
        /// The end point of the <see cref="Segment"/>.
        /// </summary>
        public virtual Point Point
        {
            get
            {
                return Points[Points.Length - 1];
            }
        }
```

```
        /// <summary>
        /// Creates a copy of the <see cref="Segment"/>.
        /// </summary>
        /// <returns>A copy of the <see cref="Segment"/>.</returns>
        public abstract Segment Clone();

        /// <summary>
        /// Computes the length of the <see cref="Segment"/>.
        /// </summary>
        /// <param name="previousPoint">The point from which the <see cref="Segment"/> starts (i.e. the endpoint of the previous <see cref="Segment"/>).</param>
        /// <returns>The length of the segment.</returns>
        public abstract double Measure(Point previousPoint);

        /// <summary>
        /// Gets the point on the <see cref="Segment"/> at the specified (relative) <paramref name="position"/>).
        /// </summary>
        /// <param name="previousPoint">The point from which the <see cref="Segment"/> starts (i.e. the endpoint of the previous <see cref="Segment"/>).</param>
        /// <param name="position">The relative position on the <see cref="Segment"/> (0 is the start of the <see cref="Segment"/>, 1 is the end of the <see cref="Segment"/>).</param>
        /// <returns>The point at the specified position.</returns>
        public abstract Point GetPointAt(Point previousPoint, double position);

        /// <summary>
        /// Gets the tangent to the <see cref="Segment"/> at the specified (relative) <paramref name="position"/>).
        /// </summary>
        /// <param name="previousPoint">The point from which the <see cref="Segment"/> starts (i.e. the endpoint of the previous <see cref="Segment"/>).</param>
        /// <param name="position">The relative position on the <see cref="Segment"/> (0 is the start of the <see cref="Segment"/>, 1 is the end of the <see cref="Segment"/>).</param>
        /// <returns>The tangent to the point at the specified position.</returns>
        public abstract Point GetTangentAt(Point previousPoint, double position);

        /// <summary>
        /// Transform the segment into a series of linear segments. Segments that are already linear are not changed.
        /// </summary>
        /// <param name="previousPoint">The point from which the <see cref="Segment"/> starts (i.e. the endpoint of the previous <see cref="Segment"/>).</param>
        /// <param name="resolution">The absolute length between successive samples in curve segments.</param>
        /// <returns>A collection of linear segments that approximate the current segment.</returns>
        public abstract IEnumerable<Segment> Linearise(Point? previousPoint, double resolution);

        /// <summary>
        /// Gets the tanget at the points at which the segment would be linearised.
        /// </summary>
        /// <param name="previousPoint">The point from which the <see cref="Segment"/> starts (i.e. the endpoint of the previous <see cref="Segment"/>).</param>
        /// <param name="resolution">The absolute length between successive samples in curve segments.</param>
        /// <returns>A collection of tangents at the points in which the segment would be linearised.</returns>
        public abstract IEnumerable<Point> GetLinearisationTangents(Point? previousPoint, double resolution);

        /// <summary>
        /// Applies an arbitrary transformation to all of the points of the <see cref="Segment"/>.
        /// </summary>
        /// <param name="transformationFunction">An arbitrary transformation function.</param>
        /// <returns>A collection of <see cref="Segment"/>s that have been transformed according to the <paramref name="transformationFunction"/>.</returns>
        public abstract IEnumerable<Segment> Transform(Func<Point, Point> transformationFunction);
    }

    internal class MoveSegment : Segment
    {
        public override SegmentType Type => SegmentType.Move;

        public MoveSegment(Point p)
        {
            this.Points = new Point[] { p };
        }

        public MoveSegment(double x, double y)
        {
            this.Points = new Point[] { new Point(x, y) };
        }

        public override Segment Clone()
        {
            return new MoveSegment(this.Point);
        }
```

```

        public override double Measure(Point previousPoint)
        {
            return 0;
        }

        public override Point GetPointAt(Point previousPoint, double position)
        {
            throw new InvalidOperationException();
        }

        public override Point GetTangentAt(Point previousPoint, double position)
        {
            throw new InvalidOperationException();
        }

        public override IEnumerable<Segment> Linearise(Point? previousPoint, double resolution)
        {
            yield return new MoveSegment(this.Point);
        }

        public override IEnumerable<Point> GetLinearisationTangents(Point? previousPoint, double resolution)
        {
            throw new InvalidOperationException();
        }

        public override IEnumerable<Segment> Transform(Func<Point, Point> transformationFunction)
        {
            yield return new MoveSegment(transformationFunction(this.Point));
        }
    }

    internal class LineSegment : Segment
    {
        public override SegmentType Type => SegmentType.Line;

        public LineSegment(Point p)
        {
            this.Points = new Point[] { p };
        }

        public LineSegment(double x, double y)
        {
            this.Points = new Point[] { new Point(x, y) };
        }

        public override Segment Clone()
        {
            return new LineSegment(this.Point);
        }

        private double cachedLength = double.NaN;

        public override double Measure(Point previousPoint)
        {
            if (double.IsNaN(cachedLength))
            {
                cachedLength = Math.Sqrt((this.Point.X - previousPoint.X) * (this.Point.X - previousPoint.X) + (this.Point.Y - previousPoint.Y) * (this.Point.Y - previousPoint.Y));
            }

            return cachedLength;
        }

        public override Point GetPointAt(Point previousPoint, double position)
        {
            return new Point(previousPoint.X * (1 - position) + this.Point.X * position, previousPoint.Y * (1 - position) + this.Point.Y * position);
        }

        public override Point GetTangentAt(Point previousPoint, double position)
        {
            return new Point(this.Point.X - previousPoint.X, this.Point.Y - previousPoint.Y).Normalize();
        }

        public override IEnumerable<Segment> Linearise(Point? previousPoint, double resolution)
        {
            yield return new LineSegment(this.Point);
        }

        public override IEnumerable<Point> GetLinearisationTangents(Point? previousPoint, double resolution)
        {
            yield return this.GetTangentAt(previousPoint.Value, 1);
        }
```

```
        public override IEnumerable<Segment> Transform(Func<Point, Point> transformationFunction)
        {
            yield return new LineSegment(transformationFunction(this.Point));
        }
    }

    internal class CloseSegment : Segment
    {
        public override SegmentType Type => SegmentType.Close;

        public CloseSegment() { }

        public override Segment Clone()
        {
            return new CloseSegment();
        }

        public override double Measure(Point previousPoint)
        {
            return 0;
        }

        public override Point GetPointAt(Point previousPoint, double position)
        {
            throw new InvalidOperationException();
        }

        public override Point GetTangentAt(Point previousPoint, double position)
        {
            throw new InvalidOperationException();
        }

        public override IEnumerable<Segment> Linearise(Point? previousPoint, double resolution)
        {
            yield return new CloseSegment();
        }

        public override IEnumerable<Point> GetLinearisationTangents(Point? previousPoint, double resolution)
        {
            throw new InvalidOperationException();
        }

        public override IEnumerable<Segment> Transform(Func<Point, Point> transformationFunction)
        {
            yield return new CloseSegment();
        }
    }

    internal class CubicBezierSegment : Segment
    {
        public override SegmentType Type => SegmentType.CubicBezier;
        public CubicBezierSegment(double x1, double y1, double x2, double y2, double x3, double y3)
        {
            Points = new Point[] { new Point(x1, y1), new Point(x2, y2), new Point(x3, y3) };
        }

        public CubicBezierSegment(Point p1, Point p2, Point p3)
        {
            Points = new Point[] { p1, p2, p3 };
        }

        public override Segment Clone()
        {
            return new CubicBezierSegment(Points[0], Points[1], Points[2]);
        }

        private double cachedLength = double.NaN;
        private int cachedSegments = -1;

        public override double Measure(Point previousPoint)
        {
            if (double.IsNaN(cachedLength))
            {
                int segments = 16;
                double prevLength = 0;
                double currLength = Measure(previousPoint, segments);

                while (currLength > 0.00001 && Math.Abs(currLength - prevLength) / currLength > 0.0001)
                {
                    segments *= 2;
                    prevLength = currLength;
                    currLength = Measure(previousPoint, segments);
                }

```

```
                cachedSegments = segments;

                cachedLength = currLength;
            }

            return cachedLength;
        }

        public Point GetBezierPointAt(Point previousPoint, double position)
        {
            if (position <= 1 && position >= 0)
            {
                return new Point(
                this.Points[2].X * position * position * position + 3 * this.Points[1].X * position * position * (1 - position) + 3 * this.Points[0].X * position * (1 - position) * (1 - position) + previousPoint.X * (1 - position) * (1 - position) * (1 - position),
                this.Points[2].Y * position * position * position + 3 * this.Points[1].Y * position * position * (1 - position) + 3 * this.Points[0].Y * position * (1 - position) * (1 - position) + previousPoint.Y * (1 - position) * (1 - position) * (1 - position)
                );
            }
            else if (position > 1)
            {
                Point tangent = GetBezierTangentAt(previousPoint, 1);

                double excessLength = (position - 1) * this.Measure(previousPoint);

                return new Point(this.Point.X + tangent.X * excessLength, this.Point.Y + tangent.Y * excessLength);
            }
            else
            {
                Point tangent = GetBezierTangentAt(previousPoint, 0);

                return new Point(previousPoint.X + tangent.X * position * this.Measure(previousPoint), previousPoint.Y + tangent.Y * position * this.Measure(previousPoint));
            }
        }

        public override Point GetPointAt(Point previousPoint, double position)
        {
            double t = GetTFromPosition(previousPoint, position);
            return this.GetBezierPointAt(previousPoint, t);
        }

        public override Point GetTangentAt(Point previousPoint, double position)
        {
            double t = GetTFromPosition(previousPoint, position);
            return this.GetBezierTangentAt(previousPoint, t);
        }

        private double Measure(Point startPoint, int segments)
        {
            double delta = 1.0 / segments;

            double tbr = 0;

            for (int i = 1; i < segments; i++)
            {
                Point p1 = GetBezierPointAt(startPoint, delta * (i - 1));
                Point p2 = GetBezierPointAt(startPoint, delta * i);

                tbr += Math.Sqrt((p1.X - p2.X) * (p1.X - p2.X) + (p1.Y - p2.Y) * (p1.Y - p2.Y));
            }

            return tbr;
        }

        private double Measure(Point startPoint, int segments, double maxT)
        {
            double delta = maxT / segments;

            double tbr = 0;

            for (int i = 1; i < segments; i++)
            {
                Point p1 = GetBezierPointAt(startPoint, delta * (i - 1));
                Point p2 = GetBezierPointAt(startPoint, delta * i);

                tbr += Math.Sqrt((p1.X - p2.X) * (p1.X - p2.X) + (p1.Y - p2.Y) * (p1.Y - p2.Y));
            }

            return tbr;
        }

        private Point GetBezierTangentAt(Point previousPoint, double position)
        {
```

```
            if (position <= 1 && position >= 0)
            {
                return new Point(
                    3 * this.Points[2].X * position * position +
                    3 * this.Points[1].X * position * (2 - 3 * position) +
                    3 * this.Points[0].X * (3 * position * position - 4 * position + 1) +
                    -3 * previousPoint.X * (1 - position) * (1 - position),

                    3 * this.Points[2].Y * position * position +
                    3 * this.Points[1].Y * position * (2 - 3 * position) +
                    3 * this.Points[0].Y * (3 * position * position - 4 * position + 1) +
                    -3 * previousPoint.Y * (1 - position) * (1 - position)).Normalize();
            }
            else if (position > 1)
            {
                return GetBezierTangentAt(previousPoint, 1);
            }
            else
            {
                return GetBezierTangentAt(previousPoint, 0);
            }
        }

        private double GetTFromPosition(Point previousPoint, double position)
        {
            if (position <= 0 || position >= 1)
            {
                return position;
            }
            else
            {
                double length = this.Measure(previousPoint);

                double lowerBound = 0;
                double upperBound = 0.5;

                double lowerPos = 0;
                double upperPos = Measure(previousPoint, (int)Math.Ceiling(this.cachedSegments * upperBound), upperBound) / length;

                if (upperPos < position)
                {
                    lowerBound = upperBound;
                    lowerPos = upperPos;

                    upperBound = 1;
                    upperPos = 1;
                }

                while (Math.Min(upperPos - position, position - lowerPos) > 0.001)
                {
                    double mid = (lowerBound + upperBound) * 0.5;
                    double midPos = Measure(previousPoint, (int)Math.Ceiling(this.cachedSegments * mid), mid) / length;

                    if (midPos > position)
                    {
                        upperBound = mid;
                        upperPos = midPos;
                    }
                    else
                    {
                        lowerBound = mid;
                        lowerPos = midPos;
                    }
                }

                return lowerBound + (position - lowerPos) / (upperPos - lowerPos) * (upperBound - lowerBound);
            }
        }

        public override IEnumerable<Segment> Linearise(Point? previousPoint, double resolution)
        {
            double length = this.Measure(previousPoint.Value);
            int segmentCount = (int)Math.Ceiling(length / resolution);

            for (int i = 0; i < segmentCount; i++)
            {
                yield return new LineSegment(this.GetPointAt(previousPoint.Value, (double)(i + 1) / segmentCount));
            }
        }

        public override IEnumerable<Point> GetLinearisationTangents(Point? previousPoint, double resolution)
        {
```

```
            double length = this.Measure(previousPoint.Value);
            int segmentCount = (int)Math.Ceiling(length / resolution);

            for (int i = 0; i < segmentCount; i++)
            {
                yield return this.GetTangentAt(previousPoint.Value, (double)(i + 1) / segmentCount);
            }
        }

        public override IEnumerable<Segment> Transform(Func<Point, Point> transformationFunction)
        {
            yield return new CubicBezierSegment(transformationFunction(this.Points[0]), transformationFunction(this.Points[1]), transformationFunction(this.Points[2]));
        }
    }

    internal class ArcSegment : Segment
    {
        public override SegmentType Type => SegmentType.Arc;

        public Segment[] ToBezierSegments()
        {
            List<Segment> tbr = new List<Segment>();

            if (EndAngle > StartAngle)
            {
                if (EndAngle - StartAngle <= Math.PI / 2)
                {
                    tbr.AddRange(GetBezierSegment(Points[0].X, Points[0].Y, Radius, StartAngle, EndAngle, true));
                }
                else
                {
                    int count = (int)Math.Ceiling(2 * (EndAngle - StartAngle) / Math.PI);
                    double angle = StartAngle;

                    for (int i = 0; i < count; i++)
                    {
                        tbr.AddRange(GetBezierSegment(Points[0].X, Points[0].Y, Radius, angle, angle + (EndAngle - StartAngle) / count, i == 0));
                        angle += (EndAngle - StartAngle) / count;
                    }
                }
            }
            else if (EndAngle < StartAngle)
            {
                Point startPoint = new Point(Points[0].X + Radius * Math.Cos(EndAngle), Points[0].Y + Radius * Math.Sin(EndAngle));
                if (StartAngle - EndAngle <= Math.PI / 2)
                {
                    tbr.AddRange(GetBezierSegment(Points[0].X, Points[0].Y, Radius, EndAngle, StartAngle, true));
                }
                else
                {
                    int count = (int)Math.Ceiling(2 * (StartAngle - EndAngle) / Math.PI);
                    double angle = EndAngle;

                    for (int i = 0; i < count; i++)
                    {
                        tbr.AddRange(GetBezierSegment(Points[0].X, Points[0].Y, Radius, angle, angle + (StartAngle - EndAngle) / count, i == 0));
                        angle += (StartAngle - EndAngle) / count;
                    }
                }

                return ReverseSegments(tbr, startPoint).ToArray();
            }

            return tbr.ToArray();
        }

        private static Segment[] ReverseSegments(IReadOnlyList<Segment> originalSegments, Point startPoint)
        {
            List<Segment> tbr = new List<Segment>(originalSegments.Count);

            for (int i = originalSegments.Count - 1; i >= 0; i--)
            {
                switch (originalSegments[i].Type)
                {
                    case SegmentType.Line:
                        if (i > 0)
                        {
                            tbr.Add(new LineSegment(originalSegments[i - 1].Point));
                        }
                        else
```

```
                        {
                            tbr.Add(new LineSegment(startPoint));
                        }
                        break;
                    case SegmentType.CubicBezier:
                        CubicBezierSegment originalSegment = (CubicBezierSegment)originalSegments[i];
                        if (i > 0)
                        {
                            tbr.Add(new CubicBezierSegment(originalSegment.Points[1], originalSegment.Points[0], originalSegments[i - 1].Point));
                        }
                        else
                        {
                            tbr.Add(new CubicBezierSegment(originalSegment.Points[1], originalSegment.Points[0], startPoint));
                        }
                        break;
                }
            }

            return tbr.ToArray();
        }

        const double k = 0.55191496;

        private static Segment[] GetBezierSegment(double cX, double cY, double radius, double startAngle, double endAngle, bool firstArc)
        {
            double phi = Math.PI / 4;

            double x1 = radius * Math.Cos(phi);
            double y1 = radius * Math.Sin(phi);

            double x4 = x1;
            double y4 = -y1;

            double x3 = x1 + k * radius * Math.Sin(phi);
            double y3 = y1 - k * radius * Math.Cos(phi);

            double x2 = x4 + k * radius * Math.Sin(phi);
            double y2 = y4 + k * radius * Math.Cos(phi);

            double u = 2 * (endAngle - startAngle) / Math.PI;

            double fx2 = (1 - u) * x4 + u * x2;
            double fy2 = (1 - u) * y4 + u * y2;

            double fx3 = (1 - u) * fx2 + u * ((1 - u) * x2 + u * x3);
            double fy3 = (1 - u) * fy2 + u * ((1 - u) * y2 + u * y3);

            double rX1 = cX + radius * Math.Cos(startAngle);
            double rY1 = cY + radius * Math.Sin(startAngle);

            double rX4 = cX + radius * Math.Cos(endAngle);
            double rY4 = cY + radius * Math.Sin(endAngle);

            Point rot2 = Utils.RotatePoint(new Point(fx2, fy2), phi + startAngle);
            Point rot3 = Utils.RotatePoint(new Point(fx3, fy3), phi + startAngle);

            List<Segment> tbr = new List<Segment>();

            if (firstArc)
            {
                tbr.Add(new LineSegment(rX1, rY1));
            }

            tbr.Add(new CubicBezierSegment(cX + rot2.X, cY + rot2.Y, cX + rot3.X, cY + rot3.Y, rX4, rY4));

            return tbr.ToArray();
        }
        public double Radius { get; }
        public double StartAngle { get; }
        public double EndAngle { get; }

        public ArcSegment(Point center, double radius, double startAngle, double endAngle)
        {
            this.Points = new Point[] { center };
            this.Radius = radius;
            this.StartAngle = startAngle;
            this.EndAngle = endAngle;
        }

        public ArcSegment(double centerX, double centerY, double radius, double startAngle, double endAngle)
        {
            this.Points = new Point[] { new Point(centerX, centerY) };
```

```
            this.Radius = radius;
            this.StartAngle = startAngle;
            this.EndAngle = endAngle;
        }

        public override Segment Clone()
        {
            return new ArcSegment(Point.X, Point.Y, Radius, StartAngle, EndAngle);
        }

        public override Point Point
        {
            get
            {
                return new Point(this.Points[0].X + Math.Cos(EndAngle) * Radius, this.Points[0].Y + Math.Sin(EndAngle) * Radius);
            }
        }


        private double cachedLength = double.NaN;

        public override double Measure(Point previousPoint)
        {
            if (double.IsNaN(cachedLength))
            {
                Point arcStartPoint = new Point(this.Points[0].X + Math.Cos(StartAngle) * Radius, this.Points[0].Y + Math.Sin(StartAngle) * Radius);

                cachedLength = Radius * Math.Abs(EndAngle - StartAngle) + Math.Sqrt((arcStartPoint.X - previousPoint.X) * (arcStartPoint.X - previousPoint.X) + (arcStartPoint.Y - previousPoint.Y) * (arcStartPoint.Y - previousPoint.Y));
            }

            return cachedLength;
        }

        public override Point GetPointAt(Point previousPoint, double position)
        {
            double totalLength = this.Measure(previousPoint);
            double arcLength = Radius * Math.Abs(EndAngle - StartAngle);

            double preArc = (totalLength - arcLength) / totalLength;

            if (position < preArc)
            {
                if (position >= 0)
                {
                    double relPos = position / preArc;
                    Point arcStartPoint = new Point(this.Points[0].X + Math.Cos(StartAngle) * Radius, this.Points[0].Y + Math.Sin(StartAngle) * Radius);

                    return new Point(previousPoint.X * (1 - relPos) + arcStartPoint.X * relPos, previousPoint.Y * (1 - relPos) + arcStartPoint.Y * relPos);
                }
                else
                {
                    Point arcStartPoint = new Point(this.Points[0].X + Math.Cos(StartAngle) * Radius, this.Points[0].Y + Math.Sin(StartAngle) * Radius);
                    Point tangent = GetTangentAt(previousPoint, 0);
                    double excessLength = position * this.Measure(previousPoint);
                    return new Point(arcStartPoint.X + tangent.X * excessLength, arcStartPoint.Y + tangent.Y * excessLength);
                }
            }
            else
            {
                double relPos = position - preArc / (1 - preArc);

                if (relPos <= 1)
                {
                    double angle = StartAngle * (1 - relPos) + EndAngle * relPos;
                    return new Point(this.Points[0].X + Radius * Math.Cos(angle), this.Points[0].Y + Radius * Math.Sin(angle));
                }
                else
                {
                    Point arcEndPoint = this.Point;
                    Point tangent = GetTangentAt(previousPoint, 1);
                    double excessLength = (position - 1) * this.Measure(previousPoint);
                    return new Point(arcEndPoint.X + tangent.X * excessLength, arcEndPoint.Y + tangent.Y * excessLength);
                }
            }
        }

        public override Point GetTangentAt(Point previousPoint, double position)
```

```
        {
            double totalLength = this.Measure(previousPoint);
            double arcLength = Radius * Math.Abs(EndAngle - StartAngle);

            double preArc = (totalLength - arcLength) / totalLength;

            if (position < preArc)
            {
                Point arcStartPoint = new Point(this.Points[0].X + Math.Cos(StartAngle) * Radius, this.Points[0].Y + Math.Sin(StartAngle) * Radius);
                Point tang = new Point((arcStartPoint.X - previousPoint.X) * Math.Sign(EndAngle - StartAngle), (arcStartPoint.Y - previousPoint.Y) * Math.Sign(EndAngle - StartAngle)).Normalize();

                if (tang.Modulus() > 0.001)
                {
                    return tang.Normalize();
                }
                else
                {
                    return this.GetTangentAt(previousPoint, 0);
                }
            }
            else
            {
                double relPos = position - preArc / (1 - preArc);

                if (relPos <= 1)
                {
                    double angle = StartAngle * (1 - relPos) + EndAngle * relPos;
                    return new Point(-Math.Sin(angle) * Math.Sign(EndAngle - StartAngle), Math.Cos(angle) * Math.Sign(EndAngle - StartAngle));
                }
                else
                {
                    return new Point(-Math.Sin(EndAngle) * Math.Sign(EndAngle - StartAngle), Math.Cos(EndAngle) * Math.Sign(EndAngle - StartAngle));
                }
            }

        }

        public override IEnumerable<Segment> Linearise(Point? previousPoint, double resolution)
        {
            double length = this.Measure(previousPoint.Value);
            int segmentCount = (int)Math.Ceiling(length / resolution);

            for (int i = 0; i < segmentCount; i++)
            {
                yield return new LineSegment(this.GetPointAt(previousPoint.Value, (double)(i + 1) / segmentCount));
            }
        }
        public override IEnumerable<Point> GetLinearisationTangents(Point? previousPoint, double resolution)
        {
            double length = this.Measure(previousPoint.Value);
            int segmentCount = (int)Math.Ceiling(length / resolution);

            for (int i = 0; i < segmentCount; i++)
            {
                yield return this.GetTangentAt(previousPoint.Value, (double)(i + 1) / segmentCount);
            }
        }

        public override IEnumerable<Segment> Transform(Func<Point, Point> transformationFunction)
        {
            foreach (Segment seg in this.ToBezierSegments())
            {
                foreach (Segment seg2 in seg.Transform(transformationFunction))
                {
                    yield return seg2;
                }
            }
        }
    }


}
```

8.52 SmoothSpline.cs

```
/*
    VectSharp - A light library for C# vector graphics.
```

```
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace VectSharp
{
    //derived from https://www.particleincell.com/wp-content/uploads/2012/06/bezier-spline.js

    internal static class SmoothSpline
    {
        public static Point[] SmoothSplines(Point[] points)
        {
            double[] x = (from el in points select el.X).ToArray();
            double[] y = (from el in points select el.Y).ToArray();

            (double[] p1, double[] p2) px = ComputeControlPoints(x);
            (double[] p1, double[] p2) py = ComputeControlPoints(y);

            List<Point> tbr = new List<Point>();

            for (int i = 0; i < points.Length - 1; i++)
            {
                tbr.Add(new Point(x[i], y[i]));
                tbr.Add(new Point(px.p1[i], py.p1[i]));
                tbr.Add(new Point(px.p2[i], py.p2[i]));
            }

            tbr.Add(new Point(x[x.Length - 1], y[x.Length - 1]));

            return tbr.ToArray();
        }

        /*computes control points given knots K, this is the brain of the operation*/
        private static (double[] p1, double[] p2) ComputeControlPoints(double[] K)
        {
            int n = K.Length - 1;


            double[] p1 = new double[n];
            double[] p2 = new double[n];


            /*rhs vector*/
            double[] a = new double[n];
            double[] b = new double[n];
            double[] c = new double[n];
            double[] r = new double[n];

            /*left most segment*/
            a[0] = 0;
            b[0] = 2;
            c[0] = 1;
            r[0] = K[0] + 2 * K[1];

            /*internal segments*/
            for (int i = 1; i < n - 1; i++)
            {
                a[i] = 1;
                b[i] = 4;
                c[i] = 1;
                r[i] = 4 * K[i] + 2 * K[i + 1];
            }

            /*right segment*/
            a[n - 1] = 2;
            b[n - 1] = 7;
            c[n - 1] = 0;
            r[n - 1] = 8 * K[n - 1] + K[n];

            /*solves Ax=b with the Thomas algorithm (from Wikipedia)*/
            for (int i = 1; i < n; i++)
```

```
            {
                double m = a[i] / b[i - 1];
                b[i] = b[i] - m * c[i - 1];
                r[i] = r[i] - m * r[i - 1];
            }

            p1[n - 1] = r[n - 1] / b[n - 1];
            for (int i = n - 2; i >= 0; i--)
            {
                p1[i] = (r[i] - c[i] * p1[i + 1]) / b[i];
            }

            /*we have p1, now compute p2*/
            for (int i = 0; i < n - 1; i++)
            {
                p2[i] = 2 * K[i + 1] - p1[i + 1];
            }

            p2[n - 1] = 0.5 * (K[n] + p1[n - 1]);

            return (p1, p2);
        }

    }
}
```

8.53 StandardColours.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.Text;

namespace VectSharp
{
    public partial struct Colour
    {
        private static Dictionary<string, Colour> StandardColours = new Dictionary<string, Colour>(StringComparer.OrdinalIgnoreCase)
        {
            { "Black", Colour.FromRgb(0, 0, 0) },
            { "Navy", Colour.FromRgb(0, 0, 128) },
            { "DarkBlue", Colour.FromRgb(0, 0, 139) },
            { "MediumBlue", Colour.FromRgb(0, 0, 205) },
            { "Blue", Colour.FromRgb(0, 0, 255) },
            { "DarkGreen", Colour.FromRgb(0, 100, 0) },
            { "Green", Colour.FromRgb(0, 128, 0) },
            { "Teal", Colour.FromRgb(0, 128, 128) },
            { "DarkCyan", Colour.FromRgb(0, 139, 139) },
            { "DeepSkyBlue", Colour.FromRgb(0, 191, 255) },
            { "DarkTurquoise", Colour.FromRgb(0, 206, 209) },
            { "MediumSpringGreen", Colour.FromRgb(0, 250, 154) },
            { "Lime", Colour.FromRgb(0, 255, 0) },
            { "SpringGreen", Colour.FromRgb(0, 255, 127) },
            { "Aqua", Colour.FromRgb(0, 255, 255) },
            { "Cyan", Colour.FromRgb(0, 255, 255) },
            { "MidnightBlue", Colour.FromRgb(25, 25, 112) },
            { "DodgerBlue", Colour.FromRgb(30, 144, 255) },
            { "LightSeaGreen", Colour.FromRgb(32, 178, 170) },
            { "ForestGreen", Colour.FromRgb(34, 139, 34) },
            { "SeaGreen", Colour.FromRgb(46, 139, 87) },
            { "DarkSlateGray", Colour.FromRgb(47, 79, 79) },
            { "DarkSlateGrey", Colour.FromRgb(47, 79, 79) },
            { "LimeGreen", Colour.FromRgb(50, 205, 50) },
            { "MediumSeaGreen", Colour.FromRgb(60, 179, 113) },
            { "Turquoise", Colour.FromRgb(64, 224, 208) },
            { "RoyalBlue", Colour.FromRgb(65, 105, 225) },
```

```
            { "SteelBlue", Colour.FromRgb(70, 130, 180) },
            { "DarkSlateBlue", Colour.FromRgb(72, 61, 139) },
            { "MediumTurquoise", Colour.FromRgb(72, 209, 204) },
            { "Indigo", Colour.FromRgb(75, 0, 130) },
            { "DarkOliveGreen", Colour.FromRgb(85, 107, 47) },
            { "CadetBlue", Colour.FromRgb(95, 158, 160) },
            { "CornflowerBlue", Colour.FromRgb(100, 149, 237) },
            { "RebeccaPurple", Colour.FromRgb(102, 51, 153) },
            { "MediumAquaMarine", Colour.FromRgb(102, 205, 170) },
            { "DimGray", Colour.FromRgb(105, 105, 105) },
            { "DimGrey", Colour.FromRgb(105, 105, 105) },
            { "SlateBlue", Colour.FromRgb(106, 90, 205) },
            { "OliveDrab", Colour.FromRgb(107, 142, 35) },
            { "SlateGray", Colour.FromRgb(112, 128, 144) },
            { "SlateGrey", Colour.FromRgb(112, 128, 144) },
            { "LightSlateGray", Colour.FromRgb(119, 136, 153) },
            { "LightSlateGrey", Colour.FromRgb(119, 136, 153) },
            { "MediumSlateBlue", Colour.FromRgb(123, 104, 238) },
            { "LawnGreen", Colour.FromRgb(124, 252, 0) },
            { "Chartreuse", Colour.FromRgb(127, 255, 0) },
            { "Aquamarine", Colour.FromRgb(127, 255, 212) },
            { "Maroon", Colour.FromRgb(128, 0, 0) },
            { "Purple", Colour.FromRgb(128, 0, 128) },
            { "Olive", Colour.FromRgb(128, 128, 0) },
            { "Gray", Colour.FromRgb(128, 128, 128) },
            { "Grey", Colour.FromRgb(128, 128, 128) },
            { "SkyBlue", Colour.FromRgb(135, 206, 235) },
            { "LightSkyBlue", Colour.FromRgb(135, 206, 250) },
            { "BlueViolet", Colour.FromRgb(138, 43, 226) },
            { "DarkRed", Colour.FromRgb(139, 0, 0) },
            { "DarkMagenta", Colour.FromRgb(139, 0, 139) },
            { "SaddleBrown", Colour.FromRgb(139, 69, 19) },
            { "DarkSeaGreen", Colour.FromRgb(143, 188, 143) },
            { "LightGreen", Colour.FromRgb(144, 238, 144) },
            { "MediumPurple", Colour.FromRgb(147, 112, 219) },
            { "DarkViolet", Colour.FromRgb(148, 0, 211) },
            { "PaleGreen", Colour.FromRgb(152, 251, 152) },
            { "DarkOrchid", Colour.FromRgb(153, 50, 204) },
            { "YellowGreen", Colour.FromRgb(154, 205, 50) },
            { "Sienna", Colour.FromRgb(160, 82, 45) },
            { "Brown", Colour.FromRgb(165, 42, 42) },
            { "DarkGray", Colour.FromRgb(169, 169, 169) },
            { "DarkGrey", Colour.FromRgb(169, 169, 169) },
            { "LightBlue", Colour.FromRgb(173, 216, 230) },
            { "GreenYellow", Colour.FromRgb(173, 255, 47) },
            { "PaleTurquoise", Colour.FromRgb(175, 238, 238) },
            { "LightSteelBlue", Colour.FromRgb(176, 196, 222) },
            { "PowderBlue", Colour.FromRgb(176, 224, 230) },
            { "FireBrick", Colour.FromRgb(178, 34, 34) },
            { "DarkGoldenRod", Colour.FromRgb(184, 134, 11) },
            { "MediumOrchid", Colour.FromRgb(186, 85, 211) },
            { "RosyBrown", Colour.FromRgb(188, 143, 143) },
            { "DarkKhaki", Colour.FromRgb(189, 183, 107) },
            { "Silver", Colour.FromRgb(192, 192, 192) },
            { "MediumVioletRed", Colour.FromRgb(199, 21, 133) },
            { "IndianRed", Colour.FromRgb(205, 92, 92) },
            { "Peru", Colour.FromRgb(205, 133, 63) },
            { "Chocolate", Colour.FromRgb(210, 105, 30) },
            { "Tan", Colour.FromRgb(210, 180, 140) },
            { "LightGray", Colour.FromRgb(211, 211, 211) },
            { "LightGrey", Colour.FromRgb(211, 211, 211) },
            { "Thistle", Colour.FromRgb(216, 191, 216) },
            { "Orchid", Colour.FromRgb(218, 112, 214) },
            { "GoldenRod", Colour.FromRgb(218, 165, 32) },
            { "PaleVioletRed", Colour.FromRgb(219, 112, 147) },
            { "Crimson", Colour.FromRgb(220, 20, 60) },
            { "Gainsboro", Colour.FromRgb(220, 220, 220) },
            { "Plum", Colour.FromRgb(221, 160, 221) },
            { "BurlyWood", Colour.FromRgb(222, 184, 135) },
            { "LightCyan", Colour.FromRgb(224, 255, 255) },
            { "Lavender", Colour.FromRgb(230, 230, 250) },
            { "DarkSalmon", Colour.FromRgb(233, 150, 122) },
            { "Violet", Colour.FromRgb(238, 130, 238) },
            { "PaleGoldenRod", Colour.FromRgb(238, 232, 170) },
            { "LightCoral", Colour.FromRgb(240, 128, 128) },
            { "Khaki", Colour.FromRgb(240, 230, 140) },
            { "AliceBlue", Colour.FromRgb(240, 248, 255) },
            { "HoneyDew", Colour.FromRgb(240, 255, 240) },
            { "Azure", Colour.FromRgb(240, 255, 255) },
            { "SandyBrown", Colour.FromRgb(244, 164, 96) },
            { "Wheat", Colour.FromRgb(245, 222, 179) },
            { "Beige", Colour.FromRgb(245, 245, 220) },
            { "WhiteSmoke", Colour.FromRgb(245, 245, 245) },
            { "MintCream", Colour.FromRgb(245, 255, 250) },
            { "GhostWhite", Colour.FromRgb(248, 248, 255) },
            { "Salmon", Colour.FromRgb(250, 128, 114) },
            { "AntiqueWhite", Colour.FromRgb(250, 235, 215) },
```

```
            { "Linen", Colour.FromRgb(250, 240, 230) },
            { "LightGoldenRodYellow", Colour.FromRgb(250, 250, 210) },
            { "OldLace", Colour.FromRgb(253, 245, 230) },
            { "Red", Colour.FromRgb(255, 0, 0) },
            { "Fuchsia", Colour.FromRgb(255, 0, 255) },
            { "Magenta", Colour.FromRgb(255, 0, 255) },
            { "DeepPink", Colour.FromRgb(255, 20, 147) },
            { "OrangeRed", Colour.FromRgb(255, 69, 0) },
            { "Tomato", Colour.FromRgb(255, 99, 71) },
            { "HotPink", Colour.FromRgb(255, 105, 180) },
            { "Coral", Colour.FromRgb(255, 127, 80) },
            { "DarkOrange", Colour.FromRgb(255, 140, 0) },
            { "LightSalmon", Colour.FromRgb(255, 160, 122) },
            { "Orange", Colour.FromRgb(255, 165, 0) },
            { "LightPink", Colour.FromRgb(255, 182, 193) },
            { "Pink", Colour.FromRgb(255, 192, 203) },
            { "Gold", Colour.FromRgb(255, 215, 0) },
            { "PeachPuff", Colour.FromRgb(255, 218, 185) },
            { "NavajoWhite", Colour.FromRgb(255, 222, 173) },
            { "Moccasin", Colour.FromRgb(255, 228, 181) },
            { "Bisque", Colour.FromRgb(255, 228, 196) },
            { "MistyRose", Colour.FromRgb(255, 228, 225) },
            { "BlanchedAlmond", Colour.FromRgb(255, 235, 205) },
            { "PapayaWhip", Colour.FromRgb(255, 239, 213) },
            { "LavenderBlush", Colour.FromRgb(255, 240, 245) },
            { "SeaShell", Colour.FromRgb(255, 245, 238) },
            { "Cornsilk", Colour.FromRgb(255, 248, 220) },
            { "LemonChiffon", Colour.FromRgb(255, 250, 205) },
            { "FloralWhite", Colour.FromRgb(255, 250, 240) },
            { "Snow", Colour.FromRgb(255, 250, 250) },
            { "Yellow", Colour.FromRgb(255, 255, 0) },
            { "LightYellow", Colour.FromRgb(255, 255, 224) },
            { "Ivory", Colour.FromRgb(255, 255, 240) },
            { "White", Colour.FromRgb(255, 255, 255) },
        };
    }

    /// <summary>
    /// Standard colours.
    /// </summary>
    public static class Colours
    {
        /// <summary>
        /// Black #000000
        /// </summary>
        public static Colour Black = Colour.FromRgb(0, 0, 0);
        /// <summary>
        /// Navy #000080
        /// </summary>
        public static Colour Navy = Colour.FromRgb(0, 0, 128);
        /// <summary>
        /// DarkBlue #00008B
        /// </summary>
        public static Colour DarkBlue = Colour.FromRgb(0, 0, 139);
        /// <summary>
        /// MediumBlue #0000CD
        /// </summary>
        public static Colour MediumBlue = Colour.FromRgb(0, 0, 205);
        /// <summary>
        /// Blue #0000FF
        /// </summary>
        public static Colour Blue = Colour.FromRgb(0, 0, 255);
        /// <summary>
        /// DarkGreen #006400
        /// </summary>
        public static Colour DarkGreen = Colour.FromRgb(0, 100, 0);
        /// <summary>
        /// Green #008000
        /// </summary>
        public static Colour Green = Colour.FromRgb(0, 128, 0);
        /// <summary>
        /// Teal #008080
        /// </summary>
        public static Colour Teal = Colour.FromRgb(0, 128, 128);
        /// <summary>
        /// DarkCyan #008B8B
        /// </summary>
        public static Colour DarkCyan = Colour.FromRgb(0, 139, 139);
        /// <summary>
        /// DeepSkyBlue #00BFFF
        /// </summary>
        public static Colour DeepSkyBlue = Colour.FromRgb(0, 191, 255);
        /// <summary>
        /// DarkTurquoise #00CED1
        /// </summary>
        public static Colour DarkTurquoise = Colour.FromRgb(0, 206, 209);
        /// <summary>
```

```
        /// MediumSpringGreen #00FA9A
        /// </summary>
        public static Colour MediumSpringGreen = Colour.FromRgb(0, 250, 154);
        /// <summary>
        /// Lime #00FF00
        /// </summary>
        public static Colour Lime = Colour.FromRgb(0, 255, 0);
        /// <summary>
        /// SpringGreen #00FF7F
        /// </summary>
        public static Colour SpringGreen = Colour.FromRgb(0, 255, 127);
        /// <summary>
        /// Aqua #00FFFF
        /// </summary>
        public static Colour Aqua = Colour.FromRgb(0, 255, 255);
        /// <summary>
        /// Cyan #00FFFF
        /// </summary>
        public static Colour Cyan = Colour.FromRgb(0, 255, 255);
        /// <summary>
        /// MidnightBlue #191970
        /// </summary>
        public static Colour MidnightBlue = Colour.FromRgb(25, 25, 112);
        /// <summary>
        /// DodgerBlue #1E90FF
        /// </summary>
        public static Colour DodgerBlue = Colour.FromRgb(30, 144, 255);
        /// <summary>
        /// LightSeaGreen #20B2AA
        /// </summary>
        public static Colour LightSeaGreen = Colour.FromRgb(32, 178, 170);
        /// <summary>
        /// ForestGreen #228B22
        /// </summary>
        public static Colour ForestGreen = Colour.FromRgb(34, 139, 34);
        /// <summary>
        /// SeaGreen #2E8B57
        /// </summary>
        public static Colour SeaGreen = Colour.FromRgb(46, 139, 87);
        /// <summary>
        /// DarkSlateGray #2F4F4F
        /// </summary>
        public static Colour DarkSlateGray = Colour.FromRgb(47, 79, 79);
        /// <summary>
        /// DarkSlateGrey #2F4F4F
        /// </summary>
        public static Colour DarkSlateGrey = Colour.FromRgb(47, 79, 79);
        /// <summary>
        /// LimeGreen #32CD32
        /// </summary>
        public static Colour LimeGreen = Colour.FromRgb(50, 205, 50);
        /// <summary>
        /// MediumSeaGreen #3CB371
        /// </summary>
        public static Colour MediumSeaGreen = Colour.FromRgb(60, 179, 113);
        /// <summary>
        /// Turquoise #40E0D0
        /// </summary>
        public static Colour Turquoise = Colour.FromRgb(64, 224, 208);
        /// <summary>
        /// RoyalBlue #4169E1
        /// </summary>
        public static Colour RoyalBlue = Colour.FromRgb(65, 105, 225);
        /// <summary>
        /// SteelBlue #4682B4
        /// </summary>
        public static Colour SteelBlue = Colour.FromRgb(70, 130, 180);
        /// <summary>
        /// DarkSlateBlue #483D8B
        /// </summary>
        public static Colour DarkSlateBlue = Colour.FromRgb(72, 61, 139);
        /// <summary>
        /// MediumTurquoise #48D1CC
        /// </summary>
        public static Colour MediumTurquoise = Colour.FromRgb(72, 209, 204);
        /// <summary>
        /// Indigo #4B0082
        /// </summary>
        public static Colour Indigo = Colour.FromRgb(75, 0, 130);
        /// <summary>
        /// DarkOliveGreen #556B2F
        /// </summary>
        public static Colour DarkOliveGreen = Colour.FromRgb(85, 107, 47);
        /// <summary>
        /// CadetBlue #5F9EA0
        /// </summary>
        public static Colour CadetBlue = Colour.FromRgb(95, 158, 160);
```

```
        /// <summary>
        /// CornflowerBlue #6495ED
        /// </summary>
        public static Colour CornflowerBlue = Colour.FromRgb(100, 149, 237);
        /// <summary>
        /// RebeccaPurple #663399
        /// </summary>
        public static Colour RebeccaPurple = Colour.FromRgb(102, 51, 153);
        /// <summary>
        /// MediumAquaMarine #66CDAA
        /// </summary>
        public static Colour MediumAquaMarine = Colour.FromRgb(102, 205, 170);
        /// <summary>
        /// DimGray #696969
        /// </summary>
        public static Colour DimGray = Colour.FromRgb(105, 105, 105);
        /// <summary>
        /// DimGrey #696969
        /// </summary>
        public static Colour DimGrey = Colour.FromRgb(105, 105, 105);
        /// <summary>
        /// SlateBlue #6A5ACD
        /// </summary>
        public static Colour SlateBlue = Colour.FromRgb(106, 90, 205);
        /// <summary>
        /// OliveDrab #6B8E23
        /// </summary>
        public static Colour OliveDrab = Colour.FromRgb(107, 142, 35);
        /// <summary>
        /// SlateGray #708090
        /// </summary>
        public static Colour SlateGray = Colour.FromRgb(112, 128, 144);
        /// <summary>
        /// SlateGrey #708090
        /// </summary>
        public static Colour SlateGrey = Colour.FromRgb(112, 128, 144);
        /// <summary>
        /// LightSlateGray #778899
        /// </summary>
        public static Colour LightSlateGray = Colour.FromRgb(119, 136, 153);
        /// <summary>
        /// LightSlateGrey #778899
        /// </summary>
        public static Colour LightSlateGrey = Colour.FromRgb(119, 136, 153);
        /// <summary>
        /// MediumSlateBlue #7B68EE
        /// </summary>
        public static Colour MediumSlateBlue = Colour.FromRgb(123, 104, 238);
        /// <summary>
        /// LawnGreen #7CFC00
        /// </summary>
        public static Colour LawnGreen = Colour.FromRgb(124, 252, 0);
        /// <summary>
        /// Chartreuse #7FFF00
        /// </summary>
        public static Colour Chartreuse = Colour.FromRgb(127, 255, 0);
        /// <summary>
        /// Aquamarine #7FFFD4
        /// </summary>
        public static Colour Aquamarine = Colour.FromRgb(127, 255, 212);
        /// <summary>
        /// Maroon #800000
        /// </summary>
        public static Colour Maroon = Colour.FromRgb(128, 0, 0);
        /// <summary>
        /// Purple #800080
        /// </summary>
        public static Colour Purple = Colour.FromRgb(128, 0, 128);
        /// <summary>
        /// Olive #808000
        /// </summary>
        public static Colour Olive = Colour.FromRgb(128, 128, 0);
        /// <summary>
        /// Gray #808080
        /// </summary>
        public static Colour Gray = Colour.FromRgb(128, 128, 128);
        /// <summary>
        /// Grey #808080
        /// </summary>
        public static Colour Grey = Colour.FromRgb(128, 128, 128);
        /// <summary>
        /// SkyBlue #87CEEB
        /// </summary>
        public static Colour SkyBlue = Colour.FromRgb(135, 206, 235);
        /// <summary>
        /// LightSkyBlue #87CEFA
        /// </summary>
```

```
        public static Colour LightSkyBlue = Colour.FromRgb(135, 206, 250);
        /// <summary>
        /// BlueViolet #8A2BE2
        /// </summary>
        public static Colour BlueViolet = Colour.FromRgb(138, 43, 226);
        /// <summary>
        /// DarkRed #8B0000
        /// </summary>
        public static Colour DarkRed = Colour.FromRgb(139, 0, 0);
        /// <summary>
        /// DarkMagenta #8B008B
        /// </summary>
        public static Colour DarkMagenta = Colour.FromRgb(139, 0, 139);
        /// <summary>
        /// SaddleBrown #8B4513
        /// </summary>
        public static Colour SaddleBrown = Colour.FromRgb(139, 69, 19);
        /// <summary>
        /// DarkSeaGreen #8FBC8F
        /// </summary>
        public static Colour DarkSeaGreen = Colour.FromRgb(143, 188, 143);
        /// <summary>
        /// LightGreen #90EE90
        /// </summary>
        public static Colour LightGreen = Colour.FromRgb(144, 238, 144);
        /// <summary>
        /// MediumPurple #9370DB
        /// </summary>
        public static Colour MediumPurple = Colour.FromRgb(147, 112, 219);
        /// <summary>
        /// DarkViolet #9400D3
        /// </summary>
        public static Colour DarkViolet = Colour.FromRgb(148, 0, 211);
        /// <summary>
        /// PaleGreen #98FB98
        /// </summary>
        public static Colour PaleGreen = Colour.FromRgb(152, 251, 152);
        /// <summary>
        /// DarkOrchid #9932CC
        /// </summary>
        public static Colour DarkOrchid = Colour.FromRgb(153, 50, 204);
        /// <summary>
        /// YellowGreen #9ACD32
        /// </summary>
        public static Colour YellowGreen = Colour.FromRgb(154, 205, 50);
        /// <summary>
        /// Sienna #A0522D
        /// </summary>
        public static Colour Sienna = Colour.FromRgb(160, 82, 45);
        /// <summary>
        /// Brown #A52A2A
        /// </summary>
        public static Colour Brown = Colour.FromRgb(165, 42, 42);
        /// <summary>
        /// DarkGray #A9A9A9
        /// </summary>
        public static Colour DarkGray = Colour.FromRgb(169, 169, 169);
        /// <summary>
        /// DarkGrey #A9A9A9
        /// </summary>
        public static Colour DarkGrey = Colour.FromRgb(169, 169, 169);
        /// <summary>
        /// LightBlue #ADD8E6
        /// </summary>
        public static Colour LightBlue = Colour.FromRgb(173, 216, 230);
        /// <summary>
        /// GreenYellow #ADFF2F
        /// </summary>
        public static Colour GreenYellow = Colour.FromRgb(173, 255, 47);
        /// <summary>
        /// PaleTurquoise #AFEEEE
        /// </summary>
        public static Colour PaleTurquoise = Colour.FromRgb(175, 238, 238);
        /// <summary>
        /// LightSteelBlue #B0C4DE
        /// </summary>
        public static Colour LightSteelBlue = Colour.FromRgb(176, 196, 222);
        /// <summary>
        /// PowderBlue #B0E0E6
        /// </summary>
        public static Colour PowderBlue = Colour.FromRgb(176, 224, 230);
        /// <summary>
        /// FireBrick #B22222
        /// </summary>
        public static Colour FireBrick = Colour.FromRgb(178, 34, 34);
        /// <summary>
        /// DarkGoldenRod #B8860B
```

```
        /// </summary>
        public static Colour DarkGoldenRod = Colour.FromRgb(184, 134, 11);
        /// <summary>
        /// MediumOrchid #BA55D3
        /// </summary>
        public static Colour MediumOrchid = Colour.FromRgb(186, 85, 211);
        /// <summary>
        /// RosyBrown #BC8F8F
        /// </summary>
        public static Colour RosyBrown = Colour.FromRgb(188, 143, 143);
        /// <summary>
        /// DarkKhaki #BDB76B
        /// </summary>
        public static Colour DarkKhaki = Colour.FromRgb(189, 183, 107);
        /// <summary>
        /// Silver #C0C0C0
        /// </summary>
        public static Colour Silver = Colour.FromRgb(192, 192, 192);
        /// <summary>
        /// MediumVioletRed #C71585
        /// </summary>
        public static Colour MediumVioletRed = Colour.FromRgb(199, 21, 133);
        /// <summary>
        /// IndianRed #CD5C5C
        /// </summary>
        public static Colour IndianRed = Colour.FromRgb(205, 92, 92);
        /// <summary>
        /// Peru #CD853F
        /// </summary>
        public static Colour Peru = Colour.FromRgb(205, 133, 63);
        /// <summary>
        /// Chocolate #D2691E
        /// </summary>
        public static Colour Chocolate = Colour.FromRgb(210, 105, 30);
        /// <summary>
        /// Tan #D2B48C
        /// </summary>
        public static Colour Tan = Colour.FromRgb(210, 180, 140);
        /// <summary>
        /// LightGray #D3D3D3
        /// </summary>
        public static Colour LightGray = Colour.FromRgb(211, 211, 211);
        /// <summary>
        /// LightGrey #D3D3D3
        /// </summary>
        public static Colour LightGrey = Colour.FromRgb(211, 211, 211);
        /// <summary>
        /// Thistle #D8BFD8
        /// </summary>
        public static Colour Thistle = Colour.FromRgb(216, 191, 216);
        /// <summary>
        /// Orchid #DA70D6
        /// </summary>
        public static Colour Orchid = Colour.FromRgb(218, 112, 214);
        /// <summary>
        /// GoldenRod #DAA520
        /// </summary>
        public static Colour GoldenRod = Colour.FromRgb(218, 165, 32);
        /// <summary>
        /// PaleVioletRed #DB7093
        /// </summary>
        public static Colour PaleVioletRed = Colour.FromRgb(219, 112, 147);
        /// <summary>
        /// Crimson #DC143C
        /// </summary>
        public static Colour Crimson = Colour.FromRgb(220, 20, 60);
        /// <summary>
        /// Gainsboro #DCDCDC
        /// </summary>
        public static Colour Gainsboro = Colour.FromRgb(220, 220, 220);
        /// <summary>
        /// Plum #DDA0DD
        /// </summary>
        public static Colour Plum = Colour.FromRgb(221, 160, 221);
        /// <summary>
        /// BurlyWood #DEB887
        /// </summary>
        public static Colour BurlyWood = Colour.FromRgb(222, 184, 135);
        /// <summary>
        /// LightCyan #E0FFFF
        /// </summary>
        public static Colour LightCyan = Colour.FromRgb(224, 255, 255);
        /// <summary>
        /// Lavender #E6E6FA
        /// </summary>
        public static Colour Lavender = Colour.FromRgb(230, 230, 250);
        /// <summary>
```

```
        /// DarkSalmon #E9967A
        /// </summary>
        public static Colour DarkSalmon = Colour.FromRgb(233, 150, 122);
        /// <summary>
        /// Violet #EE82EE
        /// </summary>
        public static Colour Violet = Colour.FromRgb(238, 130, 238);
        /// <summary>
        /// PaleGoldenRod #EEE8AA
        /// </summary>
        public static Colour PaleGoldenRod = Colour.FromRgb(238, 232, 170);
        /// <summary>
        /// LightCoral #F08080
        /// </summary>
        public static Colour LightCoral = Colour.FromRgb(240, 128, 128);
        /// <summary>
        /// Khaki #F0E68C
        /// </summary>
        public static Colour Khaki = Colour.FromRgb(240, 230, 140);
        /// <summary>
        /// AliceBlue #F0F8FF
        /// </summary>
        public static Colour AliceBlue = Colour.FromRgb(240, 248, 255);
        /// <summary>
        /// HoneyDew #F0FFF0
        /// </summary>
        public static Colour HoneyDew = Colour.FromRgb(240, 255, 240);
        /// <summary>
        /// Azure #F0FFFF
        /// </summary>
        public static Colour Azure = Colour.FromRgb(240, 255, 255);
        /// <summary>
        /// SandyBrown #F4A460
        /// </summary>
        public static Colour SandyBrown = Colour.FromRgb(244, 164, 96);
        /// <summary>
        /// Wheat #F5DEB3
        /// </summary>
        public static Colour Wheat = Colour.FromRgb(245, 222, 179);
        /// <summary>
        /// Beige #F5F5DC
        /// </summary>
        public static Colour Beige = Colour.FromRgb(245, 245, 220);
        /// <summary>
        /// WhiteSmoke #F5F5F5
        /// </summary>
        public static Colour WhiteSmoke = Colour.FromRgb(245, 245, 245);
        /// <summary>
        /// MintCream #F5FFFA
        /// </summary>
        public static Colour MintCream = Colour.FromRgb(245, 255, 250);
        /// <summary>
        /// GhostWhite #F8F8FF
        /// </summary>
        public static Colour GhostWhite = Colour.FromRgb(248, 248, 255);
        /// <summary>
        /// Salmon #FA8072
        /// </summary>
        public static Colour Salmon = Colour.FromRgb(250, 128, 114);
        /// <summary>
        /// AntiqueWhite #FAEBD7
        /// </summary>
        public static Colour AntiqueWhite = Colour.FromRgb(250, 235, 215);
        /// <summary>
        /// Linen #FAF0E6
        /// </summary>
        public static Colour Linen = Colour.FromRgb(250, 240, 230);
        /// <summary>
        /// LightGoldenRodYellow #FAFAD2
        /// </summary>
        public static Colour LightGoldenRodYellow = Colour.FromRgb(250, 250, 210);
        /// <summary>
        /// OldLace #FDF5E6
        /// </summary>
        public static Colour OldLace = Colour.FromRgb(253, 245, 230);
        /// <summary>
        /// Red #FF0000
        /// </summary>
        public static Colour Red = Colour.FromRgb(255, 0, 0);
        /// <summary>
        /// Fuchsia #FF00FF
        /// </summary>
        public static Colour Fuchsia = Colour.FromRgb(255, 0, 255);
        /// <summary>
        /// Magenta #FF00FF
        /// </summary>
        public static Colour Magenta = Colour.FromRgb(255, 0, 255);
```

```
        /// <summary>
        /// DeepPink #FF1493
        /// </summary>
        public static Colour DeepPink = Colour.FromRgb(255, 20, 147);
        /// <summary>
        /// OrangeRed #FF4500
        /// </summary>
        public static Colour OrangeRed = Colour.FromRgb(255, 69, 0);
        /// <summary>
        /// Tomato #FF6347
        /// </summary>
        public static Colour Tomato = Colour.FromRgb(255, 99, 71);
        /// <summary>
        /// HotPink #FF69B4
        /// </summary>
        public static Colour HotPink = Colour.FromRgb(255, 105, 180);
        /// <summary>
        /// Coral #FF7F50
        /// </summary>
        public static Colour Coral = Colour.FromRgb(255, 127, 80);
        /// <summary>
        /// DarkOrange #FF8C00
        /// </summary>
        public static Colour DarkOrange = Colour.FromRgb(255, 140, 0);
        /// <summary>
        /// LightSalmon #FFA07A
        /// </summary>
        public static Colour LightSalmon = Colour.FromRgb(255, 160, 122);
        /// <summary>
        /// Orange #FFA500
        /// </summary>
        public static Colour Orange = Colour.FromRgb(255, 165, 0);
        /// <summary>
        /// LightPink #FFB6C1
        /// </summary>
        public static Colour LightPink = Colour.FromRgb(255, 182, 193);
        /// <summary>
        /// Pink #FFC0CB
        /// </summary>
        public static Colour Pink = Colour.FromRgb(255, 192, 203);
        /// <summary>
        /// Gold #FFD700
        /// </summary>
        public static Colour Gold = Colour.FromRgb(255, 215, 0);
        /// <summary>
        /// PeachPuff #FFDAB9
        /// </summary>
        public static Colour PeachPuff = Colour.FromRgb(255, 218, 185);
        /// <summary>
        /// NavajoWhite #FFDEAD
        /// </summary>
        public static Colour NavajoWhite = Colour.FromRgb(255, 222, 173);
        /// <summary>
        /// Moccasin #FFE4B5
        /// </summary>
        public static Colour Moccasin = Colour.FromRgb(255, 228, 181);
        /// <summary>
        /// Bisque #FFE4C4
        /// </summary>
        public static Colour Bisque = Colour.FromRgb(255, 228, 196);
        /// <summary>
        /// MistyRose #FFE4E1
        /// </summary>
        public static Colour MistyRose = Colour.FromRgb(255, 228, 225);
        /// <summary>
        /// BlanchedAlmond #FFEBCD
        /// </summary>
        public static Colour BlanchedAlmond = Colour.FromRgb(255, 235, 205);
        /// <summary>
        /// PapayaWhip #FFEFD5
        /// </summary>
        public static Colour PapayaWhip = Colour.FromRgb(255, 239, 213);
        /// <summary>
        /// LavenderBlush #FFF0F5
        /// </summary>
        public static Colour LavenderBlush = Colour.FromRgb(255, 240, 245);
        /// <summary>
        /// SeaShell #FFF5EE
        /// </summary>
        public static Colour SeaShell = Colour.FromRgb(255, 245, 238);
        /// <summary>
        /// Cornsilk #FFF8DC
        /// </summary>
        public static Colour Cornsilk = Colour.FromRgb(255, 248, 220);
        /// <summary>
        /// LemonChiffon #FFFACD
        /// </summary>
```

```
        public static Colour LemonChiffon = Colour.FromRgb(255, 250, 205);
        /// <summary>
        /// FloralWhite #FFFAF0
        /// </summary>
        public static Colour FloralWhite = Colour.FromRgb(255, 250, 240);
        /// <summary>
        /// Snow #FFFAFA
        /// </summary>
        public static Colour Snow = Colour.FromRgb(255, 250, 250);
        /// <summary>
        /// Yellow #FFFF00
        /// </summary>
        public static Colour Yellow = Colour.FromRgb(255, 255, 0);
        /// <summary>
        /// LightYellow #FFFFE0
        /// </summary>
        public static Colour LightYellow = Colour.FromRgb(255, 255, 224);
        /// <summary>
        /// Ivory #FFFFF0
        /// </summary>
        public static Colour Ivory = Colour.FromRgb(255, 255, 240);
        /// <summary>
        /// White #FFFFFF
        /// </summary>
        public static Colour White = Colour.FromRgb(255, 255, 255);
    }
}
```

8.54 TrueType.cs

```
/*
    VectSharp - A light library for C# vector graphics.
    Copyright (C) 2020-2022 Giorgio Bianchini

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, version 3.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with this program. If not, see <https://www.gnu.org/licenses/>.
*/

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Text;

namespace VectSharp
{
    /// <summary>
    /// Represents a font file in TrueType format. Reference: http://stevehanov.ca/blog/?id=143, https://developer.apple.com/fonts/TrueType-Reference-Manual/, https://docs.microsoft.com/en-us/typography/opentype/spec/
    /// </summary>
    public class TrueTypeFile
    {
        private static readonly Dictionary<string, TrueTypeFile> FontCache = new Dictionary<string, TrueTypeFile>();
        private static readonly Dictionary<Stream, TrueTypeFile> StreamFontCache = new Dictionary<Stream, TrueTypeFile>();
        private static object FontCacheLock = new object();
        internal uint ScalarType { get; }
        internal ushort NumTables { get; }
        internal ushort SearchRange { get; }
        internal ushort EntrySelector { get; }
        internal ushort RangeShift { get; }
        internal Dictionary<string, TrueTypeTableOffset> TableOffsets { get; } = new Dictionary<string, TrueTypeTableOffset>();
        internal Dictionary<string, ITrueTypeTable> Tables { get; } = new Dictionary<string, ITrueTypeTable>();


        /// <summary>
        /// A stream pointing to the TrueType file source (either on disk or in memory). Never dispose this stream directly; if you really need to, call <see cref="Destroy"/> instead.
        /// </summary>
```

```
        public Stream FontStream { get; }

        /// <summary>
        /// Remove this TrueType file from the cache, clear the tables and release the <see cref="FontStream"/>.
        /// Only call this when the actual file that was used to create this object needs to be changed!
        /// </summary>
        public void Destroy()
        {
            string keyString = null;


            lock (FontCacheLock)
            {
                foreach (KeyValuePair<string, TrueTypeFile> kvp in FontCache)
                {
                    if (kvp.Value == this)
                    {
                        keyString = kvp.Key;
                        break;
                    }
                }


                if (!string.IsNullOrEmpty(keyString))
                {
                    FontCache.Remove(keyString);
                }

                Stream keyStream = null;

                foreach (KeyValuePair<Stream, TrueTypeFile> kvp in StreamFontCache)
                {
                    if (kvp.Value == this)
                    {
                        keyStream = kvp.Key;
                    }
                }

                if (keyStream != null)
                {
                    StreamFontCache.Remove(keyStream);
                }
            }

            this.FontStream.Dispose();
            this.TableOffsets.Clear();
            this.Tables.Clear();
        }

        internal TrueTypeFile(Dictionary<string, ITrueTypeTable> tables)
        {
            this.Tables = tables;

            this.ScalarType = (uint)65536;
            this.NumTables = (ushort)tables.Count;
            this.EntrySelector = (ushort)Math.Floor(Math.Log(tables.Count, 2));
            this.SearchRange = (ushort)(16 * (1 « this.EntrySelector));
            this.RangeShift = (ushort)(tables.Count * 16 - this.SearchRange);

            uint offset = 12 + 16 * (uint)tables.Count;

            uint fontChecksum = this.ScalarType;
            fontChecksum += ((uint)this.NumTables « 16) + (uint)this.SearchRange;
            fontChecksum += ((uint)this.EntrySelector « 16) + (uint)this.RangeShift;

            //For some reason, OpenType Sanitizer (https://github.com/khaledhosny/ots), integrated into Chromium and Firefox, complains if the tables are ordered differently.
            List<string> tableOrdering = new List<string>()
            {
                "OS/2", "cmap", "glyf", "head", "hhea", "hmtx", "loca", "maxp", "name", "post"
            };

            Dictionary<string, byte[]> allBytes = new Dictionary<string, byte[]>();

            foreach (KeyValuePair<string, ITrueTypeTable> kvp in from el in Tables let ind = tableOrdering.IndexOf(el.Key) orderby (ind >= 0 ? ind : tableOrdering.Count) ascending select el)
            {
                if (kvp.Key == "head")
                {
                    ((TrueTypeHeadTable)kvp.Value).ChecksumAdjustment = 0;
                }

                byte[] tableBytes = kvp.Value.GetBytes();
                allBytes.Add(kvp.Key, tableBytes);
                int length = tableBytes.Length;
```

```
                uint checksum = 0;

                for (int i = 0; i < tableBytes.Length; i += 4)
                {
                    byte b1 = tableBytes[i];
                    byte b2 = (i + 1 < tableBytes.Length) ? tableBytes[i + 1] : (byte)0;
                    byte b3 = (i + 2 < tableBytes.Length) ? tableBytes[i + 2] : (byte)0;
                    byte b4 = (i + 3 < tableBytes.Length) ? tableBytes[i + 3] : (byte)0;

                    checksum += (uint)((b1 « 24) + (b2 « 16) + (b3 « 8) + b4);
                }

                TableOffsets.Add(kvp.Key, new TrueTypeTableOffset(checksum, offset, (uint)length));

                fontChecksum += checksum;

                fontChecksum += (uint)(((byte)kvp.Key[0] « 24) + ((byte)kvp.Key[1] « 16) + ((byte)kvp.Key[2] « 8) + (byte)kvp.Key[3]);
                fontChecksum += offset;
                fontChecksum += checksum;
                fontChecksum += (uint)length;

                offset += (uint)length;

                switch (length % 4)
                {
                    case 1:
                        offset += 3;
                        break;
                    case 2:
                        offset += 2;
                        break;
                    case 3:
                        offset++;
                        break;
                }
            }

            ((TrueTypeHeadTable)this.Tables["head"]).ChecksumAdjustment = 0xB1B0AFBA - fontChecksum;


            FontStream = new MemoryStream();
            FontStream.WriteUInt(this.ScalarType);
            FontStream.WriteUShort(this.NumTables);
            FontStream.WriteUShort(this.SearchRange);
            FontStream.WriteUShort(this.EntrySelector);
            FontStream.WriteUShort(this.RangeShift);

            foreach (KeyValuePair<string, ITrueTypeTable> kvp in from el in Tables let ind = tableOrdering.IndexOf(el.Key) orderby (ind >= 0 ? ind : tableOrdering.Count) ascending select el)
            {
                FontStream.Write(Encoding.ASCII.GetBytes(kvp.Key), 0, 4);
                FontStream.WriteUInt(this.TableOffsets[kvp.Key].Checksum);
                FontStream.WriteUInt(this.TableOffsets[kvp.Key].Offset);
                FontStream.WriteUInt(this.TableOffsets[kvp.Key].Length);
            }

            foreach (KeyValuePair<string, ITrueTypeTable> kvp in from el in Tables let ind = tableOrdering.IndexOf(el.Key) orderby (ind >= 0 ? ind : tableOrdering.Count) ascending select el)
            {
                FontStream.Write(allBytes[kvp.Key], 0, allBytes[kvp.Key].Length);
                switch (allBytes[kvp.Key].Length % 4)
                {
                    case 1:
                        FontStream.WriteByte(0);
                        FontStream.WriteByte(0);
                        FontStream.WriteByte(0);
                        break;
                    case 2:
                        FontStream.WriteByte(0);
                        FontStream.WriteByte(0);
                        break;
                    case 3:
                        FontStream.WriteByte(0);
                        break;
                }
            }

            FontStream.Seek(0, SeekOrigin.Begin);

            BuildGlyphCache();
        }

        internal static TrueTypeFile CreateTrueTypeFile(string fileName)
        {
            lock (FontCacheLock)
```

```
            {
                if (!FontCache.ContainsKey(fileName))
                {
                    FontCache.Add(fileName, new TrueTypeFile(fileName));
                }

                return FontCache[fileName];
            }
        }

        internal static TrueTypeFile CreateTrueTypeFile(Stream fontStream)
        {
            lock (FontCacheLock)
            {
                if (!StreamFontCache.ContainsKey(fontStream))
                {
                    StreamFontCache.Add(fontStream, new TrueTypeFile(fontStream));
                }

                return StreamFontCache[fontStream];
            }
        }

        private TrueTypeFile(string fileName) : this(new FileStream(fileName, FileMode.Open, FileAccess.Read, FileShare.Read))
        {

        }

        private TrueTypeFile(Stream fs)
        {
            FontStream = fs;
            this.ScalarType = fs.ReadUInt();
            this.NumTables = fs.ReadUShort();
            this.SearchRange = fs.ReadUShort();
            this.EntrySelector = fs.ReadUShort();
            this.RangeShift = fs.ReadUShort();

            for (int i = 0; i < NumTables; i++)
            {
                this.TableOffsets.Add(fs.ReadString(4), new TrueTypeTableOffset(fs.ReadUInt(), fs.ReadUInt(), fs.ReadUInt()));
            }

            fs.Seek(this.TableOffsets["head"].Offset, SeekOrigin.Begin);

            Tables.Add("head", new TrueTypeHeadTable()
            {
                Version = fs.ReadFixed(),
                FontRevision = fs.ReadFixed(),
                ChecksumAdjustment = fs.ReadUInt(),
                MagicNumber = fs.ReadUInt(),
                Flags = fs.ReadUShort(),
                UnitsPerEm = fs.ReadUShort(),
                Created = fs.ReadDate(),
                Modified = fs.ReadDate(),
                XMin = fs.ReadShort(),
                YMin = fs.ReadShort(),
                XMax = fs.ReadShort(),
                YMax = fs.ReadShort(),
                MacStyle = fs.ReadUShort(),
                LowestRecPPEM = fs.ReadUShort(),
                FontDirectionInt = fs.ReadShort(),
                IndexToLocFormat = fs.ReadShort(),
                GlyphDataFormat = fs.ReadShort()
            });

            fs.Seek(this.TableOffsets["hhea"].Offset, SeekOrigin.Begin);

            Tables.Add("hhea", new TrueTypeHHeaTable()
            {
                Version = fs.ReadFixed(),
                Ascent = fs.ReadShort(),
                Descent = fs.ReadShort(),
                LineGap = fs.ReadShort(),
                AdvanceWidthMax = fs.ReadUShort(),
                MinLeftSideBearing = fs.ReadShort(),
                MinRightSideBearing = fs.ReadShort(),
                XMaxExtent = fs.ReadShort(),
                CaretSlopeRise = fs.ReadShort(),
                CaretSlopeRun = fs.ReadShort(),
                CaretOffset = fs.ReadShort()
            });

            fs.ReadShort();
            fs.ReadShort();
            fs.ReadShort();
```

```
            fs.ReadShort();

            ((TrueTypeHHeaTable)Tables["hhea"]).MetricDataFormat = fs.ReadShort();
            ((TrueTypeHHeaTable)Tables["hhea"]).NumOfLongHorMetrics = fs.ReadUShort();

            fs.Seek(this.TableOffsets["maxp"].Offset, SeekOrigin.Begin);

            Tables.Add("maxp", new TrueTypeMaxpTable()
            {
                Version = fs.ReadFixed(),
                NumGlyphs = fs.ReadUShort()
            });

            if (((TrueTypeMaxpTable)Tables["maxp"]).Version.Bits == 65536)
            {
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxPoints = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxContours = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxComponentPoints = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxComponentContours = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxZones = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxTwilightPoints = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxStorage = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxFunctionDefs = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxInstructionDefs = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxStackElements = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxSizeOfInstructions = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxComponentElements = fs.ReadUShort();
                ((TrueTypeMaxpTable)Tables["maxp"]).MaxComponentDepth = fs.ReadUShort();
            }

            int totalGlyphs = ((TrueTypeMaxpTable)Tables["maxp"]).NumGlyphs;
            int numMetrics = ((TrueTypeHHeaTable)Tables["hhea"]).NumOfLongHorMetrics;

            fs.Seek(this.TableOffsets["hmtx"].Offset, SeekOrigin.Begin);
            Tables.Add("hmtx", new TrueTypeHmtxTable() { HMetrics = new LongHorFixed[numMetrics],
    LeftSideBearing = new short[totalGlyphs - numMetrics] });

            for (int i = 0; i < numMetrics; i++)
            {
                ((TrueTypeHmtxTable)Tables["hmtx"]).HMetrics[i] = new
    TrueTypeFile.LongHorFixed(fs.ReadUShort(), fs.ReadShort());
            }

            for (int i = 0; i < totalGlyphs - numMetrics; i++)
            {
                ((TrueTypeHmtxTable)Tables["hmtx"]).LeftSideBearing[i] = fs.ReadShort();
            }

            fs.Seek(this.TableOffsets["cmap"].Offset, SeekOrigin.Begin);
            Tables.Add("cmap", new TrueTypeCmapTable() { Version = fs.ReadUShort(), NumberSubTables =
    fs.ReadUShort() });

            ((TrueTypeCmapTable)Tables["cmap"]).SubTables = new
    CmapSubTable[((TrueTypeCmapTable)Tables["cmap"]).NumberSubTables];
            ((TrueTypeCmapTable)Tables["cmap"]).ActualCmapTables = new
    ICmapTable[((TrueTypeCmapTable)Tables["cmap"]).NumberSubTables];

            for (int i = 0; i < ((TrueTypeCmapTable)Tables["cmap"]).NumberSubTables; i++)
            {
                ((TrueTypeCmapTable)Tables["cmap"]).SubTables[i] = new
    TrueTypeFile.CmapSubTable(fs.ReadUShort(), fs.ReadUShort(), fs.ReadUInt());
            }


            for (int i = 0; i < ((TrueTypeCmapTable)Tables["cmap"]).NumberSubTables; i++)
            {
                fs.Seek(((TrueTypeCmapTable)Tables["cmap"]).SubTables[i].Offset +
    TableOffsets["cmap"].Offset, SeekOrigin.Begin);

                ushort format = fs.ReadUShort();
                ushort length = fs.ReadUShort();
                ushort language = fs.ReadUShort();

                if (format == 0)
                {
                    byte[] glyphIndexArray = new byte[256];
                    fs.Read(glyphIndexArray, 0, 256);
                    ((TrueTypeCmapTable)Tables["cmap"]).ActualCmapTables[i] = new CmapTable0(format,
    length, language, glyphIndexArray);
                }
                else if (format == 4)
                {
                    ((TrueTypeCmapTable)Tables["cmap"]).ActualCmapTables[i] = new CmapTable4(format,
    length, language, fs);
                }
            }

```

```
            if (this.TableOffsets.ContainsKey("OS/2"))
            {
                fs.Seek(this.TableOffsets["OS/2"].Offset, SeekOrigin.Begin);

                TrueTypeOS2Table os2 = new TrueTypeOS2Table()
                {
                    Version = fs.ReadUShort(),
                    XAvgCharWidth = fs.ReadShort(),
                    UsWeightClass = fs.ReadUShort(),
                    UsWidthClass = fs.ReadUShort(),
                    FsType = fs.ReadShort(),
                    YSubscriptXSize = fs.ReadShort(),
                    YSubscriptYSize = fs.ReadShort(),
                    YSubscriptXOffset = fs.ReadShort(),
                    YSubscriptYOffset = fs.ReadShort(),
                    YSuperscriptXSize = fs.ReadShort(),
                    YSuperscriptYSize = fs.ReadShort(),
                    YSuperscriptXOffset = fs.ReadShort(),
                    YSuperscriptYOffset = fs.ReadShort(),
                    YStrikeoutSize = fs.ReadShort(),
                    YStrikeoutPosition = fs.ReadShort(),
                    SFamilyClass = (byte)fs.ReadByte(),
                    SFamilySubClass = (byte)fs.ReadByte(),
                    Panose = new TrueTypeOS2Table.PANOSE((byte)fs.ReadByte(), (byte)fs.ReadByte(),
(byte)fs.ReadByte(), (byte)fs.ReadByte(), (byte)fs.ReadByte(), (byte)fs.ReadByte(),
(byte)fs.ReadByte(), (byte)fs.ReadByte(), (byte)fs.ReadByte(), (byte)fs.ReadByte()),
                    UlUnicodeRange = new uint[] { fs.ReadUInt(), fs.ReadUInt(), fs.ReadUInt(),
fs.ReadUInt() },
                    AchVendID = new byte[] { (byte)fs.ReadByte(), (byte)fs.ReadByte(),
(byte)fs.ReadByte(), (byte)fs.ReadByte() },
                    FsSelection = fs.ReadUShort(),
                    FsFirstCharIndex = fs.ReadUShort(),
                    FsLastCharIndex = fs.ReadUShort()
                };

                if (os2.Version > 0 || os2.Version == 0 && this.TableOffsets["OS/2"].Length > 68)
                {
                    os2.STypoAscender = fs.ReadShort();
                    os2.STypoDescender = fs.ReadShort();
                    os2.STypoLineGap = fs.ReadShort();
                    os2.UsWinAscent = fs.ReadUShort();
                    os2.UsWinDescent = fs.ReadUShort();

                    if (os2.Version >= 1)
                    {
                        os2.UlCodePageRange = new uint[] { fs.ReadUInt(), fs.ReadUInt() };

                        if (os2.Version >= 2)
                        {
                            os2.SxHeight = fs.ReadShort();
                            os2.SCapHeight = fs.ReadShort();
                            os2.UsDefaultChar = fs.ReadUShort();
                            os2.UsBreakChar = fs.ReadUShort();
                            os2.UsMaxContext = fs.ReadUShort();

                            if (os2.Version >= 5)
                            {
                                os2.UsLowerOpticalPointSize = fs.ReadUShort();
                                os2.UsUpperOpticalPointSize = fs.ReadUShort();
                            }
                        }
                    }
                }


                Tables.Add("OS/2", os2);
            }

            if (this.TableOffsets.ContainsKey("post"))
            {
                fs.Seek(this.TableOffsets["post"].Offset, SeekOrigin.Begin);

                TrueTypePostTable post = new TrueTypePostTable()
                {
                    Version = fs.ReadFixed(),
                    ItalicAngle = fs.ReadFixed(),
                    UnderlinePosition = fs.ReadShort(),
                    UnderlineThickness = fs.ReadShort(),
                    IsFixedPitch = fs.ReadUInt(),
                    MinMemType42 = fs.ReadUInt(),
                    MaxMemType42 = fs.ReadUInt(),
                    MinMemType1 = fs.ReadUInt(),
                    MaxMemType1 = fs.ReadUInt()
                };

                if (post.Version.Bits == 0x00020000)
                {
```

```
                    post.NumGlyphs = fs.ReadUShort();
                    post.GlyphNameIndex = new ushort[post.NumGlyphs];

                    int numberNewGlyphs = 0;

                    for (int i = 0; i < post.NumGlyphs; i++)
                    {
                        post.GlyphNameIndex[i] = fs.ReadUShort();
                        if (post.GlyphNameIndex[i] >= 258)
                        {
                            numberNewGlyphs++;
                        }
                    }

                    post.Names = new string[numberNewGlyphs];

                    for (int i = 0; i < numberNewGlyphs; i++)
                    {
                        post.Names[i] = fs.ReadPascalString();
                    }
                }
                Tables.Add("post", post);
            }

            if (this.TableOffsets.ContainsKey("name"))
            {
                fs.Seek(this.TableOffsets["name"].Offset, SeekOrigin.Begin);
                Tables.Add("name", new TrueTypeNameTable(this.TableOffsets["name"].Offset, fs));
            }

            fs.Seek(this.TableOffsets["loca"].Offset, SeekOrigin.Begin);
            Tables.Add("loca", new TrueTypeLocaTable(fs,
    ((TrueTypeMaxpTable)this.Tables["maxp"]).NumGlyphs,
    ((TrueTypeHeadTable)this.Tables["head"]).IndexToLocFormat == 0));

            if (this.TableOffsets.ContainsKey("cvt "))
            {
                fs.Seek(this.TableOffsets["cvt "].Offset, SeekOrigin.Begin);
                Tables.Add("cvt ", new TrueTypeRawTable(fs, this.TableOffsets["cvt "].Length));
            }

            if (this.TableOffsets.ContainsKey("prep"))
            {
                fs.Seek(this.TableOffsets["prep"].Offset, SeekOrigin.Begin);
                Tables.Add("prep", new TrueTypeRawTable(fs, this.TableOffsets["prep"].Length));
            }

            if (this.TableOffsets.ContainsKey("fpgm"))
            {
                fs.Seek(this.TableOffsets["fpgm"].Offset, SeekOrigin.Begin);
                Tables.Add("fpgm", new TrueTypeRawTable(fs, this.TableOffsets["fpgm"].Length));
            }

            if (this.TableOffsets.ContainsKey("GPOS"))
            {
                fs.Seek(this.TableOffsets["GPOS"].Offset, SeekOrigin.Begin);
                Tables.Add("GPOS", new TrueTypeGPOSTable(fs));
            }

            Glyph[] glyphs = new Glyph[((TrueTypeMaxpTable)this.Tables["maxp"]).NumGlyphs];

            for (int i = 0; i < ((TrueTypeMaxpTable)this.Tables["maxp"]).NumGlyphs; i++)
            {
                uint offset = ((TrueTypeLocaTable)Tables["loca"]).GetOffset(i);
                uint length = ((TrueTypeLocaTable)Tables["loca"]).Lengths[i];

                if (length > 0)
                {
                    fs.Seek(this.TableOffsets["glyf"].Offset + offset, SeekOrigin.Begin);
                    glyphs[i] = Glyph.Parse(fs);
                }
                else
                {
                    glyphs[i] = new EmptyGlyph();
                }

            }

            Tables.Add("glyf", new TrueTypeGlyfTable() { Glyphs = glyphs });

            BuildGlyphCache();
        }


        private double[] GlyphWidthsCache = new double[256];
        private VerticalMetrics[] VerticalMetricsCache = new VerticalMetrics[256];
        private Bearings[] BearingsCache = new Bearings[256];
```

```
        private void BuildGlyphCache()
        {
            for (int i = 0; i < 256; i++)
            {
                GlyphWidthsCache[i] = this.Get1000EmGlyphWidth(GetGlyphIndex((char)i));
                VerticalMetricsCache[i] = this.Get1000EmGlyphVerticalMetrics(GetGlyphIndex((char)i));
                BearingsCache[i] = this.Get1000EmGlyphBearings(GetGlyphIndex((char)i));
            }
        }

        /// <summary>
        /// Create a subset of the TrueType file, containing only the glyphs for the specified characters.
        /// </summary>
        /// <param name="charactersToInclude">A string containing the characters for which the glyphs should be included.</param>
        /// <param name="consolidateAt32">If true, the character map is rearranged so that the included glyphs start at the unicode U+0032 control point.</param>
        /// <param name="outputEncoding">If <paramref name="consolidateAt32"/> is true, entries will be added to this dictionary mapping the original characters to the new map (that starts at U+0033).</param>
        /// <returns></returns>
        public TrueTypeFile SubsetFont(string charactersToInclude, bool consolidateAt32 = false, Dictionary<char, char> outputEncoding = null)
        {
            if (!this.HasCmap4Table())
            {
                return this;
            }
            else
            {

                TrueTypeHeadTable head = (TrueTypeHeadTable)this.Tables["head"];

                TrueTypeHHeaTable originalHhea = (TrueTypeHHeaTable)this.Tables["hhea"];
                TrueTypeMaxpTable originalMaxp = (TrueTypeMaxpTable)this.Tables["maxp"];
                TrueTypeCmapTable originalCmap = (TrueTypeCmapTable)this.Tables["cmap"];
                TrueTypeLocaTable originalLoca = (TrueTypeLocaTable)this.Tables["loca"];
                TrueTypeGlyfTable originalGlyf = (TrueTypeGlyfTable)this.Tables["glyf"];

                CmapTable4 cmap4 = null;
                int platformId = -1;
                int platformSpecificID = -1;

                for (int i = 0; i < originalCmap.ActualCmapTables.Length; i++)
                {
                    if (originalCmap.ActualCmapTables[i] != null && originalCmap.ActualCmapTables[i].Format == 4 && originalCmap.SubTables[i].PlatformID == 3 && originalCmap.SubTables[i].PlatformSpecificID == 1)
                    {
                        cmap4 = (CmapTable4)originalCmap.ActualCmapTables[i];
                        platformId = originalCmap.SubTables[i].PlatformID;
                        platformSpecificID = originalCmap.SubTables[i].PlatformSpecificID;
                        break;
                    }
                }

                if (cmap4 == null)
                {
                    return this;
                }

                List<int> characterCodes = new List<int>();

                for (int i = 0; i < charactersToInclude.Length; i++)
                {
                    characterCodes.Add((int)charactersToInclude[i]);
                }

                characterCodes.Sort();

                List<(int, int)> segments = new List<(int, int)>();

                for (int i = 0; i < characterCodes.Count; i++)
                {
                    if (segments.Count > 0 && segments.Last().Item1 + segments.Last().Item2 == characterCodes[i])
                    {
                        segments[segments.Count - 1] = (segments[segments.Count - 1].Item1, segments[segments.Count - 1].Item2 + 1);
                    }
                    else
                    {
                        segments.Add((characterCodes[i], 1));
                    }
                }

```

```
                segments.Add((cmap4.StartCode.Last(), cmap4.EndCode.Last() - cmap4.StartCode.Last() + 1));

                int totalGlyphIndexCount = characterCodes.Count + 1;

                segments.Sort((a, b) => a.Item1 + a.Item2 - b.Item1 - b.Item2);

                ushort entrySelector = (ushort)Math.Floor(Math.Log(segments.Count, 2));


                CmapTable4 newCmap4 = new CmapTable4()
                {
                    Format = 4,
                    Length = (ushort)(16 + 8 * segments.Count),
                    Language = cmap4.Language,
                    SegCountX2 = (ushort)(2 * segments.Count),
                    SearchRange = (ushort)(2 * (1 « entrySelector)),
                    EntrySelector = entrySelector,
                    RangeShift = (ushort)(2 * segments.Count - (2 * (1 « entrySelector))),
                    EndCode = new ushort[segments.Count],
                    ReservedPad = cmap4.ReservedPad,
                    StartCode = new ushort[segments.Count],
                    IdDelta = new ushort[segments.Count],
                    IdRangeOffset = new ushort[segments.Count],
                    GlyphIndexArray = new ushort[0]
                };

                List<char> includedGlyphs = new List<char>
                {
                    (char)0
                };

                List<ushort> glyphIndexArray = new List<ushort>();

                if (!consolidateAt32)
                {
                    for (int i = 0; i < segments.Count; i++)
                    {
                        newCmap4.EndCode[i] = (ushort)(segments[i].Item1 + segments[i].Item2 - 1);
                        newCmap4.StartCode[i] = (ushort)segments[i].Item1;

                        newCmap4.IdRangeOffset[i] = 0;

                        if (i < segments.Count - 1)
                        {
                            newCmap4.IdDelta[i] = (ushort)(includedGlyphs.Count - newCmap4.StartCode[i]);

                            for (int j = newCmap4.StartCode[i]; j <= newCmap4.EndCode[i]; j++)
                            {
                                includedGlyphs.Add((char)j);
                            }
                        }
                        else
                        {
                            newCmap4.IdDelta[i] = 1;
                            newCmap4.IdRangeOffset[i] = 0;
                        }
                    }
                }
                else
                {
                    for (int i = 0; i < segments.Count - 1; i++)
                    {
                        newCmap4.EndCode[i] = (ushort)(32 + includedGlyphs.Count + segments[i].Item2 - 1);
                        newCmap4.StartCode[i] = (ushort)(32 + includedGlyphs.Count);
                        newCmap4.IdRangeOffset[i] = 0;

                        if (i < segments.Count - 1)
                        {
                            newCmap4.IdDelta[i] = (ushort)(includedGlyphs.Count - newCmap4.StartCode[i]);

                            for (int j = segments[i].Item1; j < segments[i].Item1 + segments[i].Item2; j++)
                            {
                                includedGlyphs.Add((char)j);
                            }
                        }
                        else
                        {
                            newCmap4.IdDelta[i] = 1;
                            newCmap4.IdRangeOffset[i] = 0;
                        }
                    }

```

```
                    {
                        int i = segments.Count - 1;
                        newCmap4.EndCode[i] = (ushort)(segments[i].Item1 + segments[i].Item2 - 1);
                        newCmap4.StartCode[i] = (ushort)segments[i].Item1;
                        newCmap4.IdRangeOffset[i] = 0;

                        if (i < segments.Count - 1)
                        {
                            newCmap4.IdDelta[i] = (ushort)(includedGlyphs.Count - newCmap4.StartCode[i]);

                            for (int j = newCmap4.StartCode[i]; j <= newCmap4.EndCode[i]; j++)
                            {
                                includedGlyphs.Add((char)j);
                            }
                        }
                        else
                        {
                            newCmap4.IdDelta[i] = 1;
                            newCmap4.IdRangeOffset[i] = 0;
                        }
                    }

                    if (outputEncoding != null)
                    {
                        for (int i = 1; i < includedGlyphs.Count; i++)
                        {
                            outputEncoding.Add(includedGlyphs[i], (char)(32 + i));
                        }
                    }
                }


                newCmap4.GlyphIndexArray = glyphIndexArray.ToArray();
                newCmap4.Length += (ushort)(newCmap4.GlyphIndexArray.Length * 2);

                TrueTypeCmapTable cmap = new TrueTypeCmapTable() { ActualCmapTables = new ICmapTable[] { newCmap4 }, NumberSubTables = 1, Version = 0, SubTables = new CmapSubTable[] { new CmapSubTable((ushort)platformId, (ushort)platformSpecificID, 12) } };


                List<Glyph> glyphs = new List<Glyph>();
                List<int> originalGlyphIndices = new List<int>();

                for (int i = 0; i < includedGlyphs.Count; i++)
                {
                    int index = this.GetGlyphIndex(includedGlyphs[i]);
                    originalGlyphIndices.Add(index);
                    glyphs.Add(originalGlyf.Glyphs[index].Clone());
                }

                for (int i = 0; i < glyphs.Count; i++)
                {
                    if (glyphs[i] is CompositeGlyph)
                    {
                        for (int j = 0; j < ((CompositeGlyph)glyphs[i]).GlyphIndex.Length; j++)
                        {
                            if (originalGlyphIndices.Contains(((CompositeGlyph)glyphs[i]).GlyphIndex[j]))
                            {
                                ((CompositeGlyph)glyphs[i]).GlyphIndex[j] = (ushort)originalGlyphIndices.IndexOf(((CompositeGlyph)glyphs[i]).GlyphIndex[j]);
                            }
                            else
                            {
                                originalGlyphIndices.Add(((CompositeGlyph)glyphs[i]).GlyphIndex[j]);
                                glyphs.Add(originalGlyf.Glyphs[((CompositeGlyph)glyphs[i]).GlyphIndex[j]]);
                                ((CompositeGlyph)glyphs[i]).GlyphIndex[j] = (ushort)originalGlyphIndices.IndexOf(((CompositeGlyph)glyphs[i]).GlyphIndex[j]);
                            }
                        }
                    }
                }

                TrueTypeGlyfTable glyf = new TrueTypeGlyfTable() { Glyphs = glyphs.ToArray() };

                TrueTypeLocaTable loca = new TrueTypeLocaTable(glyphs.Count, head.IndexToLocFormat == 0);

                for (int i = 0; i < glyphs.Count + 1; i++)
                {
                    if (i == 0)
                    {
                        loca.SetOffset(0, 0);
                        loca.Lengths[i] = (uint)(glyphs[i].GetBytes()).Length;
                    }
```

```
                    else
                    {
                        if (i < glyphs.Count)
                        {
                            loca.Lengths[i] = (uint)(glyphs[i].GetBytes()).Length;
                        }

                        loca.SetOffset(i, loca.GetOffset(i - 1) + loca.Lengths[i - 1]);
                    }
                }

                TrueTypeHHeaTable hhea = new TrueTypeHHeaTable()
                {
                    Version = originalHhea.Version,
                    Ascent = originalHhea.Ascent,
                    Descent = originalHhea.Descent,
                    LineGap = originalHhea.LineGap,
                    AdvanceWidthMax = originalHhea.AdvanceWidthMax,
                    MinLeftSideBearing = originalHhea.MinLeftSideBearing,
                    MinRightSideBearing = originalHhea.MinRightSideBearing,
                    XMaxExtent = originalHhea.XMaxExtent,
                    CaretSlopeRise = originalHhea.CaretSlopeRise,
                    CaretSlopeRun = originalHhea.CaretSlopeRun,
                    CaretOffset = originalHhea.CaretOffset,
                    MetricDataFormat = originalHhea.MetricDataFormat,
                    NumOfLongHorMetrics = (ushort)glyphs.Count
                };


                LongHorFixed[] metrics = new LongHorFixed[glyphs.Count];

                for (int i = 0; i < glyphs.Count; i++)
                {
                    metrics[i] = this.GetGlyphMetrics(originalGlyphIndices[i]);
                }

                TrueTypeHmtxTable hmtx = new TrueTypeHmtxTable()
                {
                    LeftSideBearing = new short[0],
                    HMetrics = metrics
                };

                TrueTypeMaxpTable maxp = new TrueTypeMaxpTable()
                {
                    Version = originalMaxp.Version,
                    NumGlyphs = (ushort)glyphs.Count,
                    MaxPoints = originalMaxp.MaxPoints,
                    MaxContours = originalMaxp.MaxContours,
                    MaxComponentPoints = originalMaxp.MaxComponentPoints,
                    MaxComponentContours = originalMaxp.MaxComponentContours,
                    MaxZones = originalMaxp.MaxZones,
                    MaxTwilightPoints = originalMaxp.MaxTwilightPoints,
                    MaxStorage = originalMaxp.MaxStorage,
                    MaxFunctionDefs = originalMaxp.MaxFunctionDefs,
                    MaxInstructionDefs = originalMaxp.MaxInstructionDefs,
                    MaxStackElements = originalMaxp.MaxStackElements,
                    MaxSizeOfInstructions = originalMaxp.MaxSizeOfInstructions,
                    MaxComponentElements = originalMaxp.MaxComponentElements,
                    MaxComponentDepth = originalMaxp.MaxComponentDepth
                };

                Dictionary<string, ITrueTypeTable> newTables = new Dictionary<string, ITrueTypeTable>() {
                    {"head", head },
                    {"hhea", hhea },
                    {"maxp", maxp },
                };

                if (Tables.ContainsKey("OS/2"))
                {
                    TrueTypeOS2Table originalOS2 = (TrueTypeOS2Table)Tables["OS/2"];

                    TrueTypeOS2Table os2 = new TrueTypeOS2Table()
                    {
                        AchVendID = originalOS2.AchVendID,
                        FsSelection = originalOS2.FsSelection,
                        FsType = originalOS2.FsType,
                        Panose = originalOS2.Panose,
                        SCapHeight = originalOS2.SCapHeight,
                        SFamilyClass = originalOS2.SFamilyClass,
                        SFamilySubClass = originalOS2.SFamilySubClass,
                        STypoAscender = originalOS2.STypoAscender,
                        STypoDescender = originalOS2.STypoDescender,
                        STypoLineGap = originalOS2.STypoLineGap,
                        SxHeight = originalOS2.SxHeight,
                        UlCodePageRange = originalOS2.UlCodePageRange,
                        UlUnicodeRange = originalOS2.UlUnicodeRange,
```

```
                        UsLowerOpticalPointSize = originalOS2.UsLowerOpticalPointSize,
                        UsBreakChar = originalOS2.UsBreakChar,
                        UsDefaultChar = originalOS2.UsDefaultChar,
                        UsMaxContext = originalOS2.UsMaxContext,
                        UsUpperOpticalPointSize = originalOS2.UsUpperOpticalPointSize,
                        UsWeightClass = originalOS2.UsWeightClass,
                        UsWidthClass = originalOS2.UsWidthClass,
                        UsWinAscent = originalOS2.UsWinAscent,
                        UsWinDescent = originalOS2.UsWinDescent,
                        Version = originalOS2.Version,
                        XAvgCharWidth = originalOS2.XAvgCharWidth,
                        YStrikeoutPosition = originalOS2.YStrikeoutPosition,
                        YStrikeoutSize = originalOS2.YStrikeoutSize,
                        YSubscriptXOffset = originalOS2.YSubscriptXOffset,
                        YSubscriptXSize = originalOS2.YSubscriptXSize,
                        YSubscriptYOffset = originalOS2.YSubscriptYOffset,
                        YSubscriptYSize = originalOS2.YSubscriptYSize,
                        YSuperscriptXOffset = originalOS2.YSuperscriptXOffset,
                        YSuperscriptXSize = originalOS2.YSuperscriptXSize,
                        YSuperscriptYOffset = originalOS2.YSuperscriptYOffset,
                        YSuperscriptYSize = originalOS2.YSuperscriptYSize,
                        FsFirstCharIndex = (ushort)characterCodes[0],
                        FsLastCharIndex = (ushort)characterCodes[characterCodes.Count - 1]
                    };

                    newTables.Add("OS/2", os2);
                }

                newTables.Add("hmtx", hmtx);
                newTables.Add("cmap", cmap);
                if (Tables.ContainsKey("fpgm"))
                {
                    TrueTypeRawTable fpgm = (TrueTypeRawTable)Tables["fpgm"];
                    newTables.Add("fpgm", fpgm);
                }
                if (Tables.ContainsKey("prep"))
                {
                    TrueTypeRawTable prep = (TrueTypeRawTable)Tables["prep"];
                    newTables.Add("prep", prep);
                }
                if (Tables.ContainsKey("cvt "))
                {
                    TrueTypeRawTable cvt = (TrueTypeRawTable)Tables["cvt "];
                    newTables.Add("cvt ", cvt);
                }
                newTables.Add("loca", loca);
                newTables.Add("glyf", glyf);

                if (Tables.ContainsKey("name"))
                {
                    TrueTypeNameTable name = (TrueTypeNameTable)Tables["name"];
                    newTables.Add("name", name);
                }

                if (Tables.ContainsKey("post"))
                {
                    TrueTypePostTable oldPost = (TrueTypePostTable)Tables["post"];

                    TrueTypePostTable post = new TrueTypePostTable()
                    {
                        //We don't really care about PostScript names
                        Version = new Fixed(0x00030000, 16),
                        ItalicAngle = oldPost.ItalicAngle,
                        UnderlinePosition = oldPost.UnderlinePosition,
                        UnderlineThickness = oldPost.UnderlineThickness,
                        IsFixedPitch = oldPost.IsFixedPitch,
                        MinMemType42 = oldPost.MinMemType42,
                        MaxMemType42 = oldPost.MaxMemType42,
                        MinMemType1 = oldPost.MinMemType1,
                        MaxMemType1 = oldPost.MaxMemType1
                    };

                    newTables.Add("post", post);
                }

                TrueTypeFile newFile = new TrueTypeFile(newTables);

                return newFile;
            }

        }

        internal class TrueTypeGlyfTable : ITrueTypeTable
        {
            public Glyph[] Glyphs;

            public byte[] GetBytes()
```

```
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    for (int i = 0; i < Glyphs.Length; i++)
                    {
                        byte[] glyph = Glyphs[i].GetBytes();
                        ms.Write(glyph, 0, glyph.Length);
                    }
                    return ms.ToArray();
                }
            }

        }

        internal abstract class Glyph
        {
            public short NumberOfContours { get; set; }
            public short XMin { get; set; }
            public short YMin { get; set; }
            public short XMax { get; set; }
            public short YMax { get; set; }

            public abstract byte[] GetBytes();
            public abstract Glyph Clone();
            public static Glyph Parse(Stream sr)
            {
                short numOfContours = sr.ReadShort();
                if (numOfContours >= 0)
                {
                    return new SimpleGlyph(sr, numOfContours);
                }
                else
                {
                    return new CompositeGlyph(sr, numOfContours);
                }
            }

            public abstract TrueTypePoint[][] GetGlyphPath(double size, int emSize, Glyph[] glyphCollection);
        }

        internal class EmptyGlyph : Glyph
        {
            public override byte[] GetBytes()
            {
                return new byte[0];
            }

            public override Glyph Clone()
            {
                return new EmptyGlyph();
            }

            public override TrueTypePoint[][] GetGlyphPath(double size, int emSize, Glyph[] glyphCollection)
            {
                return new TrueTypePoint[0][];
            }
        }
        internal class CompositeGlyph : Glyph
        {
            public ushort[] Flags { get; set; }
            public ushort[] GlyphIndex { get; set; }
            public byte[][] Argument1 { get; set; }
            public byte[][] Argument2 { get; set; }
            public byte[][] TransformationOption { get; set; }
            public ushort NumInstructions { get; set; }
            public byte[] Instructions { get; set; }

            private CompositeGlyph() { }
            public override Glyph Clone()
            {
                CompositeGlyph tbr = new CompositeGlyph()
                {
                    NumberOfContours = this.NumberOfContours,
                    XMin = this.XMin,
                    YMin = this.YMin,
                    XMax = this.XMax,
                    YMax = this.YMax,

                    Flags = new ushort[this.Flags.Length],

                    GlyphIndex = new ushort[this.GlyphIndex.Length],
                    Argument1 = new byte[this.Argument1.Length][],
                    Argument2 = new byte[this.Argument2.Length][],
                    TransformationOption = new byte[this.TransformationOption.Length][],
                    NumInstructions = this.NumInstructions
```

```
                };
                this.Flags.CopyTo(tbr.Flags, 0);

                this.GlyphIndex.CopyTo(tbr.GlyphIndex, 0);

                for (int i = 0; i < this.Argument1.Length; i++)
                {
                    tbr.Argument1[i] = (byte[])this.Argument1[i].Clone();
                }


                for (int i = 0; i < this.Argument2.Length; i++)
                {
                    tbr.Argument2[i] = (byte[])this.Argument2[i].Clone();
                }


                for (int i = 0; i < this.TransformationOption.Length; i++)
                {
                    tbr.TransformationOption[i] = (byte[])this.TransformationOption[i].Clone();
                }



                if (this.Instructions != null)
                {
                    tbr.Instructions = new byte[this.Instructions.Length];
                    this.Instructions.CopyTo(tbr.Instructions, 0);
                }
                else
                {
                    tbr.Instructions = null;
                }

                return tbr;
            }
            public CompositeGlyph(Stream sr, short numberOfContours) : base()
            {
                this.NumberOfContours = numberOfContours;
                this.XMin = sr.ReadShort();
                this.YMin = sr.ReadShort();
                this.XMax = sr.ReadShort();
                this.YMax = sr.ReadShort();

                List<ushort> flags = new List<ushort>();
                List<ushort> glyphIndex = new List<ushort>();
                List<byte[]> argument1 = new List<byte[]>();
                List<byte[]> argument2 = new List<byte[]>();
                List<byte[]> transformationOption = new List<byte[]>();

                bool moreComponents = true;

                while (moreComponents)
                {
                    flags.Add(sr.ReadUShort());

                    moreComponents = (flags.Last() & 0x0020) != 0;
                    glyphIndex.Add(sr.ReadUShort());

                    if ((flags.Last() & 0x0001) != 0)
                    {
                        argument1.Add(new byte[] { (byte)sr.ReadByte(), (byte)sr.ReadByte() });
                        argument2.Add(new byte[] { (byte)sr.ReadByte(), (byte)sr.ReadByte() });
                    }
                    else
                    {
                        argument1.Add(new byte[] { (byte)sr.ReadByte() });
                        argument2.Add(new byte[] { (byte)sr.ReadByte() });
                    }

                    if ((flags.Last() & 0x0008) != 0)
                    {
                        transformationOption.Add(new byte[] { (byte)sr.ReadByte(), (byte)sr.ReadByte()
    });
                    }
                    else if ((flags.Last() & 0x0040) != 0)
                    {
                        transformationOption.Add(new byte[] { (byte)sr.ReadByte(),
    (byte)sr.ReadByte(), (byte)sr.ReadByte(), (byte)sr.ReadByte() });
                    }
                    else if ((flags.Last() & 0x0080) != 0)
                    {
                        transformationOption.Add(new byte[] { (byte)sr.ReadByte(),
    (byte)sr.ReadByte(), (byte)sr.ReadByte(), (byte)sr.ReadByte(), (byte)sr.ReadByte(),
    (byte)sr.ReadByte(), (byte)sr.ReadByte(), (byte)sr.ReadByte() });
                    }
                    else
```

```
                    {
                        transformationOption.Add(new byte[] { });
                    }
                }

                this.Flags = flags.ToArray();
                this.GlyphIndex = glyphIndex.ToArray();
                this.Argument1 = argument1.ToArray();
                this.Argument2 = argument2.ToArray();
                this.TransformationOption = transformationOption.ToArray();

                if ((flags.Last() & 0x0100) != 0)
                {
                    this.NumInstructions = sr.ReadUShort();
                    this.Instructions = new byte[this.NumInstructions];
                    sr.Read(this.Instructions, 0, this.NumInstructions);
                }
                else
                {
                    this.NumInstructions = 0;
                    this.Instructions = null;
                }
            }

            public override byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteShort(this.NumberOfContours);
                    ms.WriteShort(this.XMin);
                    ms.WriteShort(this.YMin);
                    ms.WriteShort(this.XMax);
                    ms.WriteShort(this.YMax);

                    for (int i = 0; i < this.Flags.Length; i++)
                    {
                        ms.WriteUShort(this.Flags[i]);
                        ms.WriteUShort(this.GlyphIndex[i]);
                        ms.Write(this.Argument1[i], 0, this.Argument1[i].Length);
                        ms.Write(this.Argument2[i], 0, this.Argument2[i].Length);
                        ms.Write(this.TransformationOption[i], 0, this.TransformationOption[i].Length);
                    }

                    if (this.NumInstructions > 0)
                    {
                        ms.WriteUShort(this.NumInstructions);
                        ms.Write(this.Instructions, 0, this.NumInstructions);
                    }

                    if (ms.Length % 2 != 0)
                    {
                        ms.WriteByte(0);
                    }

                    return ms.ToArray();
                }
            }

            public override TrueTypePoint[][] GetGlyphPath(double size, int emSize, Glyph[] glyphCollection)
            {
                List<short> argument1 = new List<short>();
                List<short> argument2 = new List<short>();

                for (int i = 0; i < this.Argument1.Length; i++)
                {
                    if (this.Argument1[i].Length == 1)
                    {
                        argument1.Add(this.Argument1[i][0]);
                    }
                    else if (this.Argument1[i].Length == 2)
                    {
                        argument1.Add((short)((this.Argument1[i][0] « 8) + this.Argument1[i][1]));
                    }
                }

                for (int i = 0; i < this.Argument2.Length; i++)
                {
                    if (this.Argument2[i].Length == 1)
                    {
                        argument2.Add(this.Argument2[i][0]);
                    }
                    else if (this.Argument2[i].Length == 2)
                    {
                        argument2.Add((short)((this.Argument2[i][0] « 8) + this.Argument2[i][1]));
                    }
```

```
                }

                List<double[]> transformationOption = new List<double[]>();

                for (int i = 0; i < this.TransformationOption.Length; i++)
                {
                    if (this.TransformationOption[i].Length == 0)
                    {
                        transformationOption.Add(new double[] { });
                    }
                    else if (this.TransformationOption[i].Length == 2)
                    {
                        double val = (double)((this.TransformationOption[i][0] « 8) + this.TransformationOption[i][1]) / (1 « 14);
                        transformationOption.Add(new double[] { val });
                    }
                    else if (this.TransformationOption[i].Length == 4)
                    {
                        double val1 = (double)((this.TransformationOption[i][0] « 8) + this.TransformationOption[i][1]) / (1 « 14);
                        double val2 = (double)((this.TransformationOption[i][2] « 8) + this.TransformationOption[i][3]) / (1 « 14);
                        transformationOption.Add(new double[] { val1, val2 });
                    }
                    else if (this.TransformationOption[i].Length == 8)
                    {
                        double val1 = (double)((this.TransformationOption[i][0] « 8) + this.TransformationOption[i][1]) / (1 « 14);
                        double val2 = (double)((this.TransformationOption[i][2] « 8) + this.TransformationOption[i][3]) / (1 « 14);
                        double val3 = (double)((this.TransformationOption[i][4] « 8) + this.TransformationOption[i][5]) / (1 « 14);
                        double val4 = (double)((this.TransformationOption[i][6] « 8) + this.TransformationOption[i][7]) / (1 « 14);
                        transformationOption.Add(new double[] { val1, val2, val3, val4 });
                    }
                }

                List<TrueTypePoint[]> tbr = new List<TrueTypePoint[]>();


                for (int i = 0; i < this.GlyphIndex.Length; i++)
                {
                    TrueTypePoint[][] componentContours = glyphCollection[this.GlyphIndex[i]].GetGlyphPath(size, emSize, glyphCollection);

                    double[,] transformMatrix = new double[,] { { 1, 0 }, { 0, 1 } };

                    if ((Flags[i] & 0x0008) != 0)
                    {
                        transformMatrix[0, 0] = transformationOption[i][0];
                        transformMatrix[1, 1] = transformationOption[i][0];
                    }
                    else if ((Flags[i] & 0x0040) != 0)
                    {
                        transformMatrix[0, 0] = transformationOption[i][0];
                        transformMatrix[1, 1] = transformationOption[i][1];
                    }
                    else if ((Flags[i] & 0x0080) != 0)
                    {
                        transformMatrix[0, 0] = transformationOption[i][0];
                        transformMatrix[0, 1] = transformationOption[i][1];
                        transformMatrix[1, 0] = transformationOption[i][2];
                        transformMatrix[1, 1] = transformationOption[i][3];
                    }

                    double deltaX = 0;
                    double deltaY = 0;

                    if ((Flags[i] & 0x0002) != 0)
                    {
                        deltaX = argument1[i] * size / emSize;
                        deltaY = argument2[i] * size / emSize;

                        if ((Flags[i] & 0x0800) != 0 && (Flags[i] & 0x1000) == 0)
                        {
                            deltaX *= Magnitude(Multiply(transformMatrix, new double[] { 1, 0 }));
                            deltaY *= Magnitude(Multiply(transformMatrix, new double[] { 0, 1 }));
                        }
                    }
                    else
                    {
                        TrueTypePoint reference = GetNthElementWhere(tbr, argument1[i], el => el.IsDefinedPoint);
                        TrueTypePoint destination = GetNthElementWhere(componentContours, argument2[i], el => el.IsDefinedPoint);

```

```
                        deltaX = reference.X - destination.X;
                        deltaY = reference.Y - destination.Y;

                        //Note: this would be the sensible behaviour (i.e. make sure that reference and destination coincide after the transform), but apparently it is not the correct one.
                        /*double[] transfDestination = Multiply(transformMatrix, new double[] { destination.X, destination.Y });

                        deltaX = reference.X - transfDestination[0];
                        deltaY = reference.Y - transfDestination[1];*/
                    }

                    for (int j = 0; j < componentContours.Length; j++)
                    {
                        for (int k = 0; k < componentContours[j].Length; k++)
                        {
                            double[] transformed = Multiply(transformMatrix, new double[] { componentContours[j][k].X, componentContours[j][k].Y });

                            componentContours[j][k] = new TrueTypePoint(transformed[0] + deltaX, transformed[1] + deltaY, componentContours[j][k].IsOnCurve, componentContours[j][k].IsDefinedPoint);
                        }
                    }

                    tbr.AddRange(componentContours);
                }

                return tbr.ToArray();
            }
        }

        private static double Magnitude(double[] vector)
        {
            double tbr = 0;

            for (int i = 0; i < vector.Length; i++)
            {
                tbr += vector[i] * vector[i];
            }

            return Math.Sqrt(tbr);
        }

        private static double[] Multiply(double[,] matrix, double[] vector)
        {
            double[] tbr = new double[2];

            tbr[0] = matrix[0, 0] * vector[0] + matrix[0, 1] * vector[1];
            tbr[1] = matrix[1, 0] * vector[0] + matrix[1, 1] * vector[1];

            return tbr;
        }

        private static T GetNthElementWhere<T>(IEnumerable<IEnumerable<T>> array, int n, Func<T, bool> condition)
        {
            int index = 0;

            foreach (IEnumerable<T> arr1 in array)
            {
                foreach (T el in arr1)
                {
                    if (condition(el))
                    {
                        if (index == n)
                        {
                            return el;
                        }
                        index++;
                    }
                }
            }

            throw new IndexOutOfRangeException();
        }

        /// <summary>
        /// Represents a point in a TrueType path description.
        /// </summary>
        public struct TrueTypePoint
        {
            /// <summary>
            /// The horizontal coordinate of the point.
            /// </summary>
            public double X;

            /// <summary>
```

```
            /// The vertical coordinate of the point.
            /// </summary>
            public double Y;

            /// <summary>
            /// Whether the point is a point on the curve, or a control point of a quadratic Bezier curve.
            /// </summary>
            public bool IsOnCurve;

            internal bool IsDefinedPoint;

            internal TrueTypePoint(double x, double y, bool onCurve, bool isDefinedPoint)
            {
                this.X = x;
                this.Y = y;
                this.IsOnCurve = onCurve;
                this.IsDefinedPoint = isDefinedPoint;
            }
        }

        internal class SimpleGlyph : Glyph
        {
            public ushort[] EndPtsOfContours { get; set; }
            public ushort InstructionLength { get; set; }
            public byte[] Instructions { get; set; }
            public byte[] Flags { get; set; }
            public byte[] XCoordinates { get; set; }
            public byte[] YCoordinates { get; set; }

            private SimpleGlyph() { }
            public override Glyph Clone()
            {
                SimpleGlyph tbr = new SimpleGlyph
                {
                    NumberOfContours = this.NumberOfContours,
                    XMin = this.XMin,
                    YMin = this.YMin,
                    XMax = this.XMax,
                    YMax = this.YMax,

                    EndPtsOfContours = new ushort[this.EndPtsOfContours.Length],
                    InstructionLength = this.InstructionLength,
                    Instructions = new byte[this.Instructions.Length],
                    Flags = new byte[this.Flags.Length],
                    XCoordinates = new byte[this.XCoordinates.Length],
                    YCoordinates = new byte[this.YCoordinates.Length]
                };

                this.EndPtsOfContours.CopyTo(tbr.EndPtsOfContours, 0);
                this.Instructions.CopyTo(tbr.Instructions, 0);
                this.Flags.CopyTo(tbr.Flags, 0);
                this.XCoordinates.CopyTo(tbr.XCoordinates, 0);
                this.YCoordinates.CopyTo(tbr.YCoordinates, 0);

                return tbr;
            }

            public SimpleGlyph(Stream sr, short numberOfContours) : base()
            {
                this.NumberOfContours = numberOfContours;
                this.XMin = sr.ReadShort();
                this.YMin = sr.ReadShort();
                this.XMax = sr.ReadShort();
                this.YMax = sr.ReadShort();

                this.EndPtsOfContours = new ushort[this.NumberOfContours];
                for (int i = 0; i < this.NumberOfContours; i++)
                {
                    this.EndPtsOfContours[i] = sr.ReadUShort();
                }

                this.InstructionLength = sr.ReadUShort();
                this.Instructions = new byte[this.InstructionLength];
                for (int i = 0; i < this.InstructionLength; i++)
                {
                    this.Instructions[i] = (byte)sr.ReadByte();
                }

                List<byte> logicalFlags = new List<byte>();
                List<byte> flags = new List<byte>();

                int totalPoints = this.EndPtsOfContours[this.NumberOfContours - 1] + 1;

                int countedPoints = 0;

                while (countedPoints < totalPoints)
```

```
                {
                    flags.Add((byte)sr.ReadByte());
                    logicalFlags.Add(flags.Last());
                    countedPoints++;
                    if ((flags.Last() & 0x08) != 0)
                    {
                        byte repeats = (byte)sr.ReadByte();
                        for (int i = 0; i < repeats; i++)
                        {
                            logicalFlags.Add(flags.Last());
                            countedPoints++;
                        }
                        flags.Add(repeats);
                    }
                }

                this.Flags = flags.ToArray();

                List<byte> xCoordinates = new List<byte>();

                for (int i = 0; i < totalPoints; i++)
                {
                    bool isByte = (logicalFlags[i] & 0x02) != 0;

                    if (isByte)
                    {
                        xCoordinates.Add((byte)sr.ReadByte());
                    }
                    else if ((logicalFlags[i] & 0x10) == 0)
                    {
                        xCoordinates.Add((byte)sr.ReadByte());
                        xCoordinates.Add((byte)sr.ReadByte());
                    }
                }

                this.XCoordinates = xCoordinates.ToArray();

                List<byte> yCoordinates = new List<byte>();

                List<int> yCoordinateLengths = new List<int>();

                for (int i = 0; i < totalPoints; i++)
                {
                    bool isByte = (logicalFlags[i] & 0x04) != 0;

                    if (isByte)
                    {
                        yCoordinates.Add((byte)sr.ReadByte());
                        yCoordinateLengths.Add(1);
                    }
                    else if ((logicalFlags[i] & 0x20) == 0)
                    {
                        yCoordinates.Add((byte)sr.ReadByte());
                        yCoordinates.Add((byte)sr.ReadByte());
                        yCoordinateLengths.Add(2);
                    }
                    else
                    {
                        yCoordinateLengths.Add(0);
                    }
                }

                this.YCoordinates = yCoordinates.ToArray();
            }

            public override byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteShort(this.NumberOfContours);
                    ms.WriteShort(this.XMin);
                    ms.WriteShort(this.YMin);
                    ms.WriteShort(this.XMax);
                    ms.WriteShort(this.YMax);

                    for (int i = 0; i < this.EndPtsOfContours.Length; i++)
                    {
                        ms.WriteUShort(this.EndPtsOfContours[i]);
                    }

                    ms.WriteUShort(this.InstructionLength);
                    ms.Write(this.Instructions, 0, this.InstructionLength);
                    ms.Write(this.Flags, 0, this.Flags.Length);
                    ms.Write(this.XCoordinates, 0, this.XCoordinates.Length);
                    ms.Write(this.YCoordinates, 0, this.YCoordinates.Length);

                    if (ms.Length % 2 != 0)
```

```
                    {
                        ms.WriteByte(0);
                    }

                    return ms.ToArray();
                }
            }

            public override TrueTypePoint[][] GetGlyphPath(double size, int emSize, Glyph[] glyphCollection)
            {
                List<TrueTypePoint[]> contours = new List<TrueTypePoint[]>();

                List<TrueTypePoint> currentContour = new List<TrueTypePoint>();


                List<byte> logicalFlags = new List<byte>();

                int totalPoints = this.EndPtsOfContours[this.NumberOfContours - 1] + 1;

                int countedPoints = 0;

                int index = 0;

                while (countedPoints < totalPoints)
                {
                    logicalFlags.Add(this.Flags[index]);
                    index++;
                    countedPoints++;
                    if ((logicalFlags.Last() & 0x08) != 0)
                    {
                        byte repeats = this.Flags[index];
                        index++;
                        for (int i = 0; i < repeats; i++)
                        {
                            logicalFlags.Add(logicalFlags.Last());
                            countedPoints++;
                        }
                    }
                }

                List<short> xCoordinates = new List<short>();

                index = 0;

                for (int i = 0; i < totalPoints; i++)
                {
                    bool isByte = (logicalFlags[i] & 0x02) != 0;

                    if (isByte)
                    {
                        if ((logicalFlags[i] & 0x10) != 0)
                        {
                            xCoordinates.Add(this.XCoordinates[index]);
                        }
                        else
                        {
                            xCoordinates.Add((short)(-this.XCoordinates[index]));
                        }

                        index++;
                    }
                    else if ((logicalFlags[i] & 0x10) == 0)
                    {
                        xCoordinates.Add((short)((this.XCoordinates[index] << 8) + this.XCoordinates[index + 1]));
                        index += 2;
                    }
                    else
                    {
                        xCoordinates.Add(0);
                    }
                }

                List<short> yCoordinates = new List<short>();

                index = 0;

                for (int i = 0; i < totalPoints; i++)
                {
                    bool isByte = (logicalFlags[i] & 0x04) != 0;

                    if (isByte)
                    {
                        if ((logicalFlags[i] & 0x20) != 0)
                        {
                            yCoordinates.Add(this.YCoordinates[index]);
```

```
                        }
                        else
                        {
                            yCoordinates.Add((short)(-this.YCoordinates[index]));
                        }
                        index++;
                    }
                    else if ((logicalFlags[i] & 0x20) == 0)
                    {
                        yCoordinates.Add((short)((this.YCoordinates[index] << 8) + this.YCoordinates[index + 1]));
                        index += 2;
                    }
                    else
                    {
                        yCoordinates.Add(0);
                    }
                }

                int[] previousPoint = new int[2] { 0, 0 };

                for (int i = 0; i < totalPoints; i++)
                {
                    int absoluteX = xCoordinates[i] + previousPoint[0];
                    int absoluteY = yCoordinates[i] + previousPoint[1];

                    previousPoint[0] = absoluteX;
                    previousPoint[1] = absoluteY;

                    bool onCurve = (logicalFlags[i] & 0x01) != 0;

                    if (onCurve)
                    {
                        currentContour.Add(new TrueTypePoint(size * absoluteX / emSize, size * absoluteY / emSize, onCurve, true));
                    }
                    else
                    {
                        if (currentContour.Count > 0)
                        {
                            if (currentContour.Last().IsOnCurve)
                            {
                                currentContour.Add(new TrueTypePoint(size * absoluteX / emSize, size * absoluteY / emSize, onCurve, true));
                            }
                            else
                            {
                                double newX = size * absoluteX / emSize;
                                double newY = size * absoluteY / emSize;

                                currentContour.Add(new TrueTypePoint((newX + currentContour.Last().X) * 0.5, (newY + currentContour.Last().Y) * 0.5, true, false));
                                currentContour.Add(new TrueTypePoint(newX, newY, onCurve, true));
                            }
                        }
                        else
                        {
                            currentContour.Add(new TrueTypePoint(size * absoluteX / emSize, size * absoluteY / emSize, onCurve, true));
                        }
                    }

                    if (this.EndPtsOfContours.Contains((ushort)i))
                    {
                        if (!currentContour[0].IsOnCurve)
                        {
                            if (currentContour.Last().IsOnCurve)
                            {
                                currentContour.Insert(0, new TrueTypePoint(currentContour.Last().X, currentContour.Last().Y, currentContour.Last().IsOnCurve, false));
                            }
                            else
                            {
                                currentContour.Insert(0, new TrueTypePoint((currentContour[0].X + currentContour.Last().X) * 0.5, (currentContour[0].Y + currentContour.Last().Y) * 0.5, true, false));
                            }
                        }

                        if (!currentContour.Last().IsOnCurve)
                        {
                            currentContour.Add(new TrueTypePoint(currentContour[0].X, currentContour[0].Y, currentContour[0].IsOnCurve, false));
                        }

                        contours.Add(currentContour.ToArray());
                        currentContour = new List<TrueTypePoint>();
                    }
```

```
                }

                return contours.ToArray();
            }
        }

        internal class TrueTypeLocaTable : ITrueTypeTable
        {
            public ushort[] ShortOffsets { get; }
            public uint[] IntOffsets { get; }
            public uint[] Lengths { get; }

            public uint GetOffset(int index)
            {
                if (IntOffsets == null)
                {
                    return (uint)ShortOffsets[index] * 2;
                }
                else
                {
                    return IntOffsets[index];
                }
            }

            public void SetOffset(int index, uint value)
            {
                if (IntOffsets == null)
                {
                    ShortOffsets[index] = (ushort)(value / 2);
                }
                else
                {
                    IntOffsets[index] = value;
                }
            }

            public TrueTypeLocaTable(int numGlyphs, bool isShort)
            {
                this.Lengths = new uint[numGlyphs];

                if (isShort)
                {
                    this.ShortOffsets = new ushort[numGlyphs + 1];
                }
                else
                {
                    this.IntOffsets = new uint[numGlyphs + 1];
                }
            }

            public TrueTypeLocaTable(Stream sr, int numGlyphs, bool isShort)
            {
                this.Lengths = new uint[numGlyphs];

                if (isShort)
                {
                    this.ShortOffsets = new ushort[numGlyphs + 1];
                    for (int i = 0; i < numGlyphs + 1; i++)
                    {
                        this.ShortOffsets[i] = sr.ReadUShort();
                    }

                    for (int i = 0; i < numGlyphs; i++)
                    {
                        this.Lengths[i] = 2 * ((uint)this.ShortOffsets[i + 1] - (uint)this.ShortOffsets[i]);
                    }
                }
                else
                {
                    this.IntOffsets = new uint[numGlyphs + 1];
                    for (int i = 0; i < numGlyphs + 1; i++)
                    {
                        this.IntOffsets[i] = sr.ReadUInt();
                    }

                    for (int i = 0; i < numGlyphs; i++)
                    {
                        this.Lengths[i] = this.IntOffsets[i + 1] - this.IntOffsets[i];
                    }
                }

            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
```

```
                {

                    if (IntOffsets == null)
                    {
                        for (int i = 0; i < ShortOffsets.Length; i++)
                        {
                            ms.WriteUShort(ShortOffsets[i]);
                        }
                    }
                    else
                    {
                        for (int i = 0; i < IntOffsets.Length; i++)
                        {
                            ms.WriteUInt(IntOffsets[i]);
                        }
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class TrueTypeRawTable : ITrueTypeTable
        {
            public byte[] Data { get; }

            public TrueTypeRawTable(Stream sr, uint length)
            {
                this.Data = new byte[length];
                sr.Read(this.Data, 0, (int)length);
            }

            public byte[] GetBytes()
            {
                return Data;
            }
        }

        internal bool HasCmap4Table()
        {
            foreach (ICmapTable cmap in ((TrueTypeCmapTable)Tables["cmap"]).ActualCmapTables)
            {
                if (cmap is CmapTable4)
                {
                    return true;
                }

            }
            return false;
        }

        /// <summary>
        /// Obtains the font family name from the TrueType file.
        /// </summary>
        /// <returns>The font family name, if available; <see langword="null"/> otherwise.</returns>
        public string GetFontFamilyName()
        {
            TrueTypeNameTable name = (TrueTypeNameTable)this.Tables["name"];

            for (int i = 0; i < name.Count; i++)
            {
                if (name.NameRecords[i].NameID == 16)
                {
                    return name.Name[i];
                }
            }

            for (int i = 0; i < name.Count; i++)
            {
                if (name.NameRecords[i].NameID == 1)
                {
                    return name.Name[i];
                }
            }

            return null;
        }

        /// <summary>
        /// Obtains the full font family name from the TrueType file.
        /// </summary>
        /// <returns>The full font family name, if available; <see langword="null"/>
otherwise.</returns>
        public string GetFullFontFamilyName()
        {
            TrueTypeNameTable name = (TrueTypeNameTable)this.Tables["name"];

```

```
            for (int i = 0; i < name.Count; i++)
            {
                if (name.NameRecords[i].NameID == 4)
                {
                    return name.Name[i];
                }
            }

            for (int i = 0; i < name.Count; i++)
            {
                if (name.NameRecords[i].NameID == 6)
                {
                    return name.Name[i];
                }
            }

            return null;
        }


        /// <summary>
        /// Obtains the PostScript font name from the TrueType file.
        /// </summary>
        /// <returns>The PostScript font name, if available; <see langword="null"/> otherwise.</returns>
        public string GetFontName()
        {
            TrueTypeNameTable name = (TrueTypeNameTable)this.Tables["name"];

            for (int i = 0; i < name.Count; i++)
            {
                if (name.NameRecords[i].NameID == 6)
                {
                    return name.Name[i];
                }
            }

            return null;
        }

        /// <summary>
        /// Returns the index of the first character glyph represented by the font.
        /// </summary>
        /// <returns>The index of the first character glyph represented by the font.</returns>
        public ushort GetFirstCharIndex()
        {
            TrueTypeOS2Table os2 = (TrueTypeOS2Table)this.Tables["OS/2"];

            return os2.FsFirstCharIndex;
        }

        /// <summary>
        /// Returns the index of the last character glyph represented by the font.
        /// </summary>
        /// <returns>The index of the last character glyph represented by the font.</returns>
        public ushort GetLastCharIndex()
        {
            TrueTypeOS2Table os2 = (TrueTypeOS2Table)this.Tables["OS/2"];

            return os2.FsLastCharIndex;
        }


        /// <summary>
        /// Determines whether the typeface is Italic or Oblique or not.
        /// </summary>
        /// <returns>A <see cref="bool"/> indicating whether the typeface is Italic or Oblique or not.</returns>
        public bool IsItalic()
        {
            TrueTypeOS2Table os2 = (TrueTypeOS2Table)this.Tables["OS/2"];

            return (os2.FsSelection & 1) == 1;
        }

        /// <summary>
        /// Determines whether the typeface is Oblique or not.
        /// </summary>
        /// <returns>A <see cref="bool"/> indicating whether the typeface is Oblique or not.</returns>
        public bool IsOblique()
        {
            TrueTypeOS2Table os2 = (TrueTypeOS2Table)this.Tables["OS/2"];

            return (os2.FsSelection & 512) != 0;
        }

        /// <summary>
```

```
        /// Determines whether the typeface is Bold or not.
        /// </summary>
        /// <returns>A <see cref="bool"/> indicating whether the typeface is Bold or not</returns>
        public bool IsBold()
        {
            TrueTypeOS2Table os2 = (TrueTypeOS2Table)this.Tables["OS/2"];

            return (os2.FsSelection & 32) != 0;
        }

        /// <summary>
        /// Determines whether the typeface is fixed-pitch (aka monospaces) or not.
        /// </summary>
        /// <returns>A <see cref="bool"/> indicating whether the typeface is fixed-pitch (aka monospaces) or not.</returns>
        public bool IsFixedPitch()
        {
            TrueTypeOS2Table os2 = (TrueTypeOS2Table)this.Tables["OS/2"];

            return os2.Panose.BProportion == 9;
        }

        /// <summary>
        /// Determines whether the typeface is serifed or not.
        /// </summary>
        /// <returns>A <see cref="bool"/> indicating whether the typeface is serifed or not.</returns>
        public bool IsSerif()
        {
            TrueTypeOS2Table os2 = (TrueTypeOS2Table)this.Tables["OS/2"];

            return os2.SFamilyClass == 1 || os2.SFamilyClass == 2 || os2.SFamilyClass == 3 || os2.SFamilyClass == 4 || os2.SFamilyClass == 5 || os2.SFamilyClass == 7;
        }

        /// <summary>
        /// Determines whether the typeface is a script typeface or not.
        /// </summary>
        /// <returns>A <see cref="bool"/> indicating whether the typeface is a script typeface or not.</returns>
        public bool IsScript()
        {
            TrueTypeOS2Table os2 = (TrueTypeOS2Table)this.Tables["OS/2"];

            return os2.SFamilyClass == 10;
        }

        /// <summary>
        /// Determines the index of the glyph corresponding to a certain character.
        /// </summary>
        /// <param name="glyph">The character whose glyph is sought.</param>
        /// <returns>The index of the glyph in the TrueType file.</returns>
        public int GetGlyphIndex(char glyph)
        {
            foreach (ICmapTable cmap in ((TrueTypeCmapTable)Tables["cmap"]).ActualCmapTables)
            {
                if (cmap is CmapTable4)
                {
                    return cmap.GetGlyphIndex(glyph);
                }
            }

            foreach (ICmapTable cmap in ((TrueTypeCmapTable)Tables["cmap"]).ActualCmapTables)
            {
                if (cmap is CmapTable0)
                {
                    return cmap.GetGlyphIndex(glyph);
                }
            }

            return -1;
        }

        internal int GetGlyphWidth(int glyphIndex)
        {
            if (((TrueTypeHmtxTable)this.Tables["hmtx"]).HMetrics.Length > glyphIndex)
            {
                return ((TrueTypeHmtxTable)this.Tables["hmtx"]).HMetrics[glyphIndex].AdvanceWidth;
            }
            else
            {
                return ((TrueTypeHmtxTable)this.Tables["hmtx"]).HMetrics.Last().AdvanceWidth;
            }
        }

        internal LongHorFixed GetGlyphMetrics(int glyphIndex)
        {
            if (((TrueTypeHmtxTable)this.Tables["hmtx"]).HMetrics.Length > glyphIndex)
```

```
            {
                return ((TrueTypeHmtxTable)this.Tables["hmtx"]).HMetrics[glyphIndex];
            }
            else
            {
                return new LongHorFixed(((TrueTypeHmtxTable)this.Tables["hmtx"]).HMetrics.Last().AdvanceWidth, ((TrueTypeHmtxTable)this.Tables["hmtx"]).LeftSideBearing[glyphIndex - ((TrueTypeHmtxTable)this.Tables["hmtx"]).HMetrics.Length]);
            }
        }

        /// <summary>
        /// Get the path that describes the shape of a glyph.
        /// </summary>
        /// <param name="glyphIndex">The index of the glyph whose path is sought.</param>
        /// <param name="size">The font size to be used for the font coordinates.</param>
        /// <returns>An array of contours, each of which is itself an array of TrueType points.</returns>
        public TrueTypePoint[][] GetGlyphPath(int glyphIndex, double size)
        {
            return ((TrueTypeGlyfTable)this.Tables["glyf"]).Glyphs[glyphIndex].GetGlyphPath(size, ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm, ((TrueTypeGlyfTable)this.Tables["glyf"]).Glyphs);
        }

        /// <summary>
        /// Get the path that describes the shape of a glyph.
        /// </summary>
        /// <param name="glyph">The glyph whose path is sought.</param>
        /// <param name="size">The font size to be used for the font coordinates.</param>
        /// <returns>An array of contours, each of which is itself an array of TrueType points.</returns>
        public TrueTypePoint[][] GetGlyphPath(char glyph, double size)
        {
            return ((TrueTypeGlyfTable)this.Tables["glyf"]).Glyphs[GetGlyphIndex(glyph)].GetGlyphPath(size, ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm, ((TrueTypeGlyfTable)this.Tables["glyf"]).Glyphs);
        }

        /// <summary>
        /// Computes the advance width of a glyph, in thousandths of em unit.
        /// </summary>
        /// <param name="glyph">The glyph whose advance width is to be computed.</param>
        /// <returns>The advance width of the glyph in thousandths of em unit.</returns>
        public double Get1000EmGlyphWidth(char glyph)
        {
            int num = (int)glyph;
            if (num < 256)
            {
                return GlyphWidthsCache[num];
            }
            else
            {
                return Get1000EmGlyphWidth(GetGlyphIndex(glyph));
            }
        }

        /// <summary>
        /// Computes the advance width of a glyph, in thousandths of em unit.
        /// </summary>
        /// <param name="glyphIndex">The index of the glyph whose advance width is to be computed.</param>
        /// <returns>The advance width of the glyph in thousandths of em unit.</returns>
        public double Get1000EmGlyphWidth(int glyphIndex)
        {
            int w = GetGlyphWidth(glyphIndex);

            return w * 1000 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
        }

        /// <summary>
        /// Computes the font's Win ascent, in thousandths of em unit.
        /// </summary>
        /// <returns>The font's Win ascent in thousandths of em unit.</returns>
        public double Get1000EmWinAscent()
        {
            TrueTypeOS2Table os2 = ((TrueTypeOS2Table)this.Tables["OS/2"]);

            bool useTypoMetrics = (os2.FsSelection & 128) != 0;

            if (!useTypoMetrics)
            {
                return ((TrueTypeOS2Table)this.Tables["OS/2"]).UsWinAscent * 1000.0 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
            }
```

```
            else
            {
                return ((TrueTypeOS2Table)this.Tables["OS/2"]).STypoAscender * 1000.0 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
            }
        }

        /// <summary>
        /// Computes the font ascent, in thousandths of em unit.
        /// </summary>
        /// <returns>The font ascent in thousandths of em unit.</returns>
        public double Get1000EmAscent()
        {
            return ((TrueTypeHHeaTable)this.Tables["hhea"]).Ascent * 1000.0 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
        }


        /// <summary>
        /// Computes the font descent, in thousandths of em unit.
        /// </summary>
        /// <returns>The font descent in thousandths of em unit.</returns>
        public double Get1000EmDescent()
        {
            return ((TrueTypeHHeaTable)this.Tables["hhea"]).Descent * 1000.0 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
        }

        /// <summary>
        /// Computes the maximum height over the baseline of the font, in thousandths of em unit.
        /// </summary>
        /// <returns>The maximum height over the baseline of the font in thousandths of em unit.</returns>
        public double Get1000EmYMax()
        {
            return ((TrueTypeHeadTable)this.Tables["head"]).YMax * 1000.0 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
        }

        /// <summary>
        /// Computes the maximum depth below the baseline of the font, in thousandths of em unit.
        /// </summary>
        /// <returns>The maximum depth below the baseline of the font in thousandths of em unit.</returns>
        public double Get1000EmYMin()
        {
            return ((TrueTypeHeadTable)this.Tables["head"]).YMin * 1000.0 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
        }

        /// <summary>
        /// Computes the maximum distance to the right of the glyph origin of the font, in thousandths of em unit.
        /// </summary>
        /// <returns>The maximum distance to the right of the glyph origin of the font in thousandths of em unit.</returns>
        public double Get1000EmXMax()
        {
            return ((TrueTypeHeadTable)this.Tables["head"]).XMax * 1000.0 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
        }

        /// <summary>
        /// Computes the maximum distance to the left of the glyph origin of the font, in thousandths of em unit.
        /// </summary>
        /// <returns>The maximum distance to the left of the glyph origin of the font in thousandths of em unit.</returns>
        public double Get1000EmXMin()
        {
            return ((TrueTypeHeadTable)this.Tables["head"]).XMin * 1000.0 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
        }

        /// <summary>
        /// Represents the left- and right-side bearings of a glyph.
        /// </summary>
        public struct Bearings
        {
            /// <summary>
            /// The left-side bearing of the glyph.
            /// </summary>
            public int LeftSideBearing;

            /// <summary>
            /// The right-side bearing of the glyph.
            /// </summary>
```

```
            public int RightSideBearing;

            internal Bearings(int lsb, int rsb)
            {
                LeftSideBearing = lsb;
                RightSideBearing = rsb;
            }
        }

        internal Bearings Get1000EmGlyphBearings(int glyphIndex)
        {
            LongHorFixed metrics = GetGlyphMetrics(glyphIndex);

            int lsb = metrics.LeftSideBearing;

            Glyph glyph = ((TrueTypeGlyfTable)this.Tables["glyf"]).Glyphs[glyphIndex];

            int rsb = metrics.AdvanceWidth - (lsb + glyph.XMax - glyph.XMin);

            return new Bearings(lsb * 1000 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm, rsb * 1000 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm);
        }


        /// <summary>
        /// Computes the left- and right- side bearings of a glyph, in thousandths of em unit.
        /// </summary>
        /// <param name="glyph">The glyph whose bearings are to be computed.</param>
        /// <returns>The left- and right- side bearings of the glyph in thousandths of em unit</returns>
        public Bearings Get1000EmGlyphBearings(char glyph)
        {
            int num = (int)glyph;

            if (num < 256)
            {
                return BearingsCache[num];
            }
            else
            {
                return Get1000EmGlyphBearings(GetGlyphIndex(glyph));
            }
        }

        /// <summary>
        /// Represents the maximum heigth above and depth below the baseline of a glyph.
        /// </summary>
        public struct VerticalMetrics
        {
            /// <summary>
            /// The maximum depth below the baseline of the glyph.
            /// </summary>
            public int YMin;

            /// <summary>
            /// The maximum height above the baseline of the glyph.
            /// </summary>
            public int YMax;

            internal VerticalMetrics(int yMin, int yMax)
            {
                this.YMin = yMin;
                this.YMax = yMax;
            }
        }

        internal VerticalMetrics Get1000EmGlyphVerticalMetrics(int glyphIndex)
        {
            Glyph glyph = ((TrueTypeGlyfTable)this.Tables["glyf"]).Glyphs[glyphIndex];

            return new VerticalMetrics(glyph.YMin * 1000 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm, glyph.YMax * 1000 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm);
        }

        /// <summary>
        /// Computes the vertical metrics of a glyph, in thousandths of em unit.
        /// </summary>
        /// <param name="glyph">The glyph whose vertical metrics are to be computed.</param>
        /// <returns>The vertical metrics of a glyph, in thousandths of em unit.</returns>
        public VerticalMetrics Get1000EmGlyphVerticalMetrics(char glyph)
        {
            int num = (int)glyph;
            if (num < 256)
            {
                return VerticalMetricsCache[num];
            }
```

```
            else
            {
                return Get1000EmGlyphVerticalMetrics(GetGlyphIndex(glyph));
            }
        }

        /// <summary>
        /// Computes the distance of the top of the underline from the baseline, in thousandths of em unit.
        /// </summary>
        /// <returns>The distance of the top of the underline from the baseline, in thousandths of em unit.</returns>
        public double Get1000EmUnderlinePosition()
        {
            if (this.Tables.TryGetValue("post", out ITrueTypeTable table) && table is TrueTypePostTable post)
            {
                return post.UnderlinePosition * 1000.0 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
            }
            else
            {
                return double.NaN;
            }
        }

        /// <summary>
        /// Computes the thickness of the underline, in thousandths of em unit.
        /// </summary>
        /// <returns>The thickness of the underline, in thousandths of em unit.</returns>
        public double Get1000EmUnderlineThickness()
        {
            if (this.Tables.TryGetValue("post", out ITrueTypeTable table) && table is TrueTypePostTable post)
            {
                return post.UnderlineThickness * 1000.0 / ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;
            }
            else
            {
                return double.NaN;
            }
        }

        /// <summary>
        /// Computes the italic angle for the current font, in thousandths of em unit. This is computed from the vertical and is negative for text that leans forwards.
        /// </summary>
        /// <returns></returns>
        public double GetItalicAngle()
        {
            if (this.Tables.TryGetValue("post", out ITrueTypeTable table) && table is TrueTypePostTable post)
            {
                return post.ItalicAngle.Bits / Math.Pow(2, post.ItalicAngle.BitShifts);
            }
            else
            {
                return double.NaN;
            }
        }

        /// <summary>
        /// Computes the intersections between an underline at the specified position and thickness and a glyph, in thousandths of em units.
        /// </summary>
        /// <param name="glyph">The glyph whose intersections with the underline will be computed.</param>
        /// <param name="position">The distance of the top of the underline from the baseline, in thousandths of em unit.</param>
        /// <param name="thickness">The thickness of the underline, in thousandths of em unit.</param>
        /// <returns>If the underline does not intersect the glyph, this method returns <see langword="null" />. Otherwise, it returns an array
        /// containing two elements, representing the horizontal coordinates of the leftmost and rightmost intersection points.</returns>
        public double[] Get1000EmUnderlineIntersections(char glyph, double position, double thickness)
        {
            List<double> intersections = new List<double>();

            TrueTypePoint[][] glyphPaths = GetGlyphPath(glyph, 1000);

            for (int j = 0; j < glyphPaths.Length; j++)
            {
                double[] currPoint = new double[] { glyphPaths[j][0].X, -glyphPaths[j][0].Y };

                for (int k = 1; k < glyphPaths[j].Length; k++)
                {
```

```
                    if (glyphPaths[j][k].IsOnCurve)
                    {
                        if (-glyphPaths[j][k].Y - currPoint[1] != 0)
                        {
                            double t = (position - currPoint[1]) / (-glyphPaths[j][k].Y - currPoint[1]);
                            if (t >= 0 && t <= 1)
                            {
                                intersections.Add(currPoint[0] + t * (glyphPaths[j][k].X - currPoint[0]));
                            }

                            t = (position + thickness - currPoint[1]) / (-glyphPaths[j][k].Y - currPoint[1]);
                            if (t >= 0 && t <= 1)
                            {
                                intersections.Add(currPoint[0] + t * (glyphPaths[j][k].X - currPoint[0]));
                            }
                        }
                        else
                        {
                            if (position <= currPoint[1] && position + thickness >= currPoint[1])
                            {
                                intersections.Add(currPoint[0]);
                                intersections.Add(glyphPaths[j][k].X);
                            }
                        }

                        currPoint = new double[] { glyphPaths[j][k].X, -glyphPaths[j][k].Y };
                    }
                    else
                    {
                        double[] ctrlPoint = new double[] { glyphPaths[j][k].X, -glyphPaths[j][k].Y };
                        double[] endPoint = new double[] { glyphPaths[j][k + 1].X, -glyphPaths[j][k + 1].Y };

                        double a = currPoint[1] - 2 * ctrlPoint[1] + endPoint[1];

                        if (Math.Abs(a) < 1e-7)
                        {
                            k++;

                            if (-glyphPaths[j][k].Y - currPoint[1] != 0)
                            {
                                double t = (position - currPoint[1]) / (-glyphPaths[j][k].Y - currPoint[1]);
                                if (t >= 0 && t <= 1)
                                {
                                    intersections.Add(currPoint[0] + t * (glyphPaths[j][k].X - currPoint[0]));
                                }

                                t = (position + thickness - currPoint[1]) / (-glyphPaths[j][k].Y - currPoint[1]);
                                if (t >= 0 && t <= 1)
                                {
                                    intersections.Add(currPoint[0] + t * (glyphPaths[j][k].X - currPoint[0]));
                                }
                            }
                            else
                            {
                                if (position <= currPoint[1] && position + thickness >= currPoint[1])
                                {
                                    intersections.Add(currPoint[0]);
                                    intersections.Add(glyphPaths[j][k].X);
                                }
                            }
                        }
                        else
                        {
                            double[] ts = new double[4];

                            ts[0] = (currPoint[1] - ctrlPoint[1] - Math.Sqrt(position * a + ctrlPoint[1] * ctrlPoint[1] - currPoint[1] * endPoint[1])) / a;
                            ts[1] = (currPoint[1] - ctrlPoint[1] + Math.Sqrt(position * a + ctrlPoint[1] * ctrlPoint[1] - currPoint[1] * endPoint[1])) / a;

                            ts[2] = (currPoint[1] - ctrlPoint[1] - Math.Sqrt((position + thickness) * a + ctrlPoint[1] * ctrlPoint[1] - currPoint[1] * endPoint[1])) / a;
                            ts[3] = (currPoint[1] - ctrlPoint[1] + Math.Sqrt((position + thickness) * a + ctrlPoint[1] * ctrlPoint[1] - currPoint[1] * endPoint[1])) / a;

                            double minT = double.MaxValue;
                            double maxT = double.MinValue;
                            bool found = false;
```

```

                            for (int i = 0; i < 4; i++)
                            {
                                if (!double.IsNaN(ts[i]) && ts[i] >= 0 && ts[i] <= 1)
                                {
                                    minT = Math.Min(minT, ts[i]);
                                    maxT = Math.Max(maxT, ts[i]);

                                    found = true;
                                }
                            }

                            if (found)
                            {
                                double critT = (currPoint[0] - ctrlPoint[0]) / (currPoint[0] - 2 * ctrlPoint[0] + endPoint[0]);

                                if (critT >= minT && critT <= maxT)
                                {
                                    intersections.Add((1 - critT) * (1 - critT) * currPoint[0] + 2 * critT * (1 - critT) * ctrlPoint[0] + critT * critT * endPoint[0]);
                                }

                                intersections.Add((1 - minT) * (1 - minT) * currPoint[0] + 2 * minT * (1 - minT) * ctrlPoint[0] + minT * minT * endPoint[0]);
                                intersections.Add((1 - maxT) * (1 - maxT) * currPoint[0] + 2 * maxT * (1 - maxT) * ctrlPoint[0] + maxT * maxT * endPoint[0]);
                            }

                            k++;
                        }

                        currPoint = endPoint;
                    }
                }
            }

            if (intersections.Count > 1)
            {
                double[] tbr = new double[2] { double.MaxValue, double.MinValue };

                for (int i = 0; i < intersections.Count; i++)
                {
                    if (intersections[i] <= tbr[0])
                    {
                        tbr[0] = intersections[i];
                    }

                    if (intersections[i] >= tbr[1])
                    {
                        tbr[1] = intersections[i];
                    }
                }

                return tbr;
            }
            else
            {
                return null;
            }
        }

        /// <summary>
        /// Gets the kerning between two glyphs.
        /// </summary>
        /// <param name="glyph1">The first glyph of the kerning pair.</param>
        /// <param name="glyph2">The second glyph of the kerning pair.</param>
        /// <returns>A <see cref="PairKerning"/> object containing information about how the position of each glyphs should be altered.</returns>
        public PairKerning Get1000EmKerning(char glyph1, char glyph2)
        {
            int glyph1Index = this.GetGlyphIndex(glyph1);
            int glyph2Index = this.GetGlyphIndex(glyph2);

            return Get1000EmKerning(glyph1Index, glyph2Index);
        }

        /// <summary>
        /// Gets the kerning between two glyphs.
        /// </summary>
        /// <param name="glyph1Index">The index of the first glyph of the kerning pair.</param>
        /// <param name="glyph2Index">The index of the second glyph of the kerning pair.</param>
        /// <returns>A <see cref="PairKerning"/> object containing information about how the position of each glyphs should be altered.</returns>
        public PairKerning Get1000EmKerning(int glyph1Index, int glyph2Index)
        {
            if (this.Tables.TryGetValue("GPOS", out ITrueTypeTable table) && table is
```

```
TrueTypeGPOSTable gpos)
            {
                PairKerning kerning = gpos.GetKerning(glyph1Index, glyph2Index);

                if (kerning != null)
                {
                    double units = ((TrueTypeHeadTable)this.Tables["head"]).UnitsPerEm;

                    return new PairKerning(new Point(kerning.Glyph1Placement.X * 1000 / units, kerning.Glyph1Placement.Y * 1000 / units),
                        new Point(kerning.Glyph1Advance.X * 1000 / units, kerning.Glyph1Advance.Y * 1000 / units),
                        new Point(kerning.Glyph2Placement.X * 1000 / units, kerning.Glyph2Placement.Y * 1000 / units),
                        new Point(kerning.Glyph2Advance.X * 1000 / units, kerning.Glyph2Advance.Y * 1000 / units));
                }
                else
                {
                    return null;
                }
            }
            else
            {
                return null;
            }
        }

        internal interface ITrueTypeTable
        {
            byte[] GetBytes();
        }

        internal interface ICmapTable
        {
            ushort Format { get; }
            ushort Length { get; }
            ushort Language { get; }

            int GetGlyphIndex(char glyph);

            byte[] GetBytes();
        }

        internal class CmapTable4 : ICmapTable
        {
            public ushort Format { get; set; }
            public ushort Length { get; set; }
            public ushort Language { get; set; }
            public ushort SegCountX2 { get; set; }
            public ushort SearchRange { get; set; }
            public ushort EntrySelector { get; set; }
            public ushort RangeShift { get; set; }
            public ushort[] EndCode { get; set; }
            public ushort ReservedPad { get; set; }
            public ushort[] StartCode { get; set; }
            public ushort[] IdDelta { get; set; }
            public ushort[] IdRangeOffset { get; set; }
            public ushort[] GlyphIndexArray { get; set; }

            public CmapTable4() { }

            public CmapTable4(ushort format, ushort length, ushort language, Stream sr)
            {
                this.Format = format;
                this.Length = length;
                this.Language = language;
                this.SegCountX2 = sr.ReadUShort();

                int segCount = this.SegCountX2 / 2;

                this.SearchRange = sr.ReadUShort();
                this.EntrySelector = sr.ReadUShort();
                this.RangeShift = sr.ReadUShort();

                this.EndCode = new ushort[segCount];
                for (int i = 0; i < segCount; i++)
                {
                    this.EndCode[i] = sr.ReadUShort();
                }

                this.ReservedPad = sr.ReadUShort();

                this.StartCode = new ushort[segCount];
                for (int i = 0; i < segCount; i++)
                {
                    this.StartCode[i] = sr.ReadUShort();
```

```
                }

                this.IdDelta = new ushort[segCount];
                for (int i = 0; i < segCount; i++)
                {
                    this.IdDelta[i] = sr.ReadUShort();
                }

                this.IdRangeOffset = new ushort[segCount];
                for (int i = 0; i < segCount; i++)
                {
                    this.IdRangeOffset[i] = sr.ReadUShort();
                }

                int numGlyphIndices = (this.Length - 16 + 8 * segCount) / 2;

                this.GlyphIndexArray = new ushort[numGlyphIndices];

                for (int i = 0; i < numGlyphIndices; i++)
                {
                    this.GlyphIndexArray[i] = sr.ReadUShort();
                }
            }

            public int GetGlyphIndex(char glyph)
            {
                int code = (int)glyph;

                int endCodeInd = -1;

                for (int i = 0; i < EndCode.Length; i++)
                {
                    if (EndCode[i] >= code)
                    {
                        endCodeInd = i;
                        break;
                    }
                }

                if (StartCode[endCodeInd] <= code)
                {
                    if (IdRangeOffset[endCodeInd] != 0)
                    {
                        int glyphIndexIndex = IdRangeOffset[endCodeInd] / 2 + (code - StartCode[endCodeInd]) - (IdRangeOffset.Length - endCodeInd);

                        if (GlyphIndexArray[glyphIndexIndex] != 0)
                        {
                            return (IdDelta[endCodeInd] + GlyphIndexArray[glyphIndexIndex]) % 65536;
                        }
                        else
                        {
                            return 0;
                        }
                    }
                    else
                    {
                        return (code + IdDelta[endCodeInd]) % 65536;
                    }
                }
                else
                {
                    return 0;
                }
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort(this.Format);
                    ms.WriteUShort(this.Length);
                    ms.WriteUShort(this.Language);
                    ms.WriteUShort(this.SegCountX2);
                    ms.WriteUShort(this.SearchRange);
                    ms.WriteUShort(this.EntrySelector);
                    ms.WriteUShort(this.RangeShift);
                    for (int i = 0; i < this.EndCode.Length; i++)
                    {
                        ms.WriteUShort(this.EndCode[i]);
                    }
                    ms.WriteUShort(this.ReservedPad);

                    for (int i = 0; i < this.StartCode.Length; i++)
                    {
                        ms.WriteUShort(this.StartCode[i]);
                    }
```

```
                    for (int i = 0; i < this.IdDelta.Length; i++)
                    {
                        ms.WriteUShort(this.IdDelta[i]);
                    }
                    for (int i = 0; i < this.IdRangeOffset.Length; i++)
                    {
                        ms.WriteUShort(this.IdRangeOffset[i]);
                    }
                    for (int i = 0; i < this.GlyphIndexArray.Length; i++)
                    {
                        ms.WriteUShort(this.GlyphIndexArray[i]);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class CmapTable0 : ICmapTable
        {
            public ushort Format { get; }
            public ushort Length { get; }
            public ushort Language { get; }

            public byte[] GlyphIndexArray { get; }

            public CmapTable0(ushort format, ushort length, ushort language, byte[] glyphIndexArray)
            {
                this.Format = format;
                this.Length = length;
                this.Language = language;
                this.GlyphIndexArray = glyphIndexArray;
            }

            public int GetGlyphIndex(char glyph)
            {
                return GlyphIndexArray[(byte)glyph];
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort(this.Format);
                    ms.WriteUShort(this.Length);
                    ms.WriteUShort(this.Language);
                    ms.Write(this.GlyphIndexArray, 0, this.GlyphIndexArray.Length);
                    return ms.ToArray();
                }
            }
        }

        internal struct CmapSubTable
        {
            public ushort PlatformID;
            public ushort PlatformSpecificID;
            public uint Offset;

            public CmapSubTable(ushort platformID, ushort platformSpecificID, uint offset)
            {
                this.PlatformID = platformID;
                this.PlatformSpecificID = platformSpecificID;
                this.Offset = offset;
            }
        }

        internal class TrueTypeCmapTable : ITrueTypeTable
        {
            public ushort Version;
            public ushort NumberSubTables;
            public CmapSubTable[] SubTables;
            public ICmapTable[] ActualCmapTables;

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort(this.Version);
                    ms.WriteUShort(this.NumberSubTables);

                    for (int i = 0; i < this.SubTables.Length; i++)
                    {
                        ms.WriteUShort(this.SubTables[i].PlatformID);
                        ms.WriteUShort(this.SubTables[i].PlatformSpecificID);
                        ms.WriteUInt(this.SubTables[i].Offset);
                    }

```

```
                    for (int i = 0; i < this.ActualCmapTables.Length; i++)
                    {
                        byte[] bytes = this.ActualCmapTables[i].GetBytes();
                        ms.Write(bytes, 0, bytes.Length);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class TrueTypeNameTable : ITrueTypeTable
        {
            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort(this.Format);
                    ms.WriteUShort(this.Count);
                    ms.WriteUShort(this.StringOffset);

                    for (int i = 0; i < this.NameRecords.Length; i++)
                    {
                        byte[] bytes = this.NameRecords[i].GetBytes();
                        ms.Write(bytes, 0, bytes.Length);
                    }

                    ms.Write(this.RawBytes, 0, this.RawBytes.Length);

                    return ms.ToArray();
                }
            }

            public ushort Format;
            public ushort Count;
            public ushort StringOffset;
            public NameRecord[] NameRecords;
            public string[] Name;
            private readonly byte[] RawBytes;

            internal struct NameRecord
            {
                public ushort PlatformID;
                public ushort PlatformSpecificID;
                public ushort LanguageID;
                public ushort NameID;
                public ushort Length;
                public ushort Offset;

                public NameRecord(ushort platformID, ushort platformSpecificID, ushort languageID, ushort nameID, ushort length, ushort offset)
                {
                    this.PlatformID = platformID;
                    this.PlatformSpecificID = platformSpecificID;
                    this.LanguageID = languageID;
                    this.NameID = nameID;
                    this.Length = length;
                    this.Offset = offset;
                }

                public byte[] GetBytes()
                {
                    using (MemoryStream ms = new MemoryStream())
                    {
                        ms.WriteUShort(this.PlatformID);
                        ms.WriteUShort(this.PlatformSpecificID);
                        ms.WriteUShort(this.LanguageID);
                        ms.WriteUShort(this.NameID);
                        ms.WriteUShort(this.Length);
                        ms.WriteUShort(this.Offset);
                        return ms.ToArray();
                    }
                }
            }

            internal TrueTypeNameTable(uint tableOffset, Stream sr)
            {
                this.Format = sr.ReadUShort();
                this.Count = sr.ReadUShort();
                this.StringOffset = sr.ReadUShort();
                this.NameRecords = new NameRecord[this.Count];
                this.Name = new string[this.Count];

                int maxOffsetItem = -1;

                for (int i = 0; i < this.Count; i++)
                {
```

```
                    this.NameRecords[i] = new NameRecord(sr.ReadUShort(), sr.ReadUShort(), sr.ReadUShort(), sr.ReadUShort(), sr.ReadUShort(), sr.ReadUShort());
                    if (maxOffsetItem < 0 || this.NameRecords[i].Offset > this.NameRecords[maxOffsetItem].Offset || (this.NameRecords[i].Offset == this.NameRecords[maxOffsetItem].Offset && this.NameRecords[i].Length > this.NameRecords[maxOffsetItem].Length))
                    {
                        maxOffsetItem = i;
                    }
                }

                this.RawBytes = new byte[this.NameRecords[maxOffsetItem].Offset + this.NameRecords[maxOffsetItem].Length];
                sr.Seek(tableOffset + this.StringOffset, SeekOrigin.Begin);
                sr.Read(this.RawBytes, 0, this.RawBytes.Length);

                for (int i = 0; i < this.Count; i++)
                {
                    sr.Seek(tableOffset + this.NameRecords[i].Offset + this.StringOffset, SeekOrigin.Begin);
                    byte[] stringBytes = new byte[this.NameRecords[i].Length];
                    sr.Read(stringBytes, 0, this.NameRecords[i].Length);

                    if (this.NameRecords[i].PlatformID == 0)
                    {
                        this.Name[i] = Encoding.BigEndianUnicode.GetString(stringBytes);
                    }
                    else if (this.NameRecords[i].PlatformID == 1 && this.NameRecords[i].PlatformSpecificID == 0)
                    {
                        this.Name[i] = GetMacRomanString(stringBytes);
                    }
                    else if (this.NameRecords[i].PlatformID == 3 && (this.NameRecords[i].PlatformSpecificID == 1 || this.NameRecords[i].PlatformSpecificID == 0))
                    {
                        this.Name[i] = Encoding.BigEndianUnicode.GetString(stringBytes);
                    }
                    else
                    {
                        this.Name[i] = "Unsupported encoding: " + this.NameRecords[i].PlatformID.ToString() + "/" + this.NameRecords[i].PlatformSpecificID.ToString();
                    }

                }
            }

            private static readonly char[] MacRomanChars = new char[] { '\u0020', '\u0021', '\u0022', '\u0023', '\u0024', '\u0025', '\u0026', '\u0027', '\u0028', '\u0029', '\u002a', '\u002b', '\u002c', '\u002d', '\u002e', '\u002f', '\u0030', '\u0031', '\u0032', '\u0033', '\u0034', '\u0035', '\u0036', '\u0037', '\u0038', '\u0039', '\u003a', '\u003b', '\u003c', '\u003d', '\u003e', '\u003f', '\u0040', '\u0041', '\u0042', '\u0043', '\u0044', '\u0045', '\u0046', '\u0047', '\u0048', '\u0049', '\u004a', '\u004b', '\u004c', '\u004d', '\u004e', '\u004f', '\u0050', '\u0051', '\u0052', '\u0053', '\u0054', '\u0055', '\u0056', '\u0057', '\u0058', '\u0059', '\u005a', '\u005b', '\u005c', '\u005d', '\u005e', '\u005f', '\u0060', '\u0061', '\u0062', '\u0063', '\u0064', '\u0065', '\u0066', '\u0067', '\u0068', '\u0069', '\u006a', '\u006b', '\u006c', '\u006d', '\u006e', '\u006f', '\u0070', '\u0071', '\u0072', '\u0073', '\u0074', '\u0075', '\u0076', '\u0077', '\u0078', '\u0079', '\u007a', '\u007b', '\u007c', '\u007d', '\u007e', '\u007f', '\u00c4', '\u00c5', '\u00c7', '\u00c9', '\u00d1', '\u00d6', '\u00dc', '\u00e1', '\u00e0', '\u00e2', '\u00e4', '\u00e3', '\u00e5', '\u00e7', '\u00e9', '\u00e8', '\u00ea', '\u00eb', '\u00ed', '\u00ec', '\u00ee', '\u00ef', '\u00f1', '\u00f3', '\u00f2', '\u00f4', '\u00f6', '\u00f5', '\u00fa', '\u00f9', '\u00fb', '\u00fc', '\u2020', '\u00b0', '\u00a2', '\u00a3', '\u00a7', '\u2022', '\u00b6', '\u00df', '\u00ae', '\u00a9', '\u2122', '\u00b4', '\u00a8', '\u2260', '\u00c6', '\u00d8', '\u221e', '\u00b1', '\u2264', '\u2265', '\u00a5', '\u00b5', '\u2202', '\u2211', '\u220f', '\u03c0', '\u222b', '\u00aa', '\u00ba', '\u03a9', '\u00e6', '\u00f8', '\u00bf', '\u00a1', '\u00ac', '\u221a', '\u0192', '\u2248', '\u2206', '\u00ab', '\u00bb', '\u2026', '\u00a0', '\u00c0', '\u00c3', '\u00d5', '\u0152', '\u0153', '\u2013', '\u2014', '\u201c', '\u201d', '\u2018', '\u2019', '\u00f7', '\u25ca', '\u00ff', '\u0178', '\u2044', '\u20ac', '\u2039', '\u203a', '\ufb01', '\ufb02', '\u2021', '\u00b7', '\u201a', '\u201e', '\u2030', '\u00c2', '\u00ca', '\u00c1', '\u00cb', '\u00c8', '\u00cd', '\u00ce', '\u00cf', '\u00cc', '\u00d3', '\u00d4', '\uf8ff', '\u00d2', '\u00da', '\u00db', '\u00d9', '\u0131', '\u02c6', '\u02dc', '\u00af', '\u02d8', '\u02d9', '\u02da', '\u00b8', '\u02dd', '\u02db', '\u02c7' };

            private static string GetMacRomanString(byte[] bytes)
            {
                StringBuilder bld = new StringBuilder(bytes.Length);

                for (int i = 0; i < bytes.Length; i++)
                {
                    if (bytes[i] >= 32)
                    {
                        bld.Append(MacRomanChars[bytes[i] - 32]);
                    }
                    else
                    {
                        bld.Append((char)bytes[i]);
                    }
                }

```

```
                return bld.ToString();

            }
        }

        internal class TrueTypeOS2Table : ITrueTypeTable
        {
            public ushort Version;
            public short XAvgCharWidth;
            public ushort UsWeightClass;
            public ushort UsWidthClass;
            public short FsType;
            public short YSubscriptXSize;
            public short YSubscriptYSize;
            public short YSubscriptXOffset;
            public short YSubscriptYOffset;
            public short YSuperscriptXSize;
            public short YSuperscriptYSize;
            public short YSuperscriptXOffset;
            public short YSuperscriptYOffset;
            public short YStrikeoutSize;
            public short YStrikeoutPosition;
            public byte SFamilyClass;
            public byte SFamilySubClass;
            public PANOSE Panose;
            public uint[] UlUnicodeRange;
            public byte[] AchVendID;
            public ushort FsSelection;
            public ushort FsFirstCharIndex;
            public ushort FsLastCharIndex;
            public short STypoAscender;
            public short STypoDescender;
            public short STypoLineGap;
            public ushort UsWinAscent;
            public ushort UsWinDescent;

            public uint[] UlCodePageRange;

            public short SxHeight;
            public short SCapHeight;
            public ushort UsDefaultChar;
            public ushort UsBreakChar;
            public ushort UsMaxContext;

            public ushort UsLowerOpticalPointSize;
            public ushort UsUpperOpticalPointSize;

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort(this.Version);
                    ms.WriteShort(this.XAvgCharWidth);
                    ms.WriteUShort(this.UsWeightClass);
                    ms.WriteUShort(this.UsWidthClass);
                    ms.WriteShort(this.FsType);
                    ms.WriteShort(this.YSubscriptXSize);
                    ms.WriteShort(this.YSubscriptYSize);
                    ms.WriteShort(this.YSubscriptXOffset);
                    ms.WriteShort(this.YSubscriptYOffset);
                    ms.WriteShort(this.YSuperscriptXSize);
                    ms.WriteShort(this.YSuperscriptYSize);
                    ms.WriteShort(this.YSuperscriptXOffset);
                    ms.WriteShort(this.YSuperscriptYOffset);
                    ms.WriteShort(this.YStrikeoutSize);
                    ms.WriteShort(this.YStrikeoutPosition);
                    ms.WriteByte(this.SFamilyClass);
                    ms.WriteByte(this.SFamilySubClass);
                    ms.Write(this.Panose.GetBytes(), 0, 10);
                    for (int i = 0; i < this.UlUnicodeRange.Length; i++)
                    {
                        ms.WriteUInt(this.UlUnicodeRange[i]);
                    }
                    ms.Write(this.AchVendID, 0, this.AchVendID.Length);
                    ms.WriteUShort(this.FsSelection);
                    ms.WriteUShort(this.FsFirstCharIndex);
                    ms.WriteUShort(this.FsLastCharIndex);
                    ms.WriteShort(this.STypoAscender);
                    ms.WriteShort(this.STypoDescender);
                    ms.WriteShort(this.STypoLineGap);
                    ms.WriteUShort(this.UsWinAscent);
                    ms.WriteUShort(this.UsWinDescent);

                    if (this.Version >= 1)
                    {
                        for (int i = 0; i < this.UlCodePageRange.Length; i++)
                        {
```

```
                            ms.WriteUInt(this.UlCodePageRange[i]);
                        }

                        if (this.Version >= 2)
                        {
                            ms.WriteShort(this.SxHeight);
                            ms.WriteShort(this.SCapHeight);
                            ms.WriteUShort(this.UsDefaultChar);
                            ms.WriteUShort(this.UsBreakChar);
                            ms.WriteUShort(this.UsMaxContext);

                            if (this.Version >= 5)
                            {
                                ms.WriteUShort(this.UsLowerOpticalPointSize);
                                ms.WriteUShort(this.UsUpperOpticalPointSize);
                            }
                        }
                    }

                    return ms.ToArray();
                }
            }

            internal struct PANOSE
            {
                public byte BFamilyType;
                public byte BSerifStyle;
                public byte BWeight;
                public byte BProportion;
                public byte BContrast;
                public byte BStrokeVariation;
                public byte BArmStyle;
                public byte BLetterform;
                public byte BMidline;
                public byte BXHeight;

                public PANOSE(byte bFamilyType, byte bSerifStyle, byte bWeight, byte bProportion, byte bContrast, byte bStrokeVariation, byte bArmStyle, byte bLetterform, byte bMidline, byte bXHeight)
                {
                    this.BFamilyType = bFamilyType;
                    this.BSerifStyle = bSerifStyle;
                    this.BWeight = bWeight;
                    this.BProportion = bProportion;
                    this.BContrast = bContrast;
                    this.BStrokeVariation = bStrokeVariation;
                    this.BArmStyle = bArmStyle;
                    this.BLetterform = bLetterform;
                    this.BMidline = bMidline;
                    this.BXHeight = bXHeight;
                }

                public byte[] GetBytes()
                {
                    return new byte[] { BFamilyType, BSerifStyle, BWeight, BProportion, BContrast, BStrokeVariation, BArmStyle, BLetterform, BMidline, BXHeight };
                }
            }
        }

        internal class TrueTypePostTable : ITrueTypeTable
        {
            public Fixed Version;
            public Fixed ItalicAngle;
            public short UnderlinePosition;
            public short UnderlineThickness;
            public uint IsFixedPitch;
            public uint MinMemType42;
            public uint MaxMemType42;
            public uint MinMemType1;
            public uint MaxMemType1;

            public ushort NumGlyphs;
            public ushort[] GlyphNameIndex;
            public string[] Names;

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteFixed(this.Version);
                    ms.WriteFixed(this.ItalicAngle);
                    ms.WriteShort(this.UnderlinePosition);
                    ms.WriteShort(this.UnderlineThickness);
                    ms.WriteUInt(this.IsFixedPitch);
                    ms.WriteUInt(this.MinMemType42);
                    ms.WriteUInt(this.MaxMemType42);
                    ms.WriteUInt(this.MinMemType1);
```

```
                    ms.WriteUInt(this.MaxMemType1);

                    if (this.Version.Bits == 0x00020000)
                    {
                        ms.WriteUShort(NumGlyphs);
                        for (int i = 0; i < GlyphNameIndex.Length; i++)
                        {
                            ms.WriteUShort(GlyphNameIndex[i]);
                        }

                        for (int i = 0; i < Names.Length; i++)
                        {
                            ms.WritePascalString(Names[i]);
                        }
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class TrueTypeHmtxTable : ITrueTypeTable
        {
            public LongHorFixed[] HMetrics;
            public short[] LeftSideBearing;

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    for (int i = 0; i < HMetrics.Length; i++)
                    {
                        ms.WriteUShort(HMetrics[i].AdvanceWidth);
                        ms.WriteShort(HMetrics[i].LeftSideBearing);
                    }
                    for (int i = 0; i < LeftSideBearing.Length; i++)
                    {
                        ms.WriteShort(LeftSideBearing[i]);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class TrueTypeMaxpTable : ITrueTypeTable
        {
            public Fixed Version;
            public ushort NumGlyphs;
            public ushort MaxPoints;
            public ushort MaxContours;
            public ushort MaxComponentPoints;
            public ushort MaxComponentContours;
            public ushort MaxZones;
            public ushort MaxTwilightPoints;
            public ushort MaxStorage;
            public ushort MaxFunctionDefs;
            public ushort MaxInstructionDefs;
            public ushort MaxStackElements;
            public ushort MaxSizeOfInstructions;
            public ushort MaxComponentElements;
            public ushort MaxComponentDepth;

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteFixed(this.Version);
                    ms.WriteUShort(this.NumGlyphs);
                    ms.WriteUShort(this.MaxPoints);
                    ms.WriteUShort(this.MaxContours);
                    ms.WriteUShort(this.MaxComponentPoints);
                    ms.WriteUShort(this.MaxComponentContours);
                    ms.WriteUShort(this.MaxZones);
                    ms.WriteUShort(this.MaxTwilightPoints);
                    ms.WriteUShort(this.MaxStorage);
                    ms.WriteUShort(this.MaxFunctionDefs);
                    ms.WriteUShort(this.MaxInstructionDefs);
                    ms.WriteUShort(this.MaxStackElements);
                    ms.WriteUShort(this.MaxSizeOfInstructions);
                    ms.WriteUShort(this.MaxComponentElements);
                    ms.WriteUShort(this.MaxComponentDepth);
                    return ms.ToArray();
                }
            }
        }

```

```
        internal class TrueTypeHeadTable : ITrueTypeTable
        {
            public Fixed Version;
            public Fixed FontRevision;
            public uint ChecksumAdjustment;
            public uint MagicNumber;
            public ushort Flags;
            public ushort UnitsPerEm;
            public DateTime Created;
            public DateTime Modified;
            public short XMin;
            public short YMin;
            public short XMax;
            public short YMax;
            public ushort MacStyle;
            public ushort LowestRecPPEM;
            public short FontDirectionInt;
            public short IndexToLocFormat;
            public short GlyphDataFormat;

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteFixed(this.Version);
                    ms.WriteFixed(this.FontRevision);
                    ms.WriteUInt(this.ChecksumAdjustment);
                    ms.WriteUInt(this.MagicNumber);
                    ms.WriteUShort(this.Flags);
                    ms.WriteUShort(this.UnitsPerEm);
                    ms.WriteDate(this.Created);
                    ms.WriteDate(this.Modified);
                    ms.WriteShort(this.XMin);
                    ms.WriteShort(this.YMin);
                    ms.WriteShort(this.XMax);
                    ms.WriteShort(this.YMax);
                    ms.WriteUShort(this.MacStyle);
                    ms.WriteUShort(this.LowestRecPPEM);
                    ms.WriteShort(this.FontDirectionInt);
                    ms.WriteShort(this.IndexToLocFormat);
                    ms.WriteShort(this.GlyphDataFormat);
                    return ms.ToArray();
                }
            }

        }

        internal class TrueTypeHHeaTable : ITrueTypeTable
        {
            public Fixed Version;
            public short Ascent;
            public short Descent;
            public short LineGap;
            public ushort AdvanceWidthMax;
            public short MinLeftSideBearing;
            public short MinRightSideBearing;
            public short XMaxExtent;
            public short CaretSlopeRise;
            public short CaretSlopeRun;
            public short CaretOffset;
            public short MetricDataFormat;
            public ushort NumOfLongHorMetrics;

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteFixed(this.Version);
                    ms.WriteShort(this.Ascent);
                    ms.WriteShort(this.Descent);
                    ms.WriteShort(this.LineGap);
                    ms.WriteUShort(this.AdvanceWidthMax);
                    ms.WriteShort(this.MinLeftSideBearing);
                    ms.WriteShort(this.MinRightSideBearing);
                    ms.WriteShort(this.XMaxExtent);
                    ms.WriteShort(this.CaretSlopeRise);
                    ms.WriteShort(this.CaretSlopeRun);
                    ms.WriteShort(this.CaretOffset);
                    ms.WriteShort(0);
                    ms.WriteShort(0);
                    ms.WriteShort(0);
                    ms.WriteShort(0);
                    ms.WriteShort(this.MetricDataFormat);
                    ms.WriteUShort(this.NumOfLongHorMetrics);
                    return ms.ToArray();
                }
            }
```

```
        }


        internal class LangSysTable
        {
            public string Tag;
            public ushort LookupOrderOffset;
            public ushort RequiredFeatureIndex;
            public ushort[] FeatureIndices;

            public LangSysTable(string tag, Stream fs)
            {
                this.Tag = tag;
                this.LookupOrderOffset = fs.ReadUShort();
                this.RequiredFeatureIndex = fs.ReadUShort();
                ushort featureIndexCount = fs.ReadUShort();

                this.FeatureIndices = new ushort[featureIndexCount];

                for (int i = 0; i < featureIndexCount; i++)
                {
                    this.FeatureIndices[i] = fs.ReadUShort();
                }
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream(6 + this.FeatureIndices.Length * 2))
                {
                    ms.WriteUShort(this.LookupOrderOffset);
                    ms.WriteUShort(this.RequiredFeatureIndex);
                    ms.WriteUShort((ushort)this.FeatureIndices.Length);
                    for (int i = 0; i < this.FeatureIndices.Length; i++)
                    {
                        ms.WriteUShort(this.FeatureIndices[i]);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class ScriptTable
        {
            public string Tag;
            public LangSysTable DefaultLangSys;
            public LangSysTable[] LangSysRecords;

            public ScriptTable(string tag, Stream fs)
            {
                this.Tag = tag;

                long startPosition = fs.Position;

                ushort defaultLangSysOffset = fs.ReadUShort();
                ushort langSysCount = fs.ReadUShort();

                (string, ushort)[] langSysRecords = new (string, ushort)[langSysCount];

                for (int i = 0; i < langSysCount; i++)
                {
                    StringBuilder langSysTag = new StringBuilder(4);
                    langSysTag.Append((char)fs.ReadByte());
                    langSysTag.Append((char)fs.ReadByte());
                    langSysTag.Append((char)fs.ReadByte());
                    langSysTag.Append((char)fs.ReadByte());

                    langSysRecords[i] = (langSysTag.ToString(), fs.ReadUShort());
                }

                if (defaultLangSysOffset != 0)
                {
                    fs.Seek(startPosition + defaultLangSysOffset, SeekOrigin.Begin);
                    this.DefaultLangSys = new LangSysTable(null, fs);
                }
                else
                {
                    this.DefaultLangSys = null;
                }

                this.LangSysRecords = new LangSysTable[langSysCount];

                for (int i = 0; i < langSysCount; i++)
                {
                    fs.Seek(startPosition + langSysRecords[i].Item2, SeekOrigin.Begin);
                    this.LangSysRecords[i] = new LangSysTable(langSysRecords[i].Item1, fs);
                }
```

```
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    int offset = 6 * this.LangSysRecords.Length + 2 + 2;

                    if (this.DefaultLangSys != null)
                    {
                        ms.WriteUShort((ushort)offset);
                    }
                    else
                    {
                        ms.WriteUShort(0);
                    }

                    ms.WriteUShort((ushort)this.LangSysRecords.Length);

                    byte[] defaultLangSysTable = DefaultLangSys != null ? DefaultLangSys.GetBytes() : new byte[0];

                    byte[][] langSysTables = new byte[this.LangSysRecords.Length][];

                    offset += defaultLangSysTable.Length;

                    int[] offsets = new int[this.LangSysRecords.Length];

                    for (int i = 0; i < this.LangSysRecords.Length; i++)
                    {
                        offsets[i] = offset;
                        langSysTables[i] = this.LangSysRecords[i].GetBytes();
                        offset += langSysTables[i].Length;
                    }

                    for (int i = 0; i < this.LangSysRecords.Length; i++)
                    {
                        ms.WriteByte((byte)this.LangSysRecords[i].Tag[0]);
                        ms.WriteByte((byte)this.LangSysRecords[i].Tag[1]);
                        ms.WriteByte((byte)this.LangSysRecords[i].Tag[2]);
                        ms.WriteByte((byte)this.LangSysRecords[i].Tag[3]);

                        ms.WriteUShort((ushort)offsets[i]);
                    }

                    ms.Write(defaultLangSysTable, 0, defaultLangSysTable.Length);

                    for (int i = 0; i < langSysTables[i].Length; i++)
                    {
                        ms.Write(langSysTables[i], 0, langSysTables[i].Length);
                    }

                    return ms.ToArray();
                }
            }
        }


        internal class ScriptList
        {
            public ScriptTable[] ScriptRecords;
            public ScriptList(Stream fs)
            {
                long startPosition = fs.Position;

                ushort scriptCount = fs.ReadUShort();

                (string, ushort)[] scriptRecords = new (string, ushort)[scriptCount];

                for (int i = 0; i < scriptCount; i++)
                {
                    StringBuilder scriptTag = new StringBuilder(4);
                    scriptTag.Append((char)fs.ReadByte());
                    scriptTag.Append((char)fs.ReadByte());
                    scriptTag.Append((char)fs.ReadByte());
                    scriptTag.Append((char)fs.ReadByte());

                    scriptRecords[i] = (scriptTag.ToString(), fs.ReadUShort());
                }

                this.ScriptRecords = new ScriptTable[scriptCount];

                for (int i = 0; i < scriptCount; i++)
                {
                    fs.Seek(startPosition + scriptRecords[i].Item2, SeekOrigin.Begin);
                    this.ScriptRecords[i] = new ScriptTable(scriptRecords[i].Item1, fs);
                }
```

```
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    int offset = 2 + 6 * this.ScriptRecords.Length;

                    ms.WriteUShort((ushort)this.ScriptRecords.Length);

                    byte[][] scripts = new byte[this.ScriptRecords.Length][];
                    int[] offsets = new int[this.ScriptRecords.Length];

                    for (int i = 0; i < this.ScriptRecords.Length; i++)
                    {
                        offsets[i] = offset;
                        scripts[i] = this.ScriptRecords[i].GetBytes();
                        offset += scripts[i].Length;
                    }

                    for (int i = 0; i < this.ScriptRecords.Length; i++)
                    {
                        ms.WriteByte((byte)this.ScriptRecords[i].Tag[0]);
                        ms.WriteByte((byte)this.ScriptRecords[i].Tag[1]);
                        ms.WriteByte((byte)this.ScriptRecords[i].Tag[2]);
                        ms.WriteByte((byte)this.ScriptRecords[i].Tag[3]);

                        ms.WriteUShort((ushort)offsets[i]);
                    }

                    for (int i = 0; i < this.ScriptRecords.Length; i++)
                    {
                        ms.Write(scripts[i], 0, scripts[i].Length);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class LookupTable
        {
            public ushort LookupType;
            public ushort LookupFlag;
            public ITrueTypeTable[] SubTables;
            public ushort MarkFilteringSet;

            public LookupTable(Stream fs)
            {
                long startPosition = fs.Position;

                this.LookupType = fs.ReadUShort();
                this.LookupFlag = fs.ReadUShort();

                ushort subTableCount = fs.ReadUShort();
                ushort[] subTableOffsets = new ushort[subTableCount];

                for (int i = 0; i < subTableCount; i++)
                {
                    subTableOffsets[i] = fs.ReadUShort();
                }

                if ((this.LookupFlag & 0x0010) != 0)
                {
                    this.MarkFilteringSet = fs.ReadUShort();
                }

                if (this.LookupType == 1)
                {
                    this.SubTables = new ITrueTypeTable[subTableCount];

                    for (int i = 0; i < subTableCount; i++)
                    {
                        fs.Seek(startPosition + subTableOffsets[i], SeekOrigin.Begin);
                        this.SubTables[i] = new SinglePosSubtable(fs);
                    }
                }
                else if (this.LookupType == 2)
                {
                    this.SubTables = new ITrueTypeTable[subTableCount];

                    for (int i = 0; i < subTableCount; i++)
                    {
                        fs.Seek(startPosition + subTableOffsets[i], SeekOrigin.Begin);
                        this.SubTables[i] = new PairPosSubtable(fs);
                    }
                }
```

```
                else
                {
                    this.SubTables = new ITrueTypeTable[0];
                }
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort(this.LookupType);
                    ms.WriteUShort(this.LookupFlag);
                    ms.WriteUShort((ushort)this.SubTables.Length);

                    int offset = 6 + 2 * this.SubTables.Length;

                    if ((this.LookupFlag & 0x0010) != 0)
                    {
                        offset += 2;
                    }

                    byte[][] subTables = new byte[this.SubTables.Length][];
                    int[] offsets = new int[this.SubTables.Length];

                    for (int i = 0; i < this.SubTables.Length; i++)
                    {
                        offsets[i] = offset;
                        subTables[i] = this.SubTables[i].GetBytes();
                        offset += subTables[i].Length;
                    }

                    for (int i = 0; i < this.SubTables.Length; i++)
                    {
                        ms.WriteUShort((ushort)offsets[i]);
                    }

                    if ((this.LookupFlag & 0x0010) != 0)
                    {
                        ms.WriteUShort(this.MarkFilteringSet);
                    }

                    for (int i = 0; i < this.SubTables.Length; i++)
                    {
                        ms.Write(subTables[i], 0, subTables[i].Length);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class LookupList
        {
            public LookupTable[] LookupTables;

            public LookupList(Stream fs)
            {
                long startPosition = fs.Position;

                ushort lookupCount = fs.ReadUShort();
                ushort[] lookupOffsets = new ushort[lookupCount];

                for (int i = 0; i < lookupCount; i++)
                {
                    lookupOffsets[i] = fs.ReadUShort();
                }

                this.LookupTables = new LookupTable[lookupCount];

                for (int i = 0; i < lookupCount; i++)
                {
                    fs.Seek(startPosition + lookupOffsets[i], SeekOrigin.Begin);
                    this.LookupTables[i] = new LookupTable(fs);
                }
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort((ushort)this.LookupTables.Length);

                    int offset = 2 + 2 * this.LookupTables.Length;

                    int[] offsets = new int[this.LookupTables.Length];
                    byte[][] lookupTables = new byte[this.LookupTables.Length][];

```

```
                    for (int i = 0; i < this.LookupTables.Length; i++)
                    {
                        offsets[i] = offset;
                        lookupTables[i] = this.LookupTables[i].GetBytes();
                        offset += lookupTables[i].Length;
                    }

                    for (int i = 0; i < this.LookupTables.Length; i++)
                    {
                        ms.WriteUShort((ushort)offsets[i]);
                    }

                    for (int i = 0; i < this.LookupTables.Length; i++)
                    {
                        ms.Write(lookupTables[i], 0, lookupTables[i].Length);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class CoverageTable
        {
            public struct RangeRecord
            {
                public ushort StartGlyphID;
                public ushort EndGlyphID;
                public ushort StartCoverageIndex;

                public RangeRecord(Stream fs)
                {
                    this.StartGlyphID = fs.ReadUShort();
                    this.EndGlyphID = fs.ReadUShort();
                    this.StartCoverageIndex = fs.ReadUShort();
                }
            }

            public ushort CoverageFormat;

            public ushort[] GlyphArray;
            public RangeRecord[] RangeRecords;

            public int ContainsGlyph(int glyphIndex)
            {
                if (this.CoverageFormat == 1)
                {
                    for (int i = 0; i < this.GlyphArray.Length; i++)
                    {
                        if (this.GlyphArray[i] == glyphIndex)
                        {
                            return i;
                        }
                    }

                    return -1;
                }
                else if (this.CoverageFormat == 2)
                {
                    for (int i = 0; i < this.RangeRecords.Length; i++)
                    {
                        if (this.RangeRecords[i].StartGlyphID <= glyphIndex &&
this.RangeRecords[i].EndGlyphID >= glyphIndex)
                        {
                            return this.RangeRecords[i].StartCoverageIndex + (glyphIndex -
this.RangeRecords[i].StartGlyphID);
                        }
                    }

                    return -1;
                }
                else
                {
                    return -1;
                }
            }

            public CoverageTable(Stream fs)
            {
                this.CoverageFormat = fs.ReadUShort();

                if (this.CoverageFormat == 1)
                {
                    ushort glyphCount = fs.ReadUShort();
                    this.GlyphArray = new ushort[glyphCount];
                    for (int i = 0; i < glyphCount; i++)
                    {
```

```
                        this.GlyphArray[i] = fs.ReadUShort();
                    }
                }
                else if (this.CoverageFormat == 2)
                {
                    ushort rangeCount = fs.ReadUShort();

                    this.RangeRecords = new RangeRecord[rangeCount];

                    for (int i = 0; i < rangeCount; i++)
                    {
                        this.RangeRecords[i] = new RangeRecord(fs);
                    }
                }
            }

            public byte[] GetBytes()
            {
                if (this.CoverageFormat == 1)
                {
                    using (MemoryStream ms = new MemoryStream(4 + 2 * this.GlyphArray.Length))
                    {
                        ms.WriteUShort(this.CoverageFormat);
                        ms.WriteUShort((ushort)this.GlyphArray.Length);
                        for (int i = 0; i < this.GlyphArray.Length; i++)
                        {
                            ms.WriteUShort(this.GlyphArray[i]);
                        }

                        return ms.ToArray();
                    }
                }
                else if (this.CoverageFormat == 2)
                {
                    using (MemoryStream ms = new MemoryStream(4 + 6 * this.RangeRecords.Length))
                    {
                        ms.WriteUShort(this.CoverageFormat);
                        ms.WriteUShort((ushort)this.RangeRecords.Length);
                        for (int i = 0; i < this.RangeRecords.Length; i++)
                        {
                            ms.WriteUShort(this.RangeRecords[i].StartGlyphID);
                            ms.WriteUShort(this.RangeRecords[i].EndGlyphID);
                            ms.WriteUShort(this.RangeRecords[i].StartCoverageIndex);
                        }

                        return ms.ToArray();
                    }
                }
                else
                {
                    using (MemoryStream ms = new MemoryStream(4 + 2 * this.GlyphArray.Length))
                    {
                        ms.WriteUShort(this.CoverageFormat);

                        return ms.ToArray();
                    }
                }
            }
        }

        internal class ClassDefinitionTable
        {
            public struct ClassRangeRecord
            {
                public ushort StartGlyphID;
                public ushort EndGlyphID;
                public ushort Class;

                public ClassRangeRecord(Stream fs)
                {
                    this.StartGlyphID = fs.ReadUShort();
                    this.EndGlyphID = fs.ReadUShort();
                    this.Class = fs.ReadUShort();
                }
            }

            public ushort ClassFormat;
            public ushort StartGlyphID;
            public ushort[] ClassValueArray;

            public ClassRangeRecord[] ClassRangeRecords;

            public ClassDefinitionTable(Stream fs)
            {
                this.ClassFormat = fs.ReadUShort();

                if (this.ClassFormat == 1)
```

```
                {
                    this.StartGlyphID = fs.ReadUShort();
                    ushort glyphCount = fs.ReadUShort();

                    this.ClassValueArray = new ushort[glyphCount];

                    for (int i = 0; i < this.ClassValueArray.Length; i++)
                    {
                        this.ClassValueArray[i] = fs.ReadUShort();
                    }
                }
                else if (this.ClassFormat == 2)
                {
                    ushort classRangeCount = fs.ReadUShort();

                    this.ClassRangeRecords = new ClassRangeRecord[classRangeCount];

                    for (int i = 0; i < classRangeCount; i++)
                    {
                        this.ClassRangeRecords[i] = new ClassRangeRecord(fs);
                    }
                }
            }

            public int GetClass(int glyphIndex)
            {
                if (this.ClassFormat == 1)
                {
                    if (glyphIndex >= StartGlyphID && glyphIndex - StartGlyphID < this.ClassValueArray.Length)
                    {
                        return this.ClassValueArray[glyphIndex - StartGlyphID];
                    }
                    else
                    {
                        return -1;
                    }
                }
                else if (this.ClassFormat == 2)
                {
                    for (int i = 0; i < this.ClassRangeRecords.Length; i++)
                    {
                        if (this.ClassRangeRecords[i].StartGlyphID <= glyphIndex && this.ClassRangeRecords[i].EndGlyphID >= glyphIndex)
                        {
                            return this.ClassRangeRecords[i].Class;
                        }
                    }
                    return -1;
                }
                else
                {
                    return -1;
                }
            }

            public byte[] GetBytes()
            {
                if (this.ClassFormat == 1)
                {
                    using (MemoryStream ms = new MemoryStream(6 + 2 * this.ClassValueArray.Length))
                    {
                        ms.WriteUShort(this.ClassFormat);
                        ms.WriteUShort(this.StartGlyphID);
                        ms.WriteUShort((ushort)this.ClassValueArray.Length);

                        for (int i = 0; i < this.ClassValueArray.Length; i++)
                        {
                            ms.WriteUShort(this.ClassValueArray[i]);
                        }

                        return ms.ToArray();
                    }
                }
                else if (this.ClassFormat == 2)
                {
                    using (MemoryStream ms = new MemoryStream(4 + 6 * this.ClassRangeRecords.Length))
                    {
                        ms.WriteUShort(this.ClassFormat);
                        ms.WriteUShort((ushort)this.ClassRangeRecords.Length);

                        for (int i = 0; i < this.ClassRangeRecords.Length; i++)
                        {
                            ms.WriteUShort(this.ClassRangeRecords[i].StartGlyphID);
                            ms.WriteUShort(this.ClassRangeRecords[i].EndGlyphID);
                            ms.WriteUShort(this.ClassRangeRecords[i].Class);
                        }
```

```
                        return ms.ToArray();
                    }
                }
                else
                {
                    using (MemoryStream ms = new MemoryStream(2))
                    {
                        ms.WriteUShort(this.ClassFormat);

                        return ms.ToArray();
                    }
                }
            }
        }

        internal class FeatureTable
        {
            public string Tag;
            public byte[] FeatureParams;
            public ushort[] LookupListIndices;

            public FeatureTable(string tag, Stream fs)
            {
                this.Tag = tag;

                long startPosition = fs.Position;

                ushort featureParamsOffset = fs.ReadUShort();

                ushort lookupIndexCount = fs.ReadUShort();
                this.LookupListIndices = new ushort[lookupIndexCount];

                for (int i = 0; i < lookupIndexCount; i++)
                {
                    this.LookupListIndices[i] = fs.ReadUShort();
                }

                if (featureParamsOffset != 0 && tag == "size")
                {
                    fs.Seek(startPosition + featureParamsOffset, SeekOrigin.Begin);

                    this.FeatureParams = new byte[10];

                    for (int i = 0; i < 10; i++)
                    {
                        this.FeatureParams[i] = (byte)fs.ReadByte();
                    }
                }
                else
                {
                    this.FeatureParams = null;
                }
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    if (this.FeatureParams != null)
                    {
                        ms.WriteUShort((ushort)(4 + 2 * this.LookupListIndices.Length));
                    }
                    else
                    {
                        ms.WriteUShort(0);
                    }

                    ms.WriteUShort((ushort)this.LookupListIndices.Length);

                    for (int i = 0; i < this.LookupListIndices.Length; i++)
                    {
                        ms.WriteUShort(this.LookupListIndices[i]);
                    }

                    if (this.FeatureParams != null)
                    {
                        ms.Write(this.FeatureParams, 0, this.FeatureParams.Length);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class FeatureList
        {
```

```
            public FeatureTable[] FeatureRecords;

            public FeatureList(Stream fs)
            {
                long startPosition = fs.Position;

                ushort featureCount = fs.ReadUShort();

                (string, ushort)[] featureRecords = new (string, ushort)[featureCount];

                for (int i = 0; i < featureCount; i++)
                {
                    StringBuilder featureTag = new StringBuilder(4);
                    featureTag.Append((char)fs.ReadByte());
                    featureTag.Append((char)fs.ReadByte());
                    featureTag.Append((char)fs.ReadByte());
                    featureTag.Append((char)fs.ReadByte());

                    featureRecords[i] = (featureTag.ToString(), fs.ReadUShort());
                }

                this.FeatureRecords = new FeatureTable[featureCount];

                for (int i = 0; i < featureCount; i++)
                {
                    fs.Seek(startPosition + featureRecords[i].Item2, SeekOrigin.Begin);

                    this.FeatureRecords[i] = new FeatureTable(featureRecords[i].Item1, fs);
                }
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    int offset = 2 + 6 * this.FeatureRecords.Length;

                    ms.WriteUShort((ushort)this.FeatureRecords.Length);

                    byte[][] featureRecords = new byte[this.FeatureRecords.Length][];
                    int[] offsets = new int[this.FeatureRecords.Length];

                    for (int i = 0; i < this.FeatureRecords.Length; i++)
                    {
                        offsets[i] = offset;
                        featureRecords[i] = this.FeatureRecords[i].GetBytes();
                        offset += featureRecords[i].Length;
                    }

                    for (int i = 0; i < this.FeatureRecords.Length; i++)
                    {
                        ms.WriteByte((byte)this.FeatureRecords[i].Tag[0]);
                        ms.WriteByte((byte)this.FeatureRecords[i].Tag[1]);
                        ms.WriteByte((byte)this.FeatureRecords[i].Tag[2]);
                        ms.WriteByte((byte)this.FeatureRecords[i].Tag[3]);

                        ms.WriteUShort((ushort)offsets[i]);
                    }

                    for (int i = 0; i < this.FeatureRecords.Length; i++)
                    {
                        ms.Write(featureRecords[i], 0, featureRecords[i].Length);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class ValueRecord
        {
            public short XPlacement;
            public short YPlacement;
            public short XAdvance;
            public short YAdvance;
            public ushort ValueFormat;

            public ValueRecord(Stream fs, ushort valueFormat)
            {
                this.ValueFormat = valueFormat;

                if ((valueFormat & 0x0001) != 0)
                {
                    this.XPlacement = fs.ReadShort();
                }

                if ((valueFormat & 0x0002) != 0)
```

```
                {
                    this.YPlacement = fs.ReadShort();
                }

                if ((valueFormat & 0x0004) != 0)
                {
                    this.XAdvance = fs.ReadShort();
                }

                if ((valueFormat & 0x0008) != 0)
                {
                    this.YAdvance = fs.ReadShort();
                }

                if ((valueFormat & 0x0010) != 0)
                {
                    fs.ReadUShort();
                }

                if ((valueFormat & 0x0020) != 0)
                {
                    fs.ReadUShort();
                }

                if ((valueFormat & 0x0040) != 0)
                {
                    fs.ReadUShort();
                }

                if ((valueFormat & 0x0080) != 0)
                {
                    fs.ReadUShort();
                }
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    if ((this.ValueFormat & 0x0001) != 0)
                    {
                        ms.WriteShort(this.XPlacement);
                    }

                    if ((this.ValueFormat & 0x0002) != 0)
                    {
                        ms.WriteShort(this.YPlacement);
                    }

                    if ((this.ValueFormat & 0x0004) != 0)
                    {
                        ms.WriteShort(this.XAdvance);
                    }

                    if ((this.ValueFormat & 0x0008) != 0)
                    {
                        ms.WriteShort(this.YAdvance);
                    }

                    if ((this.ValueFormat & 0x0010) != 0)
                    {
                        ms.WriteUShort(0);
                    }

                    if ((this.ValueFormat & 0x0020) != 0)
                    {
                        ms.WriteUShort(0);
                    }

                    if ((this.ValueFormat & 0x0040) != 0)
                    {
                        ms.WriteUShort(0);
                    }

                    if ((this.ValueFormat & 0x0080) != 0)
                    {
                        ms.WriteUShort(0);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class SinglePosSubtable : ITrueTypeTable
        {
            public ushort PosFormat;
```

```
            public CoverageTable CoverageTable;
            public ushort ValueFormat;
            public ValueRecord ValueRecord;
            public ValueRecord[] ValueRecords;

            public SinglePosSubtable(Stream fs)
            {
                long startPosition = fs.Position;

                this.PosFormat = fs.ReadUShort();

                if (this.PosFormat == 1)
                {
                    ushort coverageOffset = fs.ReadUShort();
                    this.ValueFormat = fs.ReadUShort();
                    this.ValueRecord = new ValueRecord(fs, this.ValueFormat);

                    fs.Seek(startPosition + coverageOffset, SeekOrigin.Begin);
                    this.CoverageTable = new CoverageTable(fs);
                }
                else if (this.PosFormat == 2)
                {
                    ushort coverageOffset = fs.ReadUShort();
                    this.ValueFormat = fs.ReadUShort();

                    ushort valueCount = fs.ReadUShort();

                    this.ValueRecords = new ValueRecord[valueCount];

                    for (int i = 0; i < valueCount; i++)
                    {
                        this.ValueRecords[i] = new ValueRecord(fs, this.ValueFormat);
                    }

                    fs.Seek(startPosition + coverageOffset, SeekOrigin.Begin);
                    this.CoverageTable = new CoverageTable(fs);
                }
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort(this.PosFormat);

                    if (this.PosFormat == 1)
                    {
                        int offset = 22;

                        ms.WriteUShort((ushort)offset);
                        ms.WriteUShort(this.ValueFormat);

                        byte[] valueBytes = this.ValueRecord.GetBytes();

                        ms.Write(valueBytes, 0, valueBytes.Length);

                        byte[] coverageBytes = this.CoverageTable.GetBytes();

                        ms.Write(coverageBytes, 0, coverageBytes.Length);
                    }
                    else if (this.PosFormat == 2)
                    {
                        int offset = 8 + 16 * ValueRecords.Length;

                        ms.WriteUShort((ushort)offset);
                        ms.WriteUShort(this.ValueFormat);
                        ms.WriteUShort((ushort)this.ValueRecords.Length);

                        for (int i = 0; i < this.ValueRecords.Length; i++)
                        {
                            byte[] valueBytes = this.ValueRecords[i].GetBytes();

                            ms.Write(valueBytes, 0, valueBytes.Length);
                        }

                        byte[] coverageBytes = this.CoverageTable.GetBytes();

                        ms.Write(coverageBytes, 0, coverageBytes.Length);
                    }

                    return ms.ToArray();
                }
            }
        }

        internal class PairValueRecord
        {
```

```
            public ushort SecondGlyph;
            public ValueRecord ValueRecord1;
            public ValueRecord ValueRecord2;

            public PairValueRecord(Stream fs, ushort valueFormat1, ushort valueFormat2)
            {
                this.SecondGlyph = fs.ReadUShort();
                this.ValueRecord1 = new ValueRecord(fs, valueFormat1);
                this.ValueRecord2 = new ValueRecord(fs, valueFormat2);
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort(SecondGlyph);

                    byte[] bytes = this.ValueRecord1.GetBytes();
                    ms.Write(bytes, 0, bytes.Length);

                    bytes = this.ValueRecord2.GetBytes();
                    ms.Write(bytes, 0, bytes.Length);

                    return ms.ToArray();
                }
            }
        }

        internal class Class2Record
        {
            public ValueRecord ValueRecord1;
            public ValueRecord ValueRecord2;

            public Class2Record(Stream fs, ushort valueFormat1, ushort valueFormat2)
            {
                this.ValueRecord1 = new ValueRecord(fs, valueFormat1);
                this.ValueRecord2 = new ValueRecord(fs, valueFormat2);
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    byte[] bytes = this.ValueRecord1.GetBytes();
                    ms.Write(bytes, 0, bytes.Length);

                    bytes = this.ValueRecord2.GetBytes();
                    ms.Write(bytes, 0, bytes.Length);

                    return ms.ToArray();
                }
            }
        }

        /// <summary>
        /// Contains information describing how the position of two glyphs in a kerning pair should be altered.
        /// </summary>
        public class PairKerning
        {
            /// <summary>
            /// This vector contains the displacement that should be applied to the first glyph of the pair.
            /// </summary>
            public Point Glyph1Placement { get; }

            /// <summary>
            /// This vector describes how the advance width of the first glyph should be altered.
            /// </summary>
            public Point Glyph1Advance { get; }

            /// <summary>
            /// This vector contains the displacement that should be applied to the second glyph of the pair.
            /// </summary>
            public Point Glyph2Placement { get; }

            /// <summary>
            /// This vector describes how the advance width of the second glyph should be altered.
            /// </summary>
            public Point Glyph2Advance { get; }

            internal PairKerning(Point glyph1Placement, Point glyph1Advance, Point glyph2Placement, Point glyph2Advance)
            {
                this.Glyph1Placement = glyph1Placement;
                this.Glyph1Advance = glyph1Advance;
```

```
                this.Glyph2Placement = glyph2Placement;
                this.Glyph2Advance = glyph2Advance;
            }
        }

        internal class PairPosSubtable : ITrueTypeTable
        {
            public ushort PosFormat;
            public CoverageTable CoverageTable;

            public ushort ValueFormat1;
            public ushort ValueFormat2;

            public PairValueRecord[][] PairSets;

            public ClassDefinitionTable ClassDef1;
            public ClassDefinitionTable ClassDef2;
            public Class2Record[][] Class1Records;

            public PairPosSubtable(Stream fs)
            {
                long startPosition = fs.Position;

                this.PosFormat = fs.ReadUShort();

                if (this.PosFormat == 1)
                {
                    ushort coverageOffset = fs.ReadUShort();

                    this.ValueFormat1 = fs.ReadUShort();
                    this.ValueFormat2 = fs.ReadUShort();

                    ushort pairSetCount = fs.ReadUShort();

                    ushort[] pairSetOffsets = new ushort[pairSetCount];

                    for (int i = 0; i < pairSetCount; i++)
                    {
                        pairSetOffsets[i] = fs.ReadUShort();
                    }

                    fs.Seek(startPosition + coverageOffset, SeekOrigin.Begin);
                    this.CoverageTable = new CoverageTable(fs);

                    this.PairSets = new PairValueRecord[pairSetCount][];

                    for (int i = 0; i < pairSetCount; i++)
                    {
                        fs.Seek(startPosition + pairSetOffsets[i], SeekOrigin.Begin);

                        ushort pairValueCount = fs.ReadUShort();

                        this.PairSets[i] = new PairValueRecord[pairValueCount];

                        for (int j = 0; j < pairValueCount; j++)
                        {
                            this.PairSets[i][j] = new PairValueRecord(fs, this.ValueFormat1, this.ValueFormat2);
                        }
                    }
                }
                else if (this.PosFormat == 2)
                {
                    ushort coverageOffset = fs.ReadUShort();

                    this.ValueFormat1 = fs.ReadUShort();
                    this.ValueFormat2 = fs.ReadUShort();

                    ushort class1DefOffset = fs.ReadUShort();
                    ushort class2DefOffset = fs.ReadUShort();

                    ushort class1Count = fs.ReadUShort();
                    ushort class2Count = fs.ReadUShort();

                    this.Class1Records = new Class2Record[class1Count][];

                    for (int i = 0; i < class1Count; i++)
                    {
                        this.Class1Records[i] = new Class2Record[class2Count];

                        for (int j = 0; j < class2Count; j++)
                        {
                            this.Class1Records[i][j] = new Class2Record(fs, this.ValueFormat1, this.ValueFormat2);
                        }
                    }

```

```
                    fs.Seek(startPosition + coverageOffset, SeekOrigin.Begin);
                    this.CoverageTable = new CoverageTable(fs);

                    fs.Seek(startPosition + class1DefOffset, SeekOrigin.Begin);
                    this.ClassDef1 = new ClassDefinitionTable(fs);

                    fs.Seek(startPosition + class2DefOffset, SeekOrigin.Begin);

                    this.ClassDef2 = new ClassDefinitionTable(fs);
                }
            }

            public PairKerning GetKerning(int glyph1CoverageIndex, int glyph1Index, int glyph2Index)
            {
                if (this.PosFormat == 1)
                {
                    for (int i = 0; i < this.PairSets[glyph1CoverageIndex].Length; i++)
                    {
                        if (this.PairSets[glyph1CoverageIndex][i].SecondGlyph == glyph2Index)
                        {
                            return new PairKerning(new Point(this.PairSets[glyph1CoverageIndex][i].ValueRecord1.XPlacement, this.PairSets[glyph1CoverageIndex][i].ValueRecord1.YPlacement),
                                                   new Point(this.PairSets[glyph1CoverageIndex][i].ValueRecord1.XAdvance, this.PairSets[glyph1CoverageIndex][i].ValueRecord1.YAdvance),
                                                   new Point(this.PairSets[glyph1CoverageIndex][i].ValueRecord2.XPlacement, this.PairSets[glyph1CoverageIndex][i].ValueRecord2.YPlacement),
                                                   new Point(this.PairSets[glyph1CoverageIndex][i].ValueRecord2.XAdvance, this.PairSets[glyph1CoverageIndex][i].ValueRecord2.YAdvance));
                        }
                    }

                    return null;
                }
                else if (this.PosFormat == 2)
                {
                    int glyph1Class = this.ClassDef1.GetClass(glyph1Index);
                    int glyph2Class = this.ClassDef2.GetClass(glyph2Index);

                    if (glyph1Class < 0 || glyph2Class < 0)
                    {
                        return null;
                    }

                    return new PairKerning(new Point(this.Class1Records[glyph1Class][glyph2Class].ValueRecord1.XPlacement, this.Class1Records[glyph1Class][glyph2Class].ValueRecord1.YPlacement),
                                           new Point(this.Class1Records[glyph1Class][glyph2Class].ValueRecord1.XAdvance, this.Class1Records[glyph1Class][glyph2Class].ValueRecord1.YAdvance),
                                           new Point(this.Class1Records[glyph1Class][glyph2Class].ValueRecord2.XPlacement, this.Class1Records[glyph1Class][glyph2Class].ValueRecord2.YPlacement),
                                           new Point(this.Class1Records[glyph1Class][glyph2Class].ValueRecord2.XAdvance, this.Class1Records[glyph1Class][glyph2Class].ValueRecord2.YAdvance));
                }
                else
                {
                    return null;
                }
            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort(this.PosFormat);

                    if (this.PosFormat == 1)
                    {
                        int offset = 10 + 2 * PairSets.Length;

                        ms.WriteUShort((ushort)offset);

                        ms.WriteUShort((ushort)ValueFormat1);
                        ms.WriteUShort((ushort)ValueFormat2);

                        ms.WriteUShort((ushort)this.PairSets.Length);

                        byte[] coverageBytes = this.CoverageTable.GetBytes();

                        offset += coverageBytes.Length;

                        byte[][][] pairSets = new byte[this.PairSets.Length][][];
                        int[] offsets = new int[this.PairSets.Length];
```

```
                    for (int i = 0; i < this.PairSets.Length; i++)
                    {
                        offsets[i] = offset;

                        pairSets[i] = new byte[this.PairSets[i].Length][];

                        for (int j = 0; j < this.PairSets[i].Length; j++)
                        {
                            pairSets[i][j] = this.PairSets[i][j].GetBytes();
                            offset += pairSets[i][j].Length;
                        }
                    }

                    for (int i = 0; i < offsets.Length; i++)
                    {
                        ms.WriteUShort((ushort)offsets[i]);
                    }

                    ms.Write(coverageBytes, 0, coverageBytes.Length);

                    for (int i = 0; i < pairSets.Length; i++)
                    {
                        for (int j = 0; j < pairSets[i].Length; j++)
                        {
                            ms.Write(pairSets[i][j], 0, pairSets[i][j].Length);
                        }
                    }

                    return ms.ToArray();
                }
                else if (this.PosFormat == 2)
                {
                    int offset = 16;

                    byte[][][] class1Records = new byte[this.Class1Records.Length][][];

                    for (int i = 0; i < class1Records.Length; i++)
                    {
                        class1Records[i] = new byte[this.Class1Records[i].Length][];

                        for (int j = 0; j < class1Records[i].Length; j++)
                        {
                            class1Records[i][j] = this.Class1Records[i][j].GetBytes();
                            offset += class1Records[i][j].Length;
                        }
                    }

                    ms.WriteUShort((ushort)offset);

                    ms.WriteUShort(this.ValueFormat1);
                    ms.WriteUShort(this.ValueFormat2);

                    byte[] coverageBytes = this.CoverageTable.GetBytes();

                    offset += coverageBytes.Length;

                    ms.WriteUShort((ushort)offset);

                    byte[] classDef1Bytes = this.ClassDef1.GetBytes();

                    offset += classDef1Bytes.Length;

                    ms.WriteUShort((ushort)offset);

                    ms.WriteUShort((ushort)this.Class1Records.Length);
                    ms.WriteUShort((ushort)this.Class1Records[0].Length);

                    for (int i = 0; i < class1Records.Length; i++)
                    {
                        for (int j = 0; j < class1Records[i].Length; j++)
                        {
                            ms.Write(class1Records[i][j], 0, class1Records[i][j].Length);
                        }
                    }

                    ms.Write(coverageBytes, 0, coverageBytes.Length);
                    ms.Write(classDef1Bytes, 0, classDef1Bytes.Length);

                    byte[] classDef2Bytes = this.ClassDef2.GetBytes();
                    ms.Write(classDef2Bytes, 0, classDef2Bytes.Length);

                    return ms.ToArray();
                }
                else
                {
                    return ms.ToArray();
```

```
                    }
                }
            }

        }

        internal class TrueTypeGPOSTable : ITrueTypeTable
        {
            public ushort MajorVersion;
            public ushort MinorVersion;

            public ScriptList ScriptList;
            public FeatureList FeatureList;
            public LookupList LookupList;

            private HashSet<int> KernLookups;

            public TrueTypeGPOSTable(Stream fs)
            {
                long startPosition = fs.Position;

                this.MajorVersion = fs.ReadUShort();
                this.MinorVersion = fs.ReadUShort();

                ushort scriptListOffset = fs.ReadUShort();
                ushort featureListOffset = fs.ReadUShort();
                ushort lookupListOffset = fs.ReadUShort();

                if (this.MinorVersion == 1)
                {
                    fs.ReadUInt();
                }

                fs.Seek(startPosition + scriptListOffset, SeekOrigin.Begin);
                this.ScriptList = new ScriptList(fs);

                fs.Seek(startPosition + featureListOffset, SeekOrigin.Begin);
                this.FeatureList = new FeatureList(fs);

                fs.Seek(startPosition + lookupListOffset, SeekOrigin.Begin);
                this.LookupList = new LookupList(fs);

                this.KernLookups = new HashSet<int>();

                for (int i = 0; i < this.FeatureList.FeatureRecords.Length; i++)
                {
                    if (this.FeatureList.FeatureRecords[i].Tag == "kern")
                    {
                        for (int j = 0; j < this.FeatureList.FeatureRecords[i].LookupListIndices.Length; j++)
                        {
                            if (this.LookupList.LookupTables[this.FeatureList.FeatureRecords[i].LookupListIndices[j]].LookupType == 2)
                            {
                                this.KernLookups.Add(this.FeatureList.FeatureRecords[i].LookupListIndices[j]);
                            }
                        }
                    }
                }

            }

            public byte[] GetBytes()
            {
                using (MemoryStream ms = new MemoryStream())
                {
                    ms.WriteUShort(this.MajorVersion);
                    ms.WriteUShort(this.MinorVersion);

                    int offset = 10;

                    if (this.MinorVersion == 1)
                    {
                        offset += 2;
                    }

                    ms.WriteUShort((ushort)offset);

                    byte[] scriptList = this.ScriptList.GetBytes();
                    offset += scriptList.Length;

                    ms.WriteUShort((ushort)offset);

                    byte[] featureList = this.FeatureList.GetBytes();
                    offset += featureList.Length;
```

```

                    ms.WriteUShort((ushort)offset);

                    if (this.MinorVersion == 1)
                    {
                        ms.WriteUInt(0);
                    }

                    ms.Write(scriptList, 0, scriptList.Length);
                    ms.Write(featureList, 0, featureList.Length);

                    byte[] lookupList = this.LookupList.GetBytes();

                    ms.Write(lookupList, 0, lookupList.Length);

                    return ms.ToArray();
                }
            }


            public PairKerning GetKerning(int glyph1Index, int glyph2Index)
            {
                foreach (int index in this.KernLookups)
                {
                    for (int i = 0; i < this.LookupList.LookupTables[index].SubTables.Length; i++)
                    {
                        if (this.LookupList.LookupTables[index].SubTables[i] is PairPosSubtable pairPos)
                        {
                            int coverageIndex = pairPos.CoverageTable.ContainsGlyph(glyph1Index);

                            if (coverageIndex >= 0)
                            {
                                return pairPos.GetKerning(coverageIndex, glyph1Index, glyph2Index);
                            }
                        }
                    }
                }

                return null;
            }
        }

        internal struct TrueTypeTableOffset
        {
            public uint Checksum;
            public uint Offset;
            public uint Length;

            public TrueTypeTableOffset(uint checksum, uint offset, uint length)
            {
                this.Checksum = checksum;
                this.Offset = offset;
                this.Length = length;
            }
        }

        internal struct LongHorFixed
        {
            public ushort AdvanceWidth;
            public short LeftSideBearing;

            public LongHorFixed(ushort advanceWidth, short leftSideBearing)
            {
                this.AdvanceWidth = advanceWidth;
                this.LeftSideBearing = leftSideBearing;
            }
        }

        internal struct Fixed
        {
            public int Bits;
            public int BitShifts;

            public Fixed(int bits, int bitShifts)
            {
                this.Bits = bits;
                this.BitShifts = bitShifts;
            }
        }
    }

    internal static class ReadUtils
    {
        public static void WritePascalString(this Stream sr, string val)
        {
            sr.WriteByte((byte)val.Length);
```

```
            for (int i = 0; i < val.Length; i++)
            {
                sr.WriteByte((byte)(int)val[i]);
            }
        }

        public static string ReadPascalString(this Stream sr)
        {
            byte length = (byte)sr.ReadByte();
            StringBuilder tbr = new StringBuilder(length);
            for (int i = 0; i < length; i++)
            {
                tbr.Append((char)sr.ReadByte());
            }
            return tbr.ToString();
        }

        public static uint ReadUInt(this Stream sr)
        {
            return ((uint)sr.ReadByte() << 24) | ((uint)sr.ReadByte() << 16) | ((uint)sr.ReadByte() << 8) | (uint)sr.ReadByte();
        }

        public static void WriteUInt(this Stream sr, uint val)
        {
            sr.Write(new byte[] { (byte)(val >> 24), (byte)((val >> 16) & 255), (byte)((val >> 8) & 255), (byte)(val & 255) }, 0, 4);
        }

        public static int ReadInt(this Stream sr)
        {
            return (sr.ReadByte() << 24) | (sr.ReadByte() << 16) | (sr.ReadByte() << 8) | sr.ReadByte();
        }

        public static void WriteInt(this Stream sr, int val)
        {
            sr.WriteUInt((uint)val);
        }

        public static ushort ReadUShort(this Stream sr)
        {
            return (ushort)(((uint)sr.ReadByte() << 8) | (uint)sr.ReadByte());
        }

        public static void WriteUShort(this Stream sr, ushort val)
        {
            sr.Write(new byte[] { (byte)(val >> 8), (byte)(val & 255) }, 0, 2);
        }

        public static short ReadShort(this Stream sr)
        {
            ushort result = sr.ReadUShort();
            short tbr;
            if ((result & 0x8000) != 0)
            {
                tbr = (short)(result - (1 << 16));
            }
            else
            {
                tbr = (short)result;
            }
            return tbr;
        }

        public static void WriteShort(this Stream sr, short val)
        {
            sr.WriteUShort((ushort)val);
        }

        public static string ReadString(this Stream sr, int length)
        {
            StringBuilder bld = new StringBuilder(length);
            for (int i = 0; i < length; i++)
            {
                bld.Append((char)sr.ReadByte());
            }
            return bld.ToString();
        }

        public static TrueTypeFile.Fixed ReadFixed(this Stream sr)
        {
            return new TrueTypeFile.Fixed(sr.ReadInt(), 16);
        }

        public static void WriteFixed(this Stream sr, TrueTypeFile.Fixed val)
        {
            sr.WriteInt(val.Bits);
```

```
        }

        public static DateTime ReadDate(this Stream sr)
        {
            long macTime = sr.ReadUInt() * 0x100000000 + sr.ReadUInt();
            return new DateTime(1904, 1, 1).AddTicks(macTime * 1000 * TimeSpan.TicksPerMillisecond);
        }

        public static void WriteDate(this Stream sr, DateTime date)
        {
            long macTime = date.Subtract(new DateTime(1904, 1, 1)).Ticks / 1000 /
       TimeSpan.TicksPerMillisecond;
            sr.WriteUInt((uint)(macTime » 32));
            sr.WriteUInt((uint)(macTime & 0xFFFFFFFF));
        }
    }

}
```

Index

A
VectSharp.Colour, 71
VectSharp.SolidColourBrush, 433
a
VectSharp.Colour, 71
A1
VectSharp.Filters.ColourMatrix, 84
A2
VectSharp.Filters.ColourMatrix, 84
A3
VectSharp.Filters.ColourMatrix, 85
A4
VectSharp.Filters.ColourMatrix, 85
A5
VectSharp.Filters.ColourMatrix, 85
ActionType
VectSharp.Canvas.RenderAction, 381
VectSharp.Canvas.SKRenderAction, 417
ActionTypes
VectSharp.Canvas.RenderAction, 378
VectSharp.Canvas.SKRenderAction, 412
Add
VectSharp.SimpleFontLibrary, 397, 398
AddElement
VectSharp.ThreeD.IScene, 273
VectSharp.ThreeD.Scene, 387
AddFrame
VectSharp.Animation, 40
AddLayer
VectSharp.Canvas.SKMultiLayerRenderCanvas, 404
AddPath
VectSharp.GraphicsPath, 225
AddRange
VectSharp.ThreeD.IScene, 273
VectSharp.ThreeD.Scene, 388
AddSmoothSpline
VectSharp.GraphicsPath, 225
AddText
VectSharp.GraphicsPath, 226
AddTextOnPath
VectSharp.GraphicsPath, 227
AddTextUnderline
VectSharp.GraphicsPath, 227
AdvanceWidth
VectSharp.Font.DetailedFontMetrics, 140
AliceBlue
VectSharp.Colours, 98
AllowPageBreak
VectSharp.Markdown.MarkdownRenderer, 297
AlphaInversion
VectSharp.Filters.ColourMatrix, 83
AmbientLightSource
VectSharp.ThreeD.AmbientLightSource, 31
AmbientReflectionCoefficient
VectSharp.ThreeD.PhongMaterial, 341
AngleAttenuationExponent
VectSharp.ThreeD.MaskedLightSource, 313
VectSharp.ThreeD.SpotlightLightSource, 439
Animation
VectSharp.Animation, 39
AntiqueWhite
VectSharp.Colours, 98
Apply
VectSharp.Filters.ColourMatrix, 78–80
Aqua
VectSharp.Colours, 98
Aquamarine
VectSharp.Colours, 98
Arc
VectSharp.GraphicsPath, 228
AreaLightSource
VectSharp.ThreeD.AreaLightSource, 45
Ascent
VectSharp.Font, 159
Azure
VectSharp.Colours, 98

B
VectSharp.Colour, 71
VectSharp.SolidColourBrush, 433
B1
VectSharp.Filters.ColourMatrix, 85
B2
VectSharp.Filters.ColourMatrix, 85
B3
VectSharp.Filters.ColourMatrix, 86
B4
VectSharp.Filters.ColourMatrix, 86
B5
VectSharp.Filters.ColourMatrix, 86
Background
VectSharp.Animation, 41
VectSharp.Page, 328
BackgroundColour
VectSharp.Markdown.MarkdownRenderer, 297
BaseFontSize
VectSharp.Markdown.MarkdownRenderer, 297
BaseImageUri

VectSharp.Markdown.MarkdownRenderer, 297
BaseLinkUri
VectSharp.Markdown.MarkdownRenderer, 298
BeamWidthAngle
VectSharp.ThreeD.SpotlightLightSource, 440
Beige
VectSharp.Colours, 99
Bias
VectSharp.Filters.ConvolutionFilter, 136
Bisque
VectSharp.Colours, 99
Black
VectSharp.Colours, 99
BlanchedAlmond
VectSharp.Colours, 99
Blue
VectSharp.Colours, 99
BlueViolet
VectSharp.Colours, 100
BoldFontFamily
VectSharp.Markdown.MarkdownRenderer, 298
BoldItalicFontFamily
VectSharp.Markdown.MarkdownRenderer, 298
BoldUnderlineThickness
VectSharp.Markdown.MarkdownRenderer, 298
Bottom
VectSharp.Font.DetailedFontMetrics, 140
VectSharp.Markdown.Margins, 285
BottomRightMargin
VectSharp.Filters.BoxBlurFilter, 56
VectSharp.Filters.ColourMatrixFilter, 91
VectSharp.Filters.CompositeLocationInvariantFilter, 130
VectSharp.Filters.ConvolutionFilter, 136
VectSharp.Filters.GaussianBlurFilter, 185
VectSharp.Filters.IFilter, 243
VectSharp.Filters.MaskFilter, 317
Bounds
VectSharp.Point, 343
BoxBlurFilter
VectSharp.Filters.BoxBlurFilter, 55
BoxRadius
VectSharp.Filters.BoxBlurFilter, 56
BringToFront
VectSharp.Canvas.RenderAction, 378
Brown
VectSharp.Colours, 100
Brush
VectSharp.FormattedText, 180
Bullets
VectSharp.Markdown.MarkdownRenderer, 298
BurlyWood
VectSharp.Colours, 100

CadetBlue
VectSharp.Colours, 100
CastsShadow
VectSharp.ThreeD.AmbientLightSource, 32
VectSharp.ThreeD.AreaLightSource, 46
VectSharp.ThreeD.ILightSource, 260
VectSharp.ThreeD.MaskedLightSource, 314
VectSharp.ThreeD.ParallelLightSource, 333
VectSharp.ThreeD.PointLightSource, 349
VectSharp.ThreeD.SpotlightLightSource, 440
Center
VectSharp.ThreeD.AreaLightSource, 46
Centre
VectSharp.RadialGradientBrush, 352
VectSharp.Rectangle, 376
Chartreuse
VectSharp.Colours, 100
Chocolate
VectSharp.Colours, 101
Class
VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord, 59
ClassRangeRecord
VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord, 59
ClearPNGCache
VectSharp.RasterImage, 361
ClipAction
VectSharp.Canvas.SKRenderAction, 412
ClippingPath
VectSharp.Canvas.RenderAction, 381
Clone
VectSharp.Segment, 390
Close
VectSharp.GraphicsPath, 229
CodeBlockBackgroundColour
VectSharp.Markdown.MarkdownRenderer, 299
CodeFont
VectSharp.Markdown.MarkdownRenderer, 299
CodeFontBold
VectSharp.Markdown.MarkdownRenderer, 299
CodeFontBoldItalic
VectSharp.Markdown.MarkdownRenderer, 299
CodeFontItalic
VectSharp.Markdown.MarkdownRenderer, 300
CodeInlineBackgroundColour
VectSharp.Markdown.MarkdownRenderer, 300
CodeInlineMargin
VectSharp.Markdown.MarkdownRenderer, 300
Colour
VectSharp.GradientStop, 188
VectSharp.Markdown.FormattedString, 176
VectSharp.SolidColourBrush, 433
VectSharp.ThreeD.ColourMaterial, 75
VectSharp.ThreeD.PhongMaterial, 341
ColourMaterial
VectSharp.ThreeD.ColourMaterial, 74
ColourMatrix
VectSharp.Filters.ColourMatrix, 78
VectSharp.Filters.ColourMatrixFilter, 91
ColourMatrixFilter
VectSharp.Filters.ColourMatrixFilter, 90
CompositeLocationInvariantFilter

VectSharp.Filters.CompositeLocationInvariantFilter, 129
CompressedFrame
VectSharp.AnimatedPNG.CompressedFrame, 132
ControlPoint1
VectSharp.SplineEasing, 436
ControlPoint2
VectSharp.SplineEasing, 436
ConvolutionFilter
VectSharp.Filters.ConvolutionFilter, 135
CopyToIGraphicsContext
VectSharp.Graphics, 195
CopyToSKRenderContext
VectSharp.Canvas.SKRenderContextInterpreter, 423–425
Coral
VectSharp.Colours, 101
CornflowerBlue
VectSharp.Colours, 101
Cornsilk
VectSharp.Colours, 101
Count
VectSharp.GradientStops, 191
Create
VectSharp.AnimatedPNG, 38
CreateCube
VectSharp.ThreeD.ObjectFactory, 319
CreateCuboid
VectSharp.ThreeD.ObjectFactory, 320
CreatePoints
VectSharp.ThreeD.ObjectFactory, 321
CreatePolygon
VectSharp.ThreeD.ObjectFactory, 321
CreatePrism
VectSharp.ThreeD.ObjectFactory, 322
CreateRectangle
VectSharp.ThreeD.ObjectFactory, 323
CreateSphere
VectSharp.ThreeD.ObjectFactory, 324
CreateTetrahedron
VectSharp.ThreeD.ObjectFactory, 324
CreateWireframe
VectSharp.ThreeD.ObjectFactory, 325
Crimson
VectSharp.Colours, 101
Crop
VectSharp.Graphics, 196
VectSharp.Page, 327, 328
CubicBezierTo
VectSharp.GraphicsPath, 229
VectSharp.IGraphicsContext, 250
CurrentFrame
VectSharp.Canvas.AnimatedCanvas, 36
CurrentFrameProperty
VectSharp.Canvas.AnimatedCanvas, 34
CutoffAngle
VectSharp.ThreeD.SpotlightLightSource, 440
Cyan
VectSharp.Colours, 102
DarkBlue
VectSharp.Colours, 102
DarkCyan
VectSharp.Colours, 102
DarkGoldenRod
VectSharp.Colours, 102
DarkGray
VectSharp.Colours, 102
DarkGreen
VectSharp.Colours, 103
DarkGrey
VectSharp.Colours, 103
DarkKhaki
VectSharp.Colours, 103
DarkMagenta
VectSharp.Colours, 103
DarkOliveGreen
VectSharp.Colours, 103
DarkOrange
VectSharp.Colours, 104
DarkOrchid
VectSharp.Colours, 104
DarkRed
VectSharp.Colours, 104
DarkSalmon
VectSharp.Colours, 104
DarkSeaGreen
VectSharp.Colours, 104
DarkSlateBlue
VectSharp.Colours, 105
DarkSlateGray
VectSharp.Colours, 105
DarkSlateGrey
VectSharp.Colours, 105
DarkTurquoise
VectSharp.Colours, 105
DarkViolet
VectSharp.Colours, 105
DataHolder
VectSharp.RasterImage, 362
Deconstruct
VectSharp.ThreeD.LightIntensity, 276
DeepPink
VectSharp.Colours, 106
DeepSkyBlue
VectSharp.Colours, 106
Default
VectSharp.Canvas.FilterOption, 146
VectSharp.PDF.PDFContextInterpreter.FilterOption, 149
VectSharp.SVG.SVGContextInterpreter.FilterOption, 151
DefaultFontLibrary
VectSharp.FontFamily, 166
Descent
VectSharp.Font, 159
Destroy

VectSharp.TrueTypeFile, 453
DiffuseReflectionCoefficient
VectSharp.ThreeD.PhongMaterial, 341
DimGray
VectSharp.Colours, 106
DimGrey
VectSharp.Colours, 106
Direction
VectSharp.ThreeD.AreaLightSource, 47
VectSharp.ThreeD.LightIntensity, 276
VectSharp.ThreeD.MaskedLightSource, 314
VectSharp.ThreeD.ParallelLightSource, 333
VectSharp.ThreeD.SpotlightLightSource, 440
Discretise
VectSharp.GraphicsPath, 230
DisposableIntPtr
VectSharp.DisposableIntPtr, 143
Dispose
VectSharp.AnimatedPNG.CompressedFrame, 133
VectSharp.Canvas.SKMultiLayerRenderCanvas, 404
VectSharp.Canvas.SKRenderAction, 412
VectSharp.DisposableIntPtr, 143
VectSharp.Filters.FilterWithRasterisableParameter, 153
VectSharp.RasterImage, 361
Disposed
VectSharp.Canvas.SKRenderAction, 417
Distance
VectSharp.ThreeD.MaskedLightSource, 314
DistanceAttenuationExponent
VectSharp.ThreeD.AreaLightSource, 47
VectSharp.ThreeD.MaskedLightSource, 314
VectSharp.ThreeD.PointLightSource, 349
VectSharp.ThreeD.SpotlightLightSource, 440
Document
VectSharp.Document, 144
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 289
DocumentProperty
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 287
DocumentSource
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 289
DocumentSourceProperty
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 288
DodgerBlue
VectSharp.Colours, 106
DrawFilteredGraphics
VectSharp.IGraphicsContext, 251
DrawFilteredGraphicsAction
VectSharp.Canvas.SKRenderAction, 412
DrawGraphics
VectSharp.Graphics, 196–198
DrawRasterImage
VectSharp.Graphics, 199–201
VectSharp.IGraphicsContext, 251
Duration
VectSharp.AnimatedPNG.CompressedFrame, 133
VectSharp.Animation, 41
VectSharp.Frame, 183
VectSharp.Transition, 450

Ease
VectSharp.IEasing, 241
VectSharp.LinearEasing, 278
VectSharp.SplineEasing, 435
Easings
VectSharp.Transition, 450
EllipticalArc
VectSharp.GraphicsPath, 230
EnableKerning
VectSharp.Font, 159
EndGlyphID
VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord, 59
VectSharp.TrueTypeFile.CoverageTable.RangeRecord, 354
EndPoint
VectSharp.LinearGradientBrush, 281
Equals
VectSharp.Colour, 62

FamilyName
VectSharp.FontFamily, 166
FileName
VectSharp.FontFamily, 167
Fill
VectSharp.Canvas.RenderAction, 381
FillPath
VectSharp.Graphics, 201
FillRectangle
VectSharp.Graphics, 202
FillStyle
VectSharp.IGraphicsContext, 256
FillText
VectSharp.Graphics, 203, 204
VectSharp.IGraphicsContext, 251
FillTextOnPath
VectSharp.Graphics, 204
FillTextUnderline
VectSharp.Graphics, 205–207
Filter
VectSharp.Canvas.SKRenderAction, 417
VectSharp.Filters.BoxBlurFilter, 55
VectSharp.Filters.ColourMatrixFilter, 90
VectSharp.Filters.CompositeLocationInvariantFilter, 129
VectSharp.Filters.ConvolutionFilter, 136
VectSharp.Filters.GaussianBlurFilter, 185
VectSharp.Filters.IFilterWithLocation, 244
VectSharp.Filters.ILocationInvariantFilter, 261
VectSharp.Filters.MaskFilter, 316
FilterOperations
VectSharp.Canvas.FilterOption, 145

VectSharp.PDF.PDFContextInterpreter.FilterOption, 148
VectSharp.SVG.SVGContextInterpreter.FilterOption, 151
FilterOption
VectSharp.Canvas.FilterOption, 146
VectSharp.PDF.PDFContextInterpreter.FilterOption, 148
VectSharp.SVG.SVGContextInterpreter.FilterOption, 151
Filters
VectSharp.Filters.CompositeLocationInvariantFilter, 130
FireBrick
VectSharp.Colours, 107
FloralWhite
VectSharp.Colours, 107
FocalPoint
VectSharp.RadialGradientBrush, 353
FollowItalicAngle
VectSharp.Font.FontUnderline, 174
Font
VectSharp.Canvas.SKRenderAction, 418
VectSharp.Font, 156
VectSharp.FormattedText, 180
VectSharp.IGraphicsContext, 256
FontFamily
VectSharp.Font, 159
VectSharp.FontFamily, 163
VectSharp.FontFamilyCreationException, 169
FontFamilyCreationException
VectSharp.FontFamilyCreationException, 169
FontSize
VectSharp.Font, 159
FontStream
VectSharp.TrueTypeFile, 464
ForegroundColour
VectSharp.Markdown.MarkdownRenderer, 300
ForestGreen
VectSharp.Colours, 107
Format
VectSharp.FormattedText, 178, 179
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.UnknownFormatException, 466
FormattedString
VectSharp.Markdown.FormattedString, 175
FormattedText
VectSharp.FormattedText, 178
Frame
VectSharp.Frame, 182
FrameCount
VectSharp.Canvas.AnimatedCanvas, 36
FrameCountProperty
VectSharp.Canvas.AnimatedCanvas, 35
FrameRate
VectSharp.Canvas.AnimatedCanvas, 36
FrameRateProperty
VectSharp.Canvas.AnimatedCanvas, 35
Frames
VectSharp.Animation, 42
FromCSSString
VectSharp.Colour, 62
FromFile
VectSharp.SVG.Parser, 334
FromHSL
VectSharp.Colour, 62
FromLab
VectSharp.Colour, 63
FromRgb
VectSharp.Colour, 63, 64
FromRgba
VectSharp.Colour, 65–67
FromStream
VectSharp.SVG.Parser, 335
FromString
VectSharp.SVG.Parser, 335
FromXYZ
VectSharp.Colour, 67
Fuchsia
VectSharp.Colours, 107

G
VectSharp.Colour, 71
VectSharp.SolidColourBrush, 434
G1
VectSharp.Filters.ColourMatrix, 86
G2
VectSharp.Filters.ColourMatrix, 86
G3
VectSharp.Filters.ColourMatrix, 87
G4
VectSharp.Filters.ColourMatrix, 87
G5
VectSharp.Filters.ColourMatrix, 87
Gainsboro
VectSharp.Colours, 107
GaussianBlurFilter
VectSharp.Filters.GaussianBlurFilter, 184
Geometry
VectSharp.Canvas.RenderAction, 381
Get1000EmAscent
VectSharp.TrueTypeFile, 453
Get1000EmDescent
VectSharp.TrueTypeFile, 454
Get1000EmGlyphBearings
VectSharp.TrueTypeFile, 454
Get1000EmGlyphVerticalMetrics
VectSharp.TrueTypeFile, 454
Get1000EmGlyphWidth
VectSharp.TrueTypeFile, 455
Get1000EmKerning
VectSharp.TrueTypeFile, 456
Get1000EmUnderlineIntersections
VectSharp.TrueTypeFile, 456
Get1000EmUnderlinePosition
VectSharp.TrueTypeFile, 457
Get1000EmUnderlineThickness

VectSharp.TrueTypeFile, 457
Get1000EmWinAscent
VectSharp.TrueTypeFile, 457
Get1000EmXMax
VectSharp.TrueTypeFile, 458
Get1000EmXMin
VectSharp.TrueTypeFile, 458
Get1000EmYMax
VectSharp.TrueTypeFile, 458
Get1000EmYMin
VectSharp.TrueTypeFile, 458
GetBounds
VectSharp.Graphics, 207
VectSharp.GraphicsPath, 231
GetColour
VectSharp.ThreeD.ColourMaterial, 74
VectSharp.ThreeD.IMaterial, 271
VectSharp.ThreeD.PhongMaterial, 340
GetColourAt
VectSharp.GradientStops, 190
GetEnumerator
VectSharp.GradientStops, 191
GetFigures
VectSharp.GraphicsPath, 231
GetFirstCharIndex
VectSharp.TrueTypeFile, 459
GetFontFamilyName
VectSharp.TrueTypeFile, 459
GetFontName
VectSharp.TrueTypeFile, 459
GetFrameAtAbsolute
VectSharp.Animation, 40
GetFrameAtRelative
VectSharp.Animation, 41
GetFullFontFamilyName
VectSharp.TrueTypeFile, 459
GetGlyphIndex
VectSharp.TrueTypeFile, 460
GetGlyphPath
VectSharp.TrueTypeFile, 460, 461
GetHashCode
VectSharp.Colour, 68
GetItalicAngle
VectSharp.TrueTypeFile, 461
GetLastCharIndex
VectSharp.TrueTypeFile, 461
GetLightAt
VectSharp.ThreeD.AmbientLightSource, 31
VectSharp.ThreeD.AreaLightSource, 45
VectSharp.ThreeD.ILightSource, 259
VectSharp.ThreeD.MaskedLightSource, 313
VectSharp.ThreeD.ParallelLightSource, 332
VectSharp.ThreeD.PointLightSource, 348
VectSharp.ThreeD.SpotlightLightSource, 439
GetLinearisationPointsNormals
VectSharp.GraphicsPath, 231
GetLinearisationTangents
VectSharp.Segment, 390
GetNormalAtAbsolute
VectSharp.GraphicsPath, 232
GetNormalAtRelative
VectSharp.GraphicsPath, 232
GetObstruction
VectSharp.ThreeD.AmbientLightSource, 32
VectSharp.ThreeD.AreaLightSource, 46
VectSharp.ThreeD.ILightSource, 259
VectSharp.ThreeD.MaskedLightSource, 313
VectSharp.ThreeD.ParallelLightSource, 332
VectSharp.ThreeD.PointLightSource, 348
VectSharp.ThreeD.SpotlightLightSource, 439
GetPointAt
VectSharp.Segment, 391
GetPointAtAbsolute
VectSharp.GraphicsPath, 232
GetPointAtRelative
VectSharp.GraphicsPath, 233
GetPoints
VectSharp.GraphicsPath, 233
GetSyntaxHighlightedLines
VectSharp.Markdown.SyntaxHighlighter, 449
GetTags
VectSharp.Graphics, 207
GetTangentAt
VectSharp.Segment, 391
GetTangentAtAbsolute
VectSharp.GraphicsPath, 233
GetTangentAtRelative
VectSharp.GraphicsPath, 234
GhostWhite
VectSharp.Colours, 108
Glyph1Advance
VectSharp.TrueTypeFile.PairKerning, 329
Glyph1Placement
VectSharp.TrueTypeFile.PairKerning, 329
Glyph2Advance
VectSharp.TrueTypeFile.PairKerning, 330
Glyph2Placement
VectSharp.TrueTypeFile.PairKerning, 330
Gold
VectSharp.Colours, 108
GoldenRod
VectSharp.Colours, 108
GradientStop
VectSharp.GradientStop, 187
GradientStops
VectSharp.GradientBrush, 187
VectSharp.GradientStops, 190
Graphics
VectSharp.Canvas.SKRenderAction, 418
VectSharp.Frame, 183
VectSharp.Page, 328
Gray
VectSharp.Colours, 108
Green
VectSharp.Colours, 108
GreenYellow

VectSharp.Colours, 109
Grey
VectSharp.Colours, 109
GreyScale
VectSharp.Filters.ColourMatrix, 83

H
VectSharp.Colour, 72
HasAlpha
VectSharp.RasterImage, 362
HeaderFontSizeMultipliers
VectSharp.Markdown.MarkdownRenderer, 300
HeaderLineColour
VectSharp.Markdown.MarkdownRenderer, 301
HeaderLineThicknesses
VectSharp.Markdown.MarkdownRenderer, 301
Height
VectSharp.Animation, 42
VectSharp.Font.DetailedFontMetrics, 141
VectSharp.IGraphicsContext, 256
VectSharp.Page, 328
VectSharp.RasterImage, 362
VectSharp.Size, 400
HoneyDew
VectSharp.Colours, 109
HotPink
VectSharp.Colours, 109

Id
VectSharp.RasterImage, 362
Identity
VectSharp.Filters.ColourMatrix, 83
ImageAction
VectSharp.Canvas.RenderAction, 379
VectSharp.Canvas.SKRenderAction, 413
ImageDataAddress
VectSharp.RasterImage, 362
ImageDestination
VectSharp.Canvas.RenderAction, 381
VectSharp.Canvas.SKRenderAction, 418
ImageId
VectSharp.Canvas.RenderAction, 382
VectSharp.Canvas.SKRenderAction, 418
ImageMarginTolerance
VectSharp.Markdown.MarkdownRenderer, 301
ImageMultiplier
VectSharp.Markdown.MarkdownRenderer, 301
ImageSideMargin
VectSharp.Markdown.MarkdownRenderer, 302
ImageSource
VectSharp.Canvas.RenderAction, 382
VectSharp.Canvas.SKRenderAction, 418
ImageUnitMultiplier
VectSharp.Markdown.MarkdownRenderer, 302
ImageUriResolver
VectSharp.Markdown.MarkdownRenderer, 302
IndentWidth
VectSharp.Markdown.MarkdownRenderer, 302
IndianRed
VectSharp.Colours, 109
Indigo
VectSharp.Colours, 110
InsertedColour
VectSharp.Markdown.MarkdownRenderer, 303
InsertLayer
VectSharp.Canvas.SKMultiLayerRenderCanvas, 405
Intensity
VectSharp.ThreeD.AmbientLightSource, 33
VectSharp.ThreeD.AreaLightSource, 47
VectSharp.ThreeD.LightIntensity, 276
VectSharp.ThreeD.MaskedLightSource, 314
VectSharp.ThreeD.ParallelLightSource, 333
VectSharp.ThreeD.PointLightSource, 349
VectSharp.ThreeD.SpotlightLightSource, 441
InterframeCompression
VectSharp.AnimatedPNG, 37
InternalPointer
VectSharp.DisposableIntPtr, 143
Interpolate
VectSharp.RasterImage, 363
Intersection
VectSharp.Rectangle, 373
InvalidateAll
VectSharp.Canvas.SKRenderAction, 413
InvalidateDirty
VectSharp.Canvas.SKMultiLayerRenderCanvas, 405
InvalidateHitTestPath
VectSharp.Canvas.SKRenderAction, 414
InvalidateVisual
VectSharp.Canvas.SKRenderAction, 414
InvalidateZIndex
VectSharp.Canvas.SKMultiLayerRenderCanvas, 405
VectSharp.Canvas.SKRenderAction, 414
InverseTransform
VectSharp.Canvas.RenderAction, 382
Inversion
VectSharp.Filters.ColourMatrix, 83
InvertedAlphaShift
VectSharp.Filters.ColourMatrix, 84
IsBold
VectSharp.FontFamily, 167
VectSharp.Markdown.FormattedString, 176
VectSharp.TrueTypeFile, 461
IsEqual
VectSharp.Point, 344
IsFixedPitch
VectSharp.TrueTypeFile, 462
IsItalic
VectSharp.FontFamily, 167
VectSharp.Markdown.FormattedString, 176
VectSharp.TrueTypeFile, 462
IsNaN
VectSharp.Rectangle, 374
IsOblique

VectSharp.FontFamily, 167
VectSharp.TrueTypeFile, 462
IsOnCurve
VectSharp.TrueTypeFile.TrueTypePoint, 464
IsPlaying
VectSharp.Canvas.AnimatedCanvas, 36
IsPlayingProperty
VectSharp.Canvas.AnimatedCanvas, 35
IsScript
VectSharp.TrueTypeFile, 462
IsSerif
VectSharp.TrueTypeFile, 463
IsStandardFamily
VectSharp.FontFamily, 167
ItalicFontFamily
VectSharp.Markdown.MarkdownRenderer, 303
Ivory
VectSharp.Colours, 110

Kernel
VectSharp.Filters.ConvolutionFilter, 136
Khaki
VectSharp.Colours, 110

L
VectSharp.Colour, 72
Lavender
VectSharp.Colours, 110
LavenderBlush
VectSharp.Colours, 110
LawnGreen
VectSharp.Colours, 111
LayerTransforms
VectSharp.Canvas.SKMultiLayerRenderCanvas, 408
Left
VectSharp.Markdown.Margins, 285
LeftSideBearing
VectSharp.Font.DetailedFontMetrics, 141
VectSharp.TrueTypeFile.Bearings, 53
LemonChiffon
VectSharp.Colours, 111
Library
VectSharp.Fonts.Nimbus, 318
LightBlue
VectSharp.Colours, 111
LightCoral
VectSharp.Colours, 111
LightCyan
VectSharp.Colours, 111
LightGoldenRodYellow
VectSharp.Colours, 112
LightGray
VectSharp.Colours, 112
LightGreen
VectSharp.Colours, 112
LightGrey
VectSharp.Colours, 112
LightIntensity
VectSharp.ThreeD.LightIntensity, 275
LightPink
VectSharp.Colours, 112
LightSalmon
VectSharp.Colours, 113
LightSeaGreen
VectSharp.Colours, 113
LightSkyBlue
VectSharp.Colours, 113
LightSlateGray
VectSharp.Colours, 113
LightSlateGrey
VectSharp.Colours, 113
LightSteelBlue
VectSharp.Colours, 114
LightYellow
VectSharp.Colours, 114
Lime
VectSharp.Colours, 114
LimeGreen
VectSharp.Colours, 114
LinearEasing
VectSharp.LinearEasing, 277
LinearGradientBrush
VectSharp.LinearGradientBrush, 279, 280
LinearisationResolution
VectSharp.Animation, 42
Linearise
VectSharp.Graphics, 207
VectSharp.GraphicsPath, 234
VectSharp.Segment, 391
LineCap
VectSharp.Font.FontUnderline, 174
VectSharp.IGraphicsContext, 257
LineCaps
VectSharp, 23
LineDash
VectSharp.LineDash, 282
LineJoin
VectSharp.IGraphicsContext, 257
LineJoins
VectSharp, 23
Linen
VectSharp.Colours, 114
LineTo
VectSharp.GraphicsPath, 235
VectSharp.IGraphicsContext, 252
LineWidth
VectSharp.IGraphicsContext, 257
LinkColour
VectSharp.Markdown.MarkdownRenderer, 303
LinkUriResolver
VectSharp.Markdown.MarkdownRenderer, 303
Location
VectSharp.Rectangle, 375
LogDownloads
VectSharp.Markdown.HTTPUtils, 240
LuminanceToAlpha

VectSharp.Filters.ColourMatrix, 80
LuminanceToColour
VectSharp.Filters.ColourMatrix, 81

Magenta
VectSharp.Colours, 115
Margins
VectSharp.Markdown.Margins, 284
VectSharp.Markdown.MarkdownRenderer, 303
MarkdownCanvasControl
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 287
MarkedColour
VectSharp.Markdown.MarkdownRenderer, 304
Maroon
VectSharp.Colours, 115
Mask
VectSharp.Filters.MaskFilter, 317
MaskedLightSource
VectSharp.ThreeD.MaskedLightSource, 311, 312
MaskFilter
VectSharp.Filters.MaskFilter, 316
MathFontScalingFactor
VectSharp.Markdown.MarkdownRenderer, 304
Max
VectSharp.Point, 344
MaxRenderWidth
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 289
MaxRenderWidthProperty
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 288
Measure
VectSharp.FormattedTextExtensions, 181
VectSharp.Segment, 392
MeasureLength
VectSharp.GraphicsPath, 235
MeasureText
VectSharp.Font, 158
VectSharp.Graphics, 208
MeasureTextAdvanced
VectSharp.Font, 158
MediumAquaMarine
VectSharp.Colours, 115
MediumBlue
VectSharp.Colours, 115
MediumOrchid
VectSharp.Colours, 115
MediumPurple
VectSharp.Colours, 116
MediumSeaGreen
VectSharp.Colours, 116
MediumSlateBlue
VectSharp.Colours, 116
MediumSpringGreen
VectSharp.Colours, 116
MediumTurquoise
VectSharp.Colours, 116
MediumVioletRed
VectSharp.Colours, 117
MidnightBlue
VectSharp.Colours, 117
Min
VectSharp.Point, 345
MinRenderWidth
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 290
MinRenderWidthProperty
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 288
MintCream
VectSharp.Colours, 117
MinVariation
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 290
MinVariationProperty
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 288
MistyRose
VectSharp.Colours, 117
Moccasin
VectSharp.Colours, 117
Modulus
VectSharp.Point, 345
MoveLayer
VectSharp.Canvas.SKMultiLayerRenderCanvas, 405
MoveTo
VectSharp.GraphicsPath, 236
VectSharp.IGraphicsContext, 252
MultiplyOpacity
VectSharp.Brush, 57
VectSharp.GradientStop, 188
VectSharp.LinearGradientBrush, 280
VectSharp.RadialGradientBrush, 352
VectSharp.SolidColourBrush, 432

NaN
VectSharp.Rectangle, 376
NavajoWhite
VectSharp.Colours, 118
Navy
VectSharp.Colours, 118
Normalisation
VectSharp.Filters.ConvolutionFilter, 137
Normalize
VectSharp.Point, 345

Offset
VectSharp.GradientStop, 188
OldLace
VectSharp.Colours, 118
Olive
VectSharp.Colours, 118
OliveDrab
VectSharp.Colours, 118
Operation
VectSharp.Canvas.FilterOption, 146

VectSharp.PDF.PDFContextInterpreter.FilterOption, 149
VectSharp.SVG.SVGContextInterpreter.FilterOption, 152
operator Brush
VectSharp.Brush, 58
operator SolidColourBrush
VectSharp.SolidColourBrush, 433
operator!=
VectSharp.Colour, 68
operator∗
VectSharp.Filters.ColourMatrix, 81
operator==
VectSharp.Colour, 68
Orange
VectSharp.Colours, 119
OrangeRed
VectSharp.Colours, 119
Orchid
VectSharp.Colours, 119
Origin
VectSharp.ThreeD.MaskedLightSource, 315
OutputFormats
VectSharp.Raster.ImageSharp, 27
OverallEasing
VectSharp.Transition, 451

Page
VectSharp.Page, 326
PageHeight
VectSharp.Canvas.SKMultiLayerRenderCanvas, 409
Pages
VectSharp.Document, 144
PageSize
VectSharp.Markdown.MarkdownRenderer, 304
PageWidth
VectSharp.Canvas.SKMultiLayerRenderCanvas, 409
Paint
VectSharp.Canvas.SKRenderAction, 419
PaintToAnimatedCanvas
VectSharp.Canvas.AvaloniaContextInterpreter, 49
PaintToCanvas
VectSharp.Canvas.AvaloniaContextInterpreter, 50–52
PaintToSKCanvas
VectSharp.Canvas.SKRenderContextInterpreter, 425–429
PaleGoldenRod
VectSharp.Colours, 119
PaleGreen
VectSharp.Colours, 119
PaleTurquoise
VectSharp.Colours, 120
PaleVioletRed
VectSharp.Colours, 120
PapayaWhip
VectSharp.Colours, 120
ParallelLightSource
VectSharp.ThreeD.ParallelLightSource, 331
Parent
VectSharp.Canvas.RenderAction, 382
VectSharp.Canvas.SKRenderAction, 419
ParseImageURI
VectSharp.SVG.Parser, 336
Parser
VectSharp.ImageSharpUtils.ImageURIParser, 269
VectSharp.MuPDFUtils.ImageURIParser, 270
ParseSVGURI
VectSharp.SVG.Parser, 335
Pastel
VectSharp.Filters.ColourMatrix, 84
Path
VectSharp.Canvas.SKRenderAction, 419
path
VectSharp.Markdown.HTTPUtils, 240
PathAction
VectSharp.Canvas.RenderAction, 379
VectSharp.Canvas.SKRenderAction, 415
Payload
VectSharp.Canvas.SKRenderAction, 419
PeachPuff
VectSharp.Colours, 120
PenumbraAttenuationExponent
VectSharp.ThreeD.AreaLightSource, 47
PenumbraRadius
VectSharp.ThreeD.AreaLightSource, 47
Peru
VectSharp.Colours, 120
Phase
VectSharp.LineDash, 283
PhongMaterial
VectSharp.ThreeD.PhongMaterial, 340
Pink
VectSharp.Colours, 121
PixelFormats
VectSharp, 23
Plum
VectSharp.Colours, 121
PNGStream
VectSharp.RasterImage, 363
Point
VectSharp.Point, 343
VectSharp.Segment, 393
PointerEnter
VectSharp.Canvas.RenderAction, 383
VectSharp.Canvas.SKRenderAction, 421
PointerLeave
VectSharp.Canvas.RenderAction, 383
VectSharp.Canvas.SKRenderAction, 421
PointerPressed
VectSharp.Canvas.RenderAction, 384
VectSharp.Canvas.SKRenderAction, 421
PointerReleased
VectSharp.Canvas.RenderAction, 384
VectSharp.Canvas.SKRenderAction, 421

PointLightSource
VectSharp.ThreeD.PointLightSource, 347
Points
VectSharp.Segment, 393
Position
VectSharp.Font.FontUnderline, 174
VectSharp.ThreeD.MaskedLightSource, 315
VectSharp.ThreeD.PointLightSource, 349
VectSharp.ThreeD.SpotlightLightSource, 441
PowderBlue
VectSharp.Colours, 121
PreserveAlpha
VectSharp.Filters.ConvolutionFilter, 137
Purple
VectSharp.Colours, 121

QuadraticBezierTo
VectSharp.GraphicsPath, 237
QuoteBlockBackgroundColour
VectSharp.Markdown.MarkdownRenderer, 304
QuoteBlockBarColour
VectSharp.Markdown.MarkdownRenderer, 304
QuoteBlockBarWidth
VectSharp.Markdown.MarkdownRenderer, 305
QuoteBlockIndentWidth
VectSharp.Markdown.MarkdownRenderer, 305

R
VectSharp.Colour, 72
VectSharp.SolidColourBrush, 434
R1
VectSharp.Filters.ColourMatrix, 87
R2
VectSharp.Filters.ColourMatrix, 87
R3
VectSharp.Filters.ColourMatrix, 88
R4
VectSharp.Filters.ColourMatrix, 88
R5
VectSharp.Filters.ColourMatrix, 88
RadialGradientBrush
VectSharp.RadialGradientBrush, 351
Radius
VectSharp.RadialGradientBrush, 353
VectSharp.ThreeD.AreaLightSource, 48
RangeRecord
VectSharp.TrueTypeFile.CoverageTable.RangeRecord, 354
RasterImage
VectSharp.RasterImage, 360, 361
RasterImageFile
VectSharp.ImageSharpUtils.RasterImageFile, 364
VectSharp.MuPDFUtils.RasterImageFile, 366
RasterImageLoader
VectSharp.Markdown.MarkdownRenderer, 305
RasterImageStream
VectSharp.ImageSharpUtils.RasterImageStream, 367, 368
VectSharp.MuPDFUtils.RasterImageStream, 369, 370
RasterisationMethod
VectSharp.Graphics, 222
RasterisationResolution
VectSharp.Canvas.FilterOption, 147
VectSharp.PDF.PDFContextInterpreter.FilterOption, 149
VectSharp.SVG.SVGContextInterpreter.FilterOption, 152
RasterisationResolutionRelative
VectSharp.Canvas.FilterOption, 147
VectSharp.PDF.PDFContextInterpreter.FilterOption, 149
VectSharp.SVG.SVGContextInterpreter.FilterOption, 152
Rasterise
VectSharp.Canvas.SKRenderContextInterpreter, 430
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter, 262
VectSharp.Raster.Raster, 355
RasteriseParameter
VectSharp.Filters.FilterWithRasterisableParameter, 154
VectSharp.Filters.IFilterWithRasterisableParameter, 245
RebeccaPurple
VectSharp.Colours, 121
Rectangle
VectSharp.IGraphicsContext, 252
VectSharp.Rectangle, 372, 373
Red
VectSharp.Colours, 122
RegularFontFamily
VectSharp.Markdown.MarkdownRenderer, 305
RelativeTo
VectSharp.LinearGradientBrush, 281
RemoveLastFrame
VectSharp.Animation, 41
RemoveLayer
VectSharp.Canvas.SKMultiLayerRenderCanvas, 406
Render
VectSharp.Canvas.AnimatedCanvas, 34
VectSharp.Canvas.SKMultiLayerRenderCanvas, 406
VectSharp.Markdown.MarkdownRenderer, 295
RenderActions
VectSharp.Canvas.SKMultiLayerRenderCanvas, 408
RenderAtResolution
VectSharp.Canvas.SKMultiLayerRenderCanvas, 406
Renderer
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 290
RenderLock

VectSharp.Canvas.SKMultiLayerRenderCanvas, 408
RenderSinglePage
VectSharp.Markdown.MarkdownRenderer, 296
RepeatCount
VectSharp.Animation, 42
Replace
VectSharp.ThreeD.IScene, 273, 274
VectSharp.ThreeD.Scene, 388
ResolveFontFamily
VectSharp.DefaultFontLibrary, 139
VectSharp.FontFamily, 164, 165
VectSharp.FontLibrary, 171, 172
VectSharp.IFontLibrary, 247, 248
VectSharp.SimpleFontLibrary, 398, 399
ResolveImageURI
VectSharp.Markdown.HTTPUtils, 239
ResourceFontFamily
VectSharp.ResourceFontFamily, 385
ResourceName
VectSharp.ResourceFontFamily, 385
Restore
VectSharp.Graphics, 209
RestoreAction
VectSharp.Canvas.SKRenderAction, 415
ReverseDirection
VectSharp.ThreeD.ParallelLightSource, 333
Right
VectSharp.Markdown.Margins, 285
RightSideBearing
VectSharp.Font.DetailedFontMetrics, 141
VectSharp.TrueTypeFile.Bearings, 53
RosyBrown
VectSharp.Colours, 122
Rotate
VectSharp.Graphics, 209
VectSharp.IGraphicsContext, 253
RotateAt
VectSharp.Graphics, 209
RoyalBlue
VectSharp.Colours, 122

S
VectSharp.Colour, 72
SaddleBrown
VectSharp.Colours, 122
Salmon
VectSharp.Colours, 122
SandyBrown
VectSharp.Colours, 123
Save
VectSharp.Graphics, 210
SaveAction
VectSharp.Canvas.SKRenderAction, 416
SaveAsAnimatedGIF
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter, 263, 264
SaveAsAnimatedPNG
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter, 265
VectSharp.Raster.Raster, 356
SaveAsAnimatedSVG
VectSharp.SVG.SVGContextInterpreter, 443, 444
SaveAsAnimatedSVGWithFrames
VectSharp.SVG.SVGContextInterpreter, 444–446
SaveAsImage
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter, 266, 267
SaveAsPDF
VectSharp.PDF.PDFContextInterpreter, 337, 338
SaveAsPNG
VectSharp.Raster.Raster, 357
SaveAsRawBytes
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter, 267, 268
VectSharp.Raster.Raster, 358
SaveAsSVG
VectSharp.SVG.SVGContextInterpreter, 447, 448
Scale
VectSharp.Filters.ConvolutionFilter, 137
VectSharp.Graphics, 210
VectSharp.IGraphicsContext, 253
Scene
VectSharp.ThreeD.Scene, 387
SceneElements
VectSharp.ThreeD.IScene, 274
VectSharp.ThreeD.Scene, 389
SceneLock
VectSharp.ThreeD.IScene, 274
VectSharp.ThreeD.Scene, 389
Script
VectSharp, 23
VectSharp.FormattedText, 181
SeaGreen
VectSharp.Colours, 123
SeaShell
VectSharp.Colours, 123
Segments
VectSharp.GraphicsPath, 238
SegmentType
VectSharp, 24
SendToBack
VectSharp.Canvas.RenderAction, 380
SetClippingPath
VectSharp.Graphics, 210, 211
SetFillStyle
VectSharp.IGraphicsContext, 253, 254
SetLineDash
VectSharp.IGraphicsContext, 254
SetStrokeStyle
VectSharp.IGraphicsContext, 254, 255
ShadowSamplingPointCount
VectSharp.ThreeD.AreaLightSource, 48
Sienna
VectSharp.Colours, 123
Silver

VectSharp.Colours, 123
SimpleFontLibrary
VectSharp.SimpleFontLibrary, 395, 396
Size
VectSharp.Rectangle, 376
VectSharp.Size, 400
SkipDescenders
VectSharp.Font.FontUnderline, 174
SKMultiLayerRenderCanvas
VectSharp.Canvas.SKMultiLayerRenderCanvas, 403, 404
SkyBlue
VectSharp.Colours, 124
SlateBlue
VectSharp.Colours, 124
SlateGray
VectSharp.Colours, 124
SlateGrey
VectSharp.Colours, 124
Snow
VectSharp.Colours, 124
SolidColourBrush
VectSharp.SolidColourBrush, 432
SolidLine
VectSharp.LineDash, 283
SourceDistance
VectSharp.ThreeD.AreaLightSource, 48
SpaceAfterHeading
VectSharp.Markdown.MarkdownRenderer, 305
SpaceAfterLine
VectSharp.Markdown.MarkdownRenderer, 306
SpaceAfterParagraph
VectSharp.Markdown.MarkdownRenderer, 306
SpaceBeforeHeading
VectSharp.Markdown.MarkdownRenderer, 306
SpaceBeforeParagaph
VectSharp.Markdown.MarkdownRenderer, 306
SpecularReflectionCoefficient
VectSharp.ThreeD.PhongMaterial, 341
SpecularShininess
VectSharp.ThreeD.PhongMaterial, 341
Spinner
VectSharp.Canvas.SKMultiLayerRenderCanvas, 409
SplineEasing
VectSharp.SplineEasing, 435
SpotlightLightSource
VectSharp.ThreeD.SpotlightLightSource, 438
SpringGreen
VectSharp.Colours, 125
StandardDeviation
VectSharp.Filters.GaussianBlurFilter, 185
StandardFamilies
VectSharp.FontFamily, 166
StandardFontFamilies
VectSharp.FontFamily, 162
StandardFontFamilyResources
VectSharp.FontFamily, 166
StartCoverageIndex
VectSharp.TrueTypeFile.CoverageTable.RangeRecord, 354
StartGlyphID
VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord, 59
VectSharp.TrueTypeFile.CoverageTable.RangeRecord, 354
StartPoint
VectSharp.LinearGradientBrush, 281
SteelBlue
VectSharp.Colours, 125
StopTolerance
VectSharp.GradientStops, 191
Stroke
VectSharp.Canvas.RenderAction, 382
StrokePath
VectSharp.Graphics, 212
StrokeRectangle
VectSharp.Graphics, 212, 213
StrokeStyle
VectSharp.IGraphicsContext, 257
StrokeText
VectSharp.Graphics, 213–215
VectSharp.IGraphicsContext, 255
StrokeTextOnPath
VectSharp.Graphics, 216
StrokeTextUnderline
VectSharp.Graphics, 217–219
SubscriptShift
VectSharp.Markdown.MarkdownRenderer, 306
SubsetFont
VectSharp.TrueTypeFile, 463
SubSuperscriptFontSize
VectSharp.Markdown.MarkdownRenderer, 307
SuperscriptShift
VectSharp.Markdown.MarkdownRenderer, 307
SwitchLayers
VectSharp.Canvas.SKMultiLayerRenderCanvas, 407
SyntaxHighlighter
VectSharp.Markdown.MarkdownRenderer, 307

TableCellMargins
VectSharp.Markdown.MarkdownRenderer, 307
TableHeaderRowSeparatorColour
VectSharp.Markdown.MarkdownRenderer, 307
TableHeaderRowSeparatorThickness
VectSharp.Markdown.MarkdownRenderer, 308
TableHeaderSeparatorThickness
VectSharp.Markdown.MarkdownRenderer, 308
TableRowSeparatorColour
VectSharp.Markdown.MarkdownRenderer, 308
TableVAlign
VectSharp.Markdown.MarkdownRenderer, 308
Tag
VectSharp.Canvas.RenderAction, 383
VectSharp.Canvas.SKRenderAction, 419
VectSharp.IGraphicsContext, 257

Tan
VectSharp.Colours, 125
TaskListCheckedBullet
VectSharp.Markdown.MarkdownRenderer, 308
TaskListUncheckedBullet
VectSharp.Markdown.MarkdownRenderer, 309
Teal
VectSharp.Colours, 125
Text
VectSharp.Canvas.RenderAction, 383
VectSharp.Canvas.SKRenderAction, 420
VectSharp.FormattedText, 181
VectSharp.Markdown.FormattedString, 176
TextAction
VectSharp.Canvas.RenderAction, 380
VectSharp.Canvas.SKRenderAction, 416
TextAnchors
VectSharp, 24
TextBaseline
VectSharp.IGraphicsContext, 258
TextBaselines
VectSharp, 24
TextConversionOption
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 290
TextConversionOptionsProperty
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 289
TextOptions
VectSharp.Canvas.AvaloniaContextInterpreter, 49
VectSharp.PDF.PDFContextInterpreter, 337
VectSharp.SVG.SVGContextInterpreter, 442
TextX
VectSharp.Canvas.SKRenderAction, 420
TextY
VectSharp.Canvas.SKRenderAction, 420
ThematicBreakLineColour
VectSharp.Markdown.MarkdownRenderer, 309
ThematicBreakThickness
VectSharp.Markdown.MarkdownRenderer, 309
Thickness
VectSharp.Font.FontUnderline, 174
this[int index]
VectSharp.GradientStops, 191
this[int y, int x]
VectSharp.Filters.ColourMatrix, 88
Thistle
VectSharp.Colours, 125
ToColour
VectSharp.Filters.ColourMatrix, 82
ToCSSString
VectSharp.Colour, 68
ToHSL
VectSharp.Colour, 69
ToLab
VectSharp.Colour, 69
Tomato
VectSharp.Colours, 126
Top
VectSharp.Font.DetailedFontMetrics, 141
VectSharp.Markdown.Margins, 285
TopLeftMargin
VectSharp.Filters.BoxBlurFilter, 56
VectSharp.Filters.ColourMatrixFilter, 91
VectSharp.Filters.CompositeLocationInvariantFilter, 130
VectSharp.Filters.ConvolutionFilter, 137
VectSharp.Filters.GaussianBlurFilter, 185
VectSharp.Filters.IFilter, 243
VectSharp.Filters.MaskFilter, 317
ToXYZ
VectSharp.Colour, 69
Transform
VectSharp.Canvas.RenderAction, 383
VectSharp.Canvas.SKRenderAction, 420
VectSharp.Graphics, 219, 220
VectSharp.GraphicsPath, 238
VectSharp.IGraphicsContext, 255
VectSharp.Segment, 392
TransformAction
VectSharp.Canvas.SKRenderAction, 417
Transition
VectSharp.Transition, 450
Transitions
VectSharp.Animation, 42
Translate
VectSharp.Graphics, 221
VectSharp.IGraphicsContext, 256
Triangulate
VectSharp.GraphicsPath, 238
TrueTypeFile
VectSharp.FontFamily, 168
TryRasterise
VectSharp.Graphics, 222
Turquoise
VectSharp.Colours, 126
Type
VectSharp.Segment, 393

UnbalancedStackAction
VectSharp.Graphics, 222
UnbalancedStackActions
VectSharp, 24
Underline
VectSharp.Font, 160
UnderlineThickness
VectSharp.Markdown.MarkdownRenderer, 310
Union
VectSharp.Rectangle, 374, 375
UnitsOff
VectSharp.LineDash, 283
UnitsOn
VectSharp.LineDash, 283
UpdateLayer
VectSharp.Canvas.SKMultiLayerRenderCanvas, 407
UpdateWith

VectSharp.Canvas.SKMultiLayerRenderCanvas, 407

VectSharp, 21
LineCaps, 23
LineJoins, 23
PixelFormats, 23
Script, 23
SegmentType, 24
TextAnchors, 24
TextBaselines, 24
UnbalancedStackActions, 24
VectSharp.AnimatedPNG, 37
Create, 38
InterframeCompression, 37
VectSharp.AnimatedPNG.CompressedFrame, 131
CompressedFrame, 132
Dispose, 133
Duration, 133
VectSharp.Animation, 38
AddFrame, 40
Animation, 39
Background, 41
Duration, 41
Frames, 42
GetFrameAtAbsolute, 40
GetFrameAtRelative, 41
Height, 42
LinearisationResolution, 42
RemoveLastFrame, 41
RepeatCount, 42
Transitions, 42
Width, 43
VectSharp.Brush, 57
MultiplyOpacity, 57
operator Brush, 58
VectSharp.Canvas, 25
VectSharp.Canvas.AnimatedCanvas, 33
CurrentFrame, 36
CurrentFrameProperty, 34
FrameCount, 36
FrameCountProperty, 35
FrameRate, 36
FrameRateProperty, 35
IsPlaying, 36
IsPlayingProperty, 35
Render, 34
VectSharp.Canvas.AvaloniaContextInterpreter, 48
PaintToAnimatedCanvas, 49
PaintToCanvas, 50–52
TextOptions, 49
VectSharp.Canvas.FilterOption, 145
Default, 146
FilterOperations, 145
FilterOption, 146
Operation, 146
RasterisationResolution, 147
RasterisationResolutionRelative, 147
VectSharp.Canvas.RenderAction, 376
ActionType, 381
ActionTypes, 378
BringToFront, 378
ClippingPath, 381
Fill, 381
Geometry, 381
ImageAction, 379
ImageDestination, 381
ImageId, 382
ImageSource, 382
InverseTransform, 382
Parent, 382
PathAction, 379
PointerEnter, 383
PointerLeave, 383
PointerPressed, 384
PointerReleased, 384
SendToBack, 380
Stroke, 382
Tag, 383
Text, 383
TextAction, 380
Transform, 383
VectSharp.Canvas.SKMultiLayerRenderCanvas, 401
AddLayer, 404
Dispose, 404
InsertLayer, 405
InvalidateDirty, 405
InvalidateZIndex, 405
LayerTransforms, 408
MoveLayer, 405
PageHeight, 409
PageWidth, 409
RemoveLayer, 406
Render, 406
RenderActions, 408
RenderAtResolution, 406
RenderLock, 408
SKMultiLayerRenderCanvas, 403, 404
Spinner, 409
SwitchLayers, 407
UpdateLayer, 407
UpdateWith, 407
VectSharp.Canvas.SKRenderAction, 409
ActionType, 417
ActionTypes, 412
ClipAction, 412
Dispose, 412
Disposed, 417
DrawFilteredGraphicsAction, 412
Filter, 417
Font, 418
Graphics, 418
ImageAction, 413
ImageDestination, 418
ImageId, 418
ImageSource, 418
InvalidateAll, 413

InvalidateHitTestPath, 414
InvalidateVisual, 414
InvalidateZIndex, 414
Paint, 419
Parent, 419
Path, 419
PathAction, 415
Payload, 419
PointerEnter, 421
PointerLeave, 421
PointerPressed, 421
PointerReleased, 421
RestoreAction, 415
SaveAction, 416
Tag, 419
Text, 420
TextAction, 416
TextX, 420
TextY, 420
Transform, 420
TransformAction, 417
ZIndex, 420
VectSharp.Canvas.SKRenderContext, 422
VectSharp.Canvas.SKRenderContextInterpreter, 422
CopyToSKRenderContext, 423–425
PaintToSKCanvas, 425–429
Rasterise, 430
VectSharp.Canvas/AnimatedCanvas.cs, 469
VectSharp.Canvas/AvaloniaContext.cs, 473
VectSharp.Canvas/RenderingParameters.cs, 507
VectSharp.Canvas/SKMultiLayerRenderCanvas.cs, 509
VectSharp.Canvas/SKRenderContext.cs, 524
VectSharp.Colour, 60
A, 71
a, 71
B, 71
Equals, 62
FromCSSString, 62
FromHSL, 62
FromLab, 63
FromRgb, 63, 64
FromRgba, 65–67
FromXYZ, 67
G, 71
GetHashCode, 68
H, 72
L, 72
operator!=, 68
operator==, 68
R, 72
S, 72
ToCSSString, 68
ToHSL, 69
ToLab, 69
ToXYZ, 69
WithAlpha, 69, 70
X, 73
Y, 73
VectSharp.Colours, 92
AliceBlue, 98
AntiqueWhite, 98
Aqua, 98
Aquamarine, 98
Azure, 98
Beige, 99
Bisque, 99
Black, 99
BlanchedAlmond, 99
Blue, 99
BlueViolet, 100
Brown, 100
BurlyWood, 100
CadetBlue, 100
Chartreuse, 100
Chocolate, 101
Coral, 101
CornflowerBlue, 101
Cornsilk, 101
Crimson, 101
Cyan, 102
DarkBlue, 102
DarkCyan, 102
DarkGoldenRod, 102
DarkGray, 102
DarkGreen, 103
DarkGrey, 103
DarkKhaki, 103
DarkMagenta, 103
DarkOliveGreen, 103
DarkOrange, 104
DarkOrchid, 104
DarkRed, 104
DarkSalmon, 104
DarkSeaGreen, 104
DarkSlateBlue, 105
DarkSlateGray, 105
DarkSlateGrey, 105
DarkTurquoise, 105
DarkViolet, 105
DeepPink, 106
DeepSkyBlue, 106
DimGray, 106
DimGrey, 106
DodgerBlue, 106
FireBrick, 107
FloralWhite, 107
ForestGreen, 107
Fuchsia, 107
Gainsboro, 107
GhostWhite, 108
Gold, 108
GoldenRod, 108
Gray, 108
Green, 108
GreenYellow, 109
Grey, 109

HoneyDew, 109
HotPink, 109
IndianRed, 109
Indigo, 110
Ivory, 110
Khaki, 110
Lavender, 110
LavenderBlush, 110
LawnGreen, 111
LemonChiffon, 111
LightBlue, 111
LightCoral, 111
LightCyan, 111
LightGoldenRodYellow, 112
LightGray, 112
LightGreen, 112
LightGrey, 112
LightPink, 112
LightSalmon, 113
LightSeaGreen, 113
LightSkyBlue, 113
LightSlateGray, 113
LightSlateGrey, 113
LightSteelBlue, 114
LightYellow, 114
Lime, 114
LimeGreen, 114
Linen, 114
Magenta, 115
Maroon, 115
MediumAquaMarine, 115
MediumBlue, 115
MediumOrchid, 115
MediumPurple, 116
MediumSeaGreen, 116
MediumSlateBlue, 116
MediumSpringGreen, 116
MediumTurquoise, 116
MediumVioletRed, 117
MidnightBlue, 117
MintCream, 117
MistyRose, 117
Moccasin, 117
NavajoWhite, 118
Navy, 118
OldLace, 118
Olive, 118
OliveDrab, 118
Orange, 119
OrangeRed, 119
Orchid, 119
PaleGoldenRod, 119
PaleGreen, 119
PaleTurquoise, 120
PaleVioletRed, 120
PapayaWhip, 120
PeachPuff, 120
Peru, 120
Pink, 121
Plum, 121
PowderBlue, 121
Purple, 121
RebeccaPurple, 121
Red, 122
RosyBrown, 122
RoyalBlue, 122
SaddleBrown, 122
Salmon, 122
SandyBrown, 123
SeaGreen, 123
SeaShell, 123
Sienna, 123
Silver, 123
SkyBlue, 124
SlateBlue, 124
SlateGray, 124
SlateGrey, 124
Snow, 124
SpringGreen, 125
SteelBlue, 125
Tan, 125
Teal, 125
Thistle, 125
Tomato, 126
Turquoise, 126
Violet, 126
Wheat, 126
White, 126
WhiteSmoke, 127
Yellow, 127
YellowGreen, 127
VectSharp.DefaultFontLibrary, 138
ResolveFontFamily, 139
VectSharp.DisposableIntPtr, 142
DisposableIntPtr, 143
Dispose, 143
InternalPointer, 143
VectSharp.Document, 144
Document, 144
Pages, 144
VectSharp.Filters, 25
VectSharp.Filters.BoxBlurFilter, 54
BottomRightMargin, 56
BoxBlurFilter, 55
BoxRadius, 56
Filter, 55
TopLeftMargin, 56
VectSharp.Filters.ColourMatrix, 75
A1, 84
A2, 84
A3, 85
A4, 85
A5, 85
AlphaInversion, 83
Apply, 78–80
B1, 85

B2, 85
B3, 86
B4, 86
B5, 86
ColourMatrix, 78
G1, 86
G2, 86
G3, 87
G4, 87
G5, 87
GreyScale, 83
Identity, 83
Inversion, 83
InvertedAlphaShift, 84
LuminanceToAlpha, 80
LuminanceToColour, 81
operator∗, 81
Pastel, 84
R1, 87
R2, 87
R3, 88
R4, 88
R5, 88
this[int y, int x], 88
ToColour, 82
WithAlpha, 82
VectSharp.Filters.ColourMatrixFilter, 89
BottomRightMargin, 91
ColourMatrix, 91
ColourMatrixFilter, 90
Filter, 90
TopLeftMargin, 91
VectSharp.Filters.CompositeLocationInvariantFilter, 128
BottomRightMargin, 130
CompositeLocationInvariantFilter, 129
Filter, 129
Filters, 130
TopLeftMargin, 130
VectSharp.Filters.ConvolutionFilter, 134
Bias, 136
BottomRightMargin, 136
ConvolutionFilter, 135
Filter, 136
Kernel, 136
Normalisation, 137
PreserveAlpha, 137
Scale, 137
TopLeftMargin, 137
VectSharp.Filters.FilterWithRasterisableParameter, 153
Dispose, 153
RasteriseParameter, 154
VectSharp.Filters.GaussianBlurFilter, 183
BottomRightMargin, 185
Filter, 185
GaussianBlurFilter, 184
StandardDeviation, 185
TopLeftMargin, 185
VectSharp.Filters.IFilter, 242
BottomRightMargin, 243
TopLeftMargin, 243
VectSharp.Filters.IFilterWithLocation, 243
Filter, 244
VectSharp.Filters.IFilterWithRasterisableParameter, 244
RasteriseParameter, 245
VectSharp.Filters.ILocationInvariantFilter, 260
Filter, 261
VectSharp.Filters.MaskFilter, 315
BottomRightMargin, 317
Filter, 316
Mask, 317
MaskFilter, 316
TopLeftMargin, 317
VectSharp.Font, 154
Ascent, 159
Descent, 159
EnableKerning, 159
Font, 156
FontFamily, 159
FontSize, 159
MeasureText, 158
MeasureTextAdvanced, 158
Underline, 160
WinAscent, 160
YMax, 160
YMin, 160
VectSharp.Font.DetailedFontMetrics, 140
AdvanceWidth, 140
Bottom, 140
Height, 141
LeftSideBearing, 141
RightSideBearing, 141
Top, 141
Width, 141
VectSharp.Font.FontUnderline, 173
FollowItalicAngle, 174
LineCap, 174
Position, 174
SkipDescenders, 174
Thickness, 174
VectSharp.FontFamily, 161
DefaultFontLibrary, 166
FamilyName, 166
FileName, 167
FontFamily, 163
IsBold, 167
IsItalic, 167
IsOblique, 167
IsStandardFamily, 167
ResolveFontFamily, 164, 165
StandardFamilies, 166
StandardFontFamilies, 162
StandardFontFamilyResources, 166
TrueTypeFile, 168
VectSharp.FontFamilyCreationException, 168
FontFamily, 169
FontFamilyCreationException, 169

VectSharp.FontLibrary, 170
ResolveFontFamily, 171, 172
VectSharp.Fonts, 26
VectSharp.Fonts.Nimbus, 318
Library, 318
VectSharp.Fonts.Nimbus/Nimbus.cs, 548
VectSharp.FormattedText, 177
Brush, 180
Font, 180
Format, 178, 179
FormattedText, 178
Script, 181
Text, 181
VectSharp.FormattedTextExtensions, 181
Measure, 181
VectSharp.Frame, 182
Duration, 183
Frame, 182
Graphics, 183
VectSharp.GradientBrush, 186
GradientStops, 187
VectSharp.GradientStop, 187
Colour, 188
GradientStop, 187
MultiplyOpacity, 188
Offset, 188
VectSharp.GradientStops, 189
Count, 191
GetColourAt, 190
GetEnumerator, 191
GradientStops, 190
StopTolerance, 191
this[int index], 191
VectSharp.Graphics, 192
CopyToIGraphicsContext, 195
Crop, 196
DrawGraphics, 196–198
DrawRasterImage, 199–201
FillPath, 201
FillRectangle, 202
FillText, 203, 204
FillTextOnPath, 204
FillTextUnderline, 205–207
GetBounds, 207
GetTags, 207
Linearise, 207
MeasureText, 208
RasterisationMethod, 222
Restore, 209
Rotate, 209
RotateAt, 209
Save, 210
Scale, 210
SetClippingPath, 210, 211
StrokePath, 212
StrokeRectangle, 212, 213
StrokeText, 213–215
StrokeTextOnPath, 216
StrokeTextUnderline, 217–219
Transform, 219, 220
Translate, 221
TryRasterise, 222
UnbalancedStackAction, 222
VectSharp.GraphicsPath, 223
AddPath, 225
AddSmoothSpline, 225
AddText, 226
AddTextOnPath, 227
AddTextUnderline, 227
Arc, 228
Close, 229
CubicBezierTo, 229
Discretise, 230
EllipticalArc, 230
GetBounds, 231
GetFigures, 231
GetLinearisationPointsNormals, 231
GetNormalAtAbsolute, 232
GetNormalAtRelative, 232
GetPointAtAbsolute, 232
GetPointAtRelative, 233
GetPoints, 233
GetTangentAtAbsolute, 233
GetTangentAtRelative, 234
Linearise, 234
LineTo, 235
MeasureLength, 235
MoveTo, 236
QuadraticBezierTo, 237
Segments, 238
Transform, 238
Triangulate, 238
VectSharp.IEasing, 241
Ease, 241
VectSharp.IFontLibrary, 246
ResolveFontFamily, 247, 248
VectSharp.IGraphicsContext, 249
CubicBezierTo, 250
DrawFilteredGraphics, 251
DrawRasterImage, 251
FillStyle, 256
FillText, 251
Font, 256
Height, 256
LineCap, 257
LineJoin, 257
LineTo, 252
LineWidth, 257
MoveTo, 252
Rectangle, 252
Rotate, 253
Scale, 253
SetFillStyle, 253, 254
SetLineDash, 254
SetStrokeStyle, 254, 255
StrokeStyle, 257

StrokeText, 255
Tag, 257
TextBaseline, 258
Transform, 255
Translate, 256
Width, 258
VectSharp.ImageSharpUtils, 26
VectSharp.ImageSharpUtils.ImageURIParser, 268
Parser, 269
VectSharp.ImageSharpUtils.RasterImageFile, 364
RasterImageFile, 364
VectSharp.ImageSharpUtils.RasterImageStream, 366
RasterImageStream, 367, 368
VectSharp.ImageSharpUtils/ImageUriParser.cs, 604
VectSharp.ImageSharpUtils/RasterImages.cs, 607
VectSharp.LinearEasing, 277
Ease, 278
LinearEasing, 277
VectSharp.LinearGradientBrush, 278
EndPoint, 281
LinearGradientBrush, 279, 280
MultiplyOpacity, 280
RelativeTo, 281
StartPoint, 281
VectSharp.LineDash, 282
LineDash, 282
Phase, 283
SolidLine, 283
UnitsOff, 283
UnitsOn, 283
VectSharp.Markdown, 26
VectSharp.Markdown.FormattedString, 175
Colour, 176
FormattedString, 175
IsBold, 176
IsItalic, 176
Text, 176
VectSharp.Markdown.HTTPUtils, 239
LogDownloads, 240
path, 240
ResolveImageURI, 239
VectSharp.Markdown.Margins, 284
Bottom, 285
Left, 285
Margins, 284
Right, 285
Top, 285
VectSharp.Markdown.MarkdownRenderer, 291
AllowPageBreak, 297
BackgroundColour, 297
BaseFontSize, 297
BaseImageUri, 297
BaseLinkUri, 298
BoldFontFamily, 298
BoldItalicFontFamily, 298
BoldUnderlineThickness, 298
Bullets, 298
CodeBlockBackgroundColour, 299
CodeFont, 299
CodeFontBold, 299
CodeFontBoldItalic, 299
CodeFontItalic, 300
CodeInlineBackgroundColour, 300
CodeInlineMargin, 300
ForegroundColour, 300
HeaderFontSizeMultipliers, 300
HeaderLineColour, 301
HeaderLineThicknesses, 301
ImageMarginTolerance, 301
ImageMultiplier, 301
ImageSideMargin, 302
ImageUnitMultiplier, 302
ImageUriResolver, 302
IndentWidth, 302
InsertedColour, 303
ItalicFontFamily, 303
LinkColour, 303
LinkUriResolver, 303
Margins, 303
MarkedColour, 304
MathFontScalingFactor, 304
PageSize, 304
QuoteBlockBackgroundColour, 304
QuoteBlockBarColour, 304
QuoteBlockBarWidth, 305
QuoteBlockIndentWidth, 305
RasterImageLoader, 305
RegularFontFamily, 305
Render, 295
RenderSinglePage, 296
SpaceAfterHeading, 305
SpaceAfterLine, 306
SpaceAfterParagraph, 306
SpaceBeforeHeading, 306
SpaceBeforeParagaph, 306
SubscriptShift, 306
SubSuperscriptFontSize, 307
SuperscriptShift, 307
SyntaxHighlighter, 307
TableCellMargins, 307
TableHeaderRowSeparatorColour, 307
TableHeaderRowSeparatorThickness, 308
TableHeaderSeparatorThickness, 308
TableRowSeparatorColour, 308
TableVAlign, 308
TaskListCheckedBullet, 308
TaskListUncheckedBullet, 309
ThematicBreakLineColour, 309
ThematicBreakThickness, 309
UnderlineThickness, 310
VerticalAlignment, 295
VectSharp.Markdown.SyntaxHighlighter, 448
GetSyntaxHighlightedLines, 449
VectSharp.Markdown/HtmlTag.cs, 549
VectSharp.Markdown/Line.cs, 554
VectSharp.Markdown/MarkdownContext.cs, 556

VectSharp.Markdown/MarkdownRenderer.cs, 559
VectSharp.Markdown/SyntaxHighlighting.cs, 592
VectSharp.Markdown/Word.cs, 596
VectSharp.MarkdownCanvas, 26
VectSharp.MarkdownCanvas.MarkdownCanvasControl, 286
Document, 289
DocumentProperty, 287
DocumentSource, 289
DocumentSourceProperty, 288
MarkdownCanvasControl, 287
MaxRenderWidth, 289
MaxRenderWidthProperty, 288
MinRenderWidth, 290
MinRenderWidthProperty, 288
MinVariation, 290
MinVariationProperty, 288
Renderer, 290
TextConversionOption, 290
TextConversionOptionsProperty, 289
VectSharp.MarkdownCanvas/ImageCache.cs, 598
VectSharp.MarkdownCanvas/MarkdownCanvas.axaml.cs, 600
VectSharp.MuPDFUtils, 27
VectSharp.MuPDFUtils.ImageURIParser, 269
Parser, 270
VectSharp.MuPDFUtils.RasterImageFile, 365
RasterImageFile, 366
VectSharp.MuPDFUtils.RasterImageStream, 368
RasterImageStream, 369, 370
VectSharp.MuPDFUtils/ImageURIParser.cs, 605
VectSharp.MuPDFUtils/RasterImages.cs, 610
VectSharp.Page, 326
Background, 328
Crop, 327, 328
Graphics, 328
Height, 328
Page, 326
Width, 328
VectSharp.PDF, 27
VectSharp.PDF.PDFContextInterpreter, 336
SaveAsPDF, 337, 338
TextOptions, 337
VectSharp.PDF.PDFContextInterpreter.FilterOption, 147
Default, 149
FilterOperations, 148
FilterOption, 148
Operation, 149
RasterisationResolution, 149
RasterisationResolutionRelative, 149
VectSharp.PDF/PDFContext.cs, 613
VectSharp.Point, 342
Bounds, 343
IsEqual, 344
Max, 344
Min, 345
Modulus, 345
Normalize, 345
Point, 343
X, 346
Y, 346
VectSharp.RadialGradientBrush, 350
Centre, 352
FocalPoint, 353
MultiplyOpacity, 352
RadialGradientBrush, 351
Radius, 353
VectSharp.Raster, 27
VectSharp.Raster.ImageSharp, 27
OutputFormats, 27
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter, 261
Rasterise, 262
SaveAsAnimatedGIF, 263, 264
SaveAsAnimatedPNG, 265
SaveAsImage, 266, 267
SaveAsRawBytes, 267, 268
VectSharp.Raster.ImageSharp.ImageSharpContextInterpreter.UnknownFo 466
Format, 466
VectSharp.Raster.ImageSharp/GradientBrushApplicator.cs, 651
VectSharp.Raster.ImageSharp/ImageSharpContext.cs, 654
VectSharp.Raster.Raster, 354
Rasterise, 355
SaveAsAnimatedPNG, 356
SaveAsPNG, 357
SaveAsRawBytes, 358
VectSharp.Raster/Raster.cs, 674
VectSharp.RasterImage, 358
ClearPNGCache, 361
DataHolder, 362
Dispose, 361
HasAlpha, 362
Height, 362
Id, 362
ImageDataAddress, 362
Interpolate, 363
PNGStream, 363
RasterImage, 360, 361
Width, 363
VectSharp.Rectangle, 370
Centre, 376
Intersection, 373
IsNaN, 374
Location, 375
NaN, 376
Rectangle, 372, 373
Size, 376
Union, 374, 375
VectSharp.ResourceFontFamily, 384
ResourceFontFamily, 385
ResourceName, 385
VectSharp.Segment, 389
Clone, 390

GetLinearisationTangents, 390
GetPointAt, 391
GetTangentAt, 391
Linearise, 391
Measure, 392
Point, 393
Points, 393
Transform, 392
Type, 393
VectSharp.SimpleFontLibrary, 394
Add, 397, 398
ResolveFontFamily, 398, 399
SimpleFontLibrary, 395, 396
VectSharp.Size, 399
Height, 400
Size, 400
Width, 401
VectSharp.SolidColourBrush, 431
A, 433
B, 433
Colour, 433
G, 434
MultiplyOpacity, 432
operator SolidColourBrush, 433
R, 434
SolidColourBrush, 432
VectSharp.SplineEasing, 434
ControlPoint1, 436
ControlPoint2, 436
Ease, 435
SplineEasing, 435
VectSharp.SVG, 28
VectSharp.SVG.Parser, 334
FromFile, 334
FromStream, 335
FromString, 335
ParseImageURI, 336
ParseSVGURI, 335
VectSharp.SVG.SVGContextInterpreter, 441
SaveAsAnimatedSVG, 443, 444
SaveAsAnimatedSVGWithFrames, 444–446
SaveAsSVG, 447, 448
TextOptions, 442
VectSharp.SVG.SVGContextInterpreter.FilterOption, 150
Default, 151
FilterOperations, 151
FilterOption, 151
Operation, 152
RasterisationResolution, 152
RasterisationResolutionRelative, 152
VectSharp.SVG/SVGContext.cs, 678
VectSharp.SVG/SVGParser.cs, 724
VectSharp.ThreeD, 28
VectSharp.ThreeD.AmbientLightSource, 29
AmbientLightSource, 31
CastsShadow, 32
GetLightAt, 31
GetObstruction, 32
Intensity, 33
VectSharp.ThreeD.AreaLightSource, 43
AreaLightSource, 45
CastsShadow, 46
Center, 46
Direction, 47
DistanceAttenuationExponent, 47
GetLightAt, 45
GetObstruction, 46
Intensity, 47
PenumbraAttenuationExponent, 47
PenumbraRadius, 47
Radius, 48
ShadowSamplingPointCount, 48
SourceDistance, 48
VectSharp.ThreeD.ColourMaterial, 73
Colour, 75
ColourMaterial, 74
GetColour, 74
VectSharp.ThreeD.ILightSource, 258
CastsShadow, 260
GetLightAt, 259
GetObstruction, 259
VectSharp.ThreeD.IMaterial, 270
GetColour, 271
VectSharp.ThreeD.IScene, 272
AddElement, 273
AddRange, 273
Replace, 273, 274
SceneElements, 274
SceneLock, 274
VectSharp.ThreeD.LightIntensity, 275
Deconstruct, 276
Direction, 276
Intensity, 276
LightIntensity, 275
VectSharp.ThreeD.MaskedLightSource, 310
AngleAttenuationExponent, 313
CastsShadow, 314
Direction, 314
Distance, 314
DistanceAttenuationExponent, 314
GetLightAt, 313
GetObstruction, 313
Intensity, 314
MaskedLightSource, 311, 312
Origin, 315
Position, 315
VectSharp.ThreeD.ObjectFactory, 319
CreateCube, 319
CreateCuboid, 320
CreatePoints, 321
CreatePolygon, 321
CreatePrism, 322
CreateRectangle, 323
CreateSphere, 324
CreateTetrahedron, 324

CreateWireframe, 325
VectSharp.ThreeD.ParallelLightSource, 330
CastsShadow, 333
Direction, 333
GetLightAt, 332
GetObstruction, 332
Intensity, 333
ParallelLightSource, 331
ReverseDirection, 333
VectSharp.ThreeD.PhongMaterial, 338
AmbientReflectionCoefficient, 341
Colour, 341
DiffuseReflectionCoefficient, 341
GetColour, 340
PhongMaterial, 340
SpecularReflectionCoefficient, 341
SpecularShininess, 341
VectSharp.ThreeD.PointLightSource, 346
CastsShadow, 349
DistanceAttenuationExponent, 349
GetLightAt, 348
GetObstruction, 348
Intensity, 349
PointLightSource, 347
Position, 349
VectSharp.ThreeD.Scene, 386
AddElement, 387
AddRange, 388
Replace, 388
Scene, 387
SceneElements, 389
SceneLock, 389
VectSharp.ThreeD.SpotlightLightSource, 437
AngleAttenuationExponent, 439
BeamWidthAngle, 440
CastsShadow, 440
CutoffAngle, 440
Direction, 440
DistanceAttenuationExponent, 440
GetLightAt, 439
GetObstruction, 439
Intensity, 441
Position, 441
SpotlightLightSource, 438
VectSharp.ThreeD/Lights.cs, 761
VectSharp.ThreeD/Materials.cs, 772
VectSharp.ThreeD/ObjectFactory.cs, 774
VectSharp.ThreeD/Scene.cs, 781
VectSharp.Transition, 449
Duration, 450
Easings, 450
OverallEasing, 451
Transition, 450
VectSharp.TrueTypeFile, 451
Destroy, 453
FontStream, 464
Get1000EmAscent, 453
Get1000EmDescent, 454
Get1000EmGlyphBearings, 454
Get1000EmGlyphVerticalMetrics, 454
Get1000EmGlyphWidth, 455
Get1000EmKerning, 456
Get1000EmUnderlineIntersections, 456
Get1000EmUnderlinePosition, 457
Get1000EmUnderlineThickness, 457
Get1000EmWinAscent, 457
Get1000EmXMax, 458
Get1000EmXMin, 458
Get1000EmYMax, 458
Get1000EmYMin, 458
GetFirstCharIndex, 459
GetFontFamilyName, 459
GetFontName, 459
GetFullFontFamilyName, 459
GetGlyphIndex, 460
GetGlyphPath, 460, 461
GetItalicAngle, 461
GetLastCharIndex, 461
IsBold, 461
IsFixedPitch, 462
IsItalic, 462
IsOblique, 462
IsScript, 462
IsSerif, 463
SubsetFont, 463
VectSharp.TrueTypeFile.Bearings, 53
LeftSideBearing, 53
RightSideBearing, 53
VectSharp.TrueTypeFile.ClassDefinitionTable.ClassRangeRecord, 58
Class, 59
ClassRangeRecord, 59
EndGlyphID, 59
StartGlyphID, 59
VectSharp.TrueTypeFile.CoverageTable.RangeRecord, 353
EndGlyphID, 354
RangeRecord, 354
StartCoverageIndex, 354
StartGlyphID, 354
VectSharp.TrueTypeFile.PairKerning, 329
Glyph1Advance, 329
Glyph1Placement, 329
Glyph2Advance, 330
Glyph2Placement, 330
VectSharp.TrueTypeFile.TrueTypePoint, 464
IsOnCurve, 464
X, 464
Y, 465
VectSharp.TrueTypeFile.VerticalMetrics, 467
YMax, 467
YMin, 467
VectSharp.UnbalancedStackException, 465
VectSharp/Animation.cs, 782
VectSharp/Brush.cs, 804
VectSharp/Colour.cs, 810

VectSharp/Document.cs, 818
VectSharp/Enums.cs, 820
VectSharp/Filters/BoxBlurFilter.cs, 822
VectSharp/Filters/ColourMatrixFilter.cs, 827
VectSharp/Filters/CompositeFilter.cs, 836
VectSharp/Filters/ConvolutionFilter.cs, 838
VectSharp/Filters/Filters.cs, 842
VectSharp/Filters/GaussianBlurFilter.cs, 845
VectSharp/Filters/MaskFilter.cs, 849
VectSharp/Font.cs, 851
VectSharp/FontLibrary.cs, 860
VectSharp/FormattedText.cs, 868
VectSharp/Graphics.cs, 877
VectSharp/Graphics.Text.cs, 899
VectSharp/GraphicsAction.cs, 912
VectSharp/GraphicsPath.cs, 915
VectSharp/ImageFormats.cs, 959
VectSharp/Point.cs, 973
VectSharp/RasterImage.cs, 977
VectSharp/Segment.cs, 981
VectSharp/SmoothSpline.cs, 990
VectSharp/StandardColours.cs, 992
VectSharp/TrueType.cs, 1001
VerticalAlignment
    VectSharp.Markdown.MarkdownRenderer, 295
Violet
    VectSharp.Colours, 126

Wheat
    VectSharp.Colours, 126
White
    VectSharp.Colours, 126
WhiteSmoke
    VectSharp.Colours, 127
Width
    VectSharp.Animation, 43
    VectSharp.Font.DetailedFontMetrics, 141
    VectSharp.IGraphicsContext, 258
    VectSharp.Page, 328
    VectSharp.RasterImage, 363
    VectSharp.Size, 401
WinAscent
    VectSharp.Font, 160
WithAlpha
    VectSharp.Colour, 69, 70
    VectSharp.Filters.ColourMatrix, 82

X
    VectSharp.Colour, 73
    VectSharp.Point, 346
    VectSharp.TrueTypeFile.TrueTypePoint, 464

Y
    VectSharp.Colour, 73
    VectSharp.Point, 346
    VectSharp.TrueTypeFile.TrueTypePoint, 465
Yellow
    VectSharp.Colours, 127
YellowGreen
    VectSharp.Colours, 127
YMax
    VectSharp.Font, 160
    VectSharp.TrueTypeFile.VerticalMetrics, 467
YMin
    VectSharp.Font, 160
    VectSharp.TrueTypeFile.VerticalMetrics, 467

ZIndex
    VectSharp.Canvas.SKRenderAction, 420