WPF201-B

Module 3

WPF Graphics, Documents & Animations

1 Vector Graphics

1.1 Lines & Shapes

Use **Line** element to render a simple line. Use **X1**, **Y1** and **X2**, **Y2** to specify the start and end points of the line. Use **Stroke** to assign the brush to paint the line and **StrokeThickness** to determine with width of the line.

Basic Line element: Lines1\MainWindow.xaml

```
<UniformGrid Rows="2" Columns="4">
    <Line X1="4" Y1="4" X2="36" Y2="36"
        Stroke="Black" StrokeThickness="2" />
</UniformGrid>
```

Points do not have to represent an absolute pixel position as the line can be resized when you set **Stretch** property. You can specify the length using **Width** and **Height** property. Alternatively you can fill up the available space by not using a fixed width and height. Alignment and **Margin** properties can also be used to adjusting the start position of the line.

Using size and alignment properties

```
<Line X1="0" Y1="0" X2="1" Y2="1"
    Stroke="Red" StrokeThickness="2"
    Stretch="Fill" Margin="4" Width="36" Height="36"
    HorizontalAlignment="Left"VerticalAlignment="Top" />
```

Creating a variable-length vertical line

```
<Line X1="0" Y1="0" X2="0" Y2="1"
    Stroke="Blue" StrokeThickness="2"
    Stretch="Fill" Margin="4" />
```

Creating a variable-length horizontal line

```
<Line X1="0" Y1="0" X2="1" Y2="0"
    Stroke="Blue" StrokeThickness="2"
    Stretch="Fill" Margin="4" />
```

When a line is thick, you notice that ends of the line will be flat. You can add a cap to the start and end using the **StrokeStartLineCap** and **StrokeLineEndCap** properties. The caps can be **Square**, **Triangle** or **Round** at either ends.

Using line caps

```
<Line X1="0" Y1="0" X2="1" Y2="1"
    StrokeStartLineCap="Round" StrokeEndLineCap="Triangle"
    Stroke="Green" StrokeThickness="8"
    Stretch="Fill" Margin="4" />
```

Lines do not have to be solid. Use **StrokeDashArray** to specify the lengths of each dash and space between dashes. Odd position numbers are the length of the dashes and even position numbers are for dash spacing. You can also use **StrokeDashCap** to add start and end cap to each dash. You should increase the lengths to accommodate caps. Alternatively set zero lengths to only render the caps. Use **StrokeDashOffset** to offset the start of the first dash.

A dashed-line

```
<Line X1="0" Y1="0" X2="1" Y2="0" StrokeDashArray="1 1 2 1"
    Stroke="Green" StrokeThickness="4"
    Stretch="Fill" Margin="4" />
```

A dashed-line with caps

```
<Line X1="0" Y1="0" X2="1" Y2="0" StrokeDashArray="2 3 4 2"
    StrokeDashCap="Round" StrokeDashOffset="3"
    Stroke="SteelBlue" StrokeThickness="4"
    Stretch="Fill" Margin="4" />
```

A caps-only line

```
<Line X1="0" Y1="0" X2="1" Y2="0" StrokeDashArray="0 2"
    StrokeDashCap="Round" StrokeDashOffset="1"
    Stroke="SteelBlue" StrokeThickness="4"
    Stretch="Fill" Margin="4" />
```

To draw a set of connected lines, you can use the **Polyline** element. It has a **Points** property that is assigned a **PointCollection** object where you can add any number of **Point** elements. Lines connecting points will be rendered. All existing line properties are available and you can additionally use **Fill** property to assign the brush to render any area surrounded by the lines. If you need to draw an enclosed shape, the last point will have to be the same as the first point. Alternatively use a **Polygon** that automatically connects the last point to the first point.

Drawing a set of connected line: Shapes1\MainWindow.xaml

```
<UniformGrid Rows="2" Columns="4">
    <Polyline Points="40,0 50,20 70,10 60,30 80,40 60,50 70,70
        50,60 40,80 30,60 10,70 20,50 0,40 20,30 10,10 30,20 40,0"
        StrokeThickness="2" Stroke="Black" />
</Uniform>
```

Drawing an enclosed shape

```
<Polygon Points="40,0 50,20 70,10 60,30 80,40 60,50 70,70
    50,60 40,80 30,60 10,70 20,50 0,40 20,30 10,10 30,20"
    StrokeThickness="2" Fill="Aquamarine" Stroke="Green" />
```

Just like **Line**, the points do not have to represent absolute pixel positions. You can use **Stretch** property to resize the shape to fit a fixed size of stretch to the available space. All **Stretch** modes are available so you choose whether you wish to maintain the aspect ratio of the original coordinates. The alignment and margin properties can be used to determine the start position of the shape and spacing around the shape.

Using size and alignment properties

```
<Polygon Points="40,0 50,20 70,10 60,30 80,40 60,50 70,70
    50,60 40,80 30,60 10,70 20,50 0,40 20,30 10,10 30,20"
    StrokeThickness="2" Fill="Aquamarine" Stroke="Green"
    Stretch="Fill" Margin="4" Width="80" Height="40"
    HorizontalAlignment="Right" VerticalAlignment="Bottom"/>
```

Stretch to available space

```
<Polygon Points="40,0 50,20 70,10 60,30 80,40 60,50 70,70
    50,60 40,80 30,60 10,70 20,50 0,40 20,30 10,10 30,20"
    StrokeThickness="2" Fill="Aquamarine" Stroke="Green"
    Stretch="Fill" Margin="4" />
```

Uniform stretch to available space

```
<Polygon Points="40,0 50,20 70,10 60,30 80,40 60,50 70,70
    50,60 40,80 30,60 10,70 20,50 0,40 20,30 10,10 30,20"
    StrokeThickness="2" Fill="Aquamarine" Stroke="Green"
    Stretch="Uniform" Margin="4" />
```

Regardless of thickness of lines, the joints between the lines will always be sharpened because **StrokeLineJoin** property is set to **Miter** by default. You can also change it to **Bevel** or **Round** as shown below.

Using bevelled line joins

```
<Polygon Points="40,0 50,20 70,10 60,30 80,40 60,50 70,70
    50,60 40,80 30,60 10,70 20,50 0,40 20,30 10,10 30,20"
    StrokeThickness="8" Fill="Red" Stroke="Maroon"
    StrokeLineJoin="Bevel" Stretch="Fill" Margin="4" />
```

If you do not wish to create a solid outline, all dash properties available for the **Line** such as **StrokeDashArray**, **StrokeDashCap** and **StrokeDashOffset** can be used for shapes as well.

Using stroke dash properties

```
<Polygon Points="40,0 50,20 70,10 60,30 80,40 60,50 70,70
    50,60 40,80 30,60 10,70 20,50 0,40 20,30 10,10 30,20"
    StrokeThickness="2" Fill="Gray" Stroke="Black"
    StrokeDashArray="0 2" StrokeDashCap="Round"
    Stretch="Fill" Margin="4" />
```

You can also use the **Ellipse** and **Rectangle** shape elements to draw simple shapes. You cannot specify the points directly but you can use **Stretch** to resize the shape to fit available space or to a fixed **Width** and **Height**. All of the dash properties are available if you do not want the stroke to be solid. Additionally a **Rectangle** element also has extra **RadiusX** and **RadiusY** properties to render rounded corners.

Using an Ellipse element: Shapes2\MainWindow.xaml

```
<UniformGrid Rows="3" Columns="2">
    <Ellipse Stroke="Maroon" StrokeThickness="4"
        Fill="Red" Margin="4" />
</UniformGrid>
```

Setting size and alignment properties

```
<Ellipse Stroke="Maroon" StrokeThickness="4"
    Fill="Red" Margin="4" Width="64" Height="64"
    HorizontalAlignment="Center" VerticalAlignment="Center" />
```

Using Rectangle element

```
<Rectangle Stroke="Gold" StrokeThickness="4"
    Fill="Yellow" Margin="4" Width="64" Height="64" />
```

Rectangle with rounded corners

```
<Rectangle Stroke="Gold" StrokeThickness="4" Width="64" Height="64"
    Fill="Yellow" Margin="4" RadiusX="32" RadiusY="12" />
```

1.2 Paths & Geometries

Even though it is possible to render a curved shape using straight lines you will need to specify a lot of points. This will be too tedious to do manually unless the points are generated with mathematical formulas. Use the **PathGeometry** to represent complex shapes using figures and segments where segments can be constructed using straight or curved lines. Geometry are resources and not UI elements so you need to assign it to the **Data** property of a **Path** UI element to present it visually.

Path element to present a geometry: Paths1\MainWindow.xaml

```
<UniformGrid Rows="2" Columns="2">
    <Path x:Name="path1" Stroke="SteelBlue" Fill="LightSteelBlue" Margin="4" />
</UniformGrid>
```

Describing a PathGeometry in XAML

```
<Path x:Name="path1"
    Stroke="SteelBlue" Fill="LightSteelBlue"
    Stretch="Fill" Margin="4">
    <Path.Data>
        <PathGeometry>
            <PathFigure StartPoint="10,0" IsClosed="True">
                <QuadraticBezierSegment Point1="0,12.5" Point2="10,25" />
                <LineSegment Point="80, 25"  />
                <LineSegment Point="90, 37.5" />
                <LineSegment Point="90, 25" />
                <QuadraticBezierSegment Point1="100,12.5" Point2="90,0" />
            </PathFigure>
        </PathGeometry>
    </Path.Data>
</Path>
```

Use the **Stretch** property to fit available space or to a fixed **Width** and **Height**. Since **PathGeometry** is object-oriented which is good when data is deserialized from XAML content but not using code to generate it. You can instead use a **StreamGeometry** which is a functional-oriented interface to generate figures and segments. Call **Freeze** method to optimize geometry. A frozen geometry can be rendered faster but cannot be edited anymore.

Creating and presenting a geometry at runtime

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    var stream = new StreamGeometry();
    var context = stream.Open();
    context.BeginFigure(new Point(10, 0), true, true);
    context.QuadraticBezierTo(new Point(0, 12.5),
        new Point(10, 25), true, true);
    context.LineTo(new Point(80, 25), true, true);
    context.LineTo(new Point(90, 37.5), true, true);
    context.LineTo(new Point(90, 25), true, true);
    context.QuadraticBezierTo(new Point(100, 12.5),
        new Point(90, 0), true, true);
    context.Close(); stream.Freeze(); path1.Data = stream;
}
```

There is a value converter for **PathGeometry** where the geometry is described using a mini-language with commands and parameters. The string can contain a list of the following commands. Note that geometries are not UI elements thus can be shared as resources between multiple UI elements.

Using converter to create geometry from a string

```
<Path x:Name="path2" Stroke="SteelBlue" Fill="LightSteelBlue"
        Stretch="Fill" Margin="4"
        Data="M10,0Q0,12.5,10,25H80L90,37.5L90,25Q100,12.5,90,0Z"/>
```

Data for Path element

`F0`	*Use even-odd fill rule (default)*
`F1`	*Use none-zero fill rule*
`M x,y`	*Move to x,y position (starts a new figure)*
`L x,y`	*Draw a line from current position to x and y*
`C x1,y1 x2,y2 x3,y3`	*Draws a cubic bezier curve using 2 control points*
`S x1,y1 x2,y2`	*Use 2nd control point from prior curve as 1st control point*
`Q x1,y1 x2,y2`	*Draws a quadratic bezier curve using one control point*
`A rx,ry d p c x,y`	*Draws arc with radius, degree, part, clockwise and point*
`H x`	*Draws a horizontal line*
`V v`	*Draws a vertical line*
`Z`	*Close figure*

Geometry as a shared resource

```
<Window.Resources>
        <PathGeometry x:Key="g1">
            M10,0Q0,12.5,10,25H80L90,37.5L90,25Q100,12.5,90,0Z</PathGeometry>
</Window.Resources>
```

Using same geometry in multiple elements

```
<Path x:Name="path2" Stroke="SteelBlue" Fill="LightSteelBlue"
    Stretch="Fill" Margin="4" Data="{StaticResource g1}" />
<Path x:Name="path3" Stroke="Maroon" Fill="Red"
    Stretch="Fill" Margin="4" Data="{StaticResource g1}" />
```

When drawing multiple figures, the figures may overlap where the inner figure will be used to cut out the outer figure because of the default **FillRule** settings. The default **FillRule** is **EvenOdd** which counts the number of edges from left to right. Space between the odd and even edges will be filled while space between the even and odd edges will not be filled. You can see this in the next example where the outer rectangle is filled but the inner one is left empty. You can set it to **NonZero** instead. The **NoneZero** fill rule will assign a **+1** to paths where the y-axis increases and **–1** where the y-axis decreases. The interior of the edges where the count is not zero is filled. This rule allows better control on whether paths are filled or not by changing the direction of the geometric path. **FillRule** property is available on all geometries. If you use mini-language to create the geometry then use **F0** for **EvenOdd** and **F1** for **NonZero** instead as shown below.

The EvenOdd fill rule: Paths2\MainWindow.xaml

```
<UniformGrid Rows="2" Columns="3">
    <Path Stroke="Black" Fill="Gold" Data="M0,0V50H50V0Z M5,5V45H45V5Z" />
</UniformGrid>
```

Using the NonZero fill rule

```
<Path Stroke="Black" Fill="Gold" Data="F1M0,0V50H50V0ZM5,5V45H45V5Z" />
```

Changing the path direction

```
<Path Stroke="Black" Fill="Gold" Data="F1M0,0V50H50V0ZM5,5H45V45H5Z" />
```

Not all figures in the geometry may be complex. Rather than having to describe each figure in detail, you can use a **GeometryGroup** to group simple geometry into a more complex one. Geometries are very useful to generate scalable iconic drawings to be used for a **ContentControl** or in a **ControlTemplate**.

Using multiple geometries as one Path with GeometryGroup

```
<Path Stroke="Green" Fill="Aquamarine" Stretch="Fill">
    <Path.Data>
        <GeometryGroup FillRule="EvenOdd">
            <RectangleGeometry Rect="0,0,100,100" />
            <EllipseGeometry Center="50,50"
                RadiusX="35" RadiusY="35" />
        </GeometryGroup>
    </Path.Data>
</Path>
```

Presenting a Metro-like icon for a button using GeometryGroup

```
<Button Width="32" Height="32">
    <Path Stroke="Black" StrokeThickness="2" Opacity="0.35"
        Fill="Transparent" Stretch="Fill" Margin="4">
        <Path.Data>
            <GeometryGroup>
                <EllipseGeometry Center="0.5,0.5"
                    RadiusX="0.5" RadiusY="0.5" />
                <PathGeometry>M0.2,0.5H0.8
                    M0.5,0.2L0.2,0.5L0.5,0.8</PathGeometry>
            </GeometryGroup>
        </Path.Data>
    </Path>
</Button>
```

Rather than grouping multiple geometry elements together, it is possible to physically combine together into a single geometry where you can decide on how to handle verlapping area between geometries. You can use a **CombinedGeometry** element to combine two geometries. Of course the geometry can also be a **GeometryGroup** or another **CombinedGeometry** to combine more. Use the **GeometryCombineMode** property to configure the resulting geometry area of the combination.

GeometryCombineMode values

Union	*Area of both geometries including overlapping area is used*
Xor	*Area of both geometries except overlapping area will be used*
Intersect	*Only the overlapping area between the geometries will be used*
Exclude	*Area of the 1st geometry without any area of the 2nd geometry*

Combining geometries into one

```
<Path Stroke="Black" StrokeThickness="2"
      Fill="DarkOrchid" Stretch="Fill" Margin="4">
    <Path.Data>
        <CombinedGeometry GeometryCombineMode="Union">
            <CombinedGeometry.Geometry1>
                <EllipseGeometry Center="0.5,0.5"
                    RadiusX="0.5" RadiusY="0.5" />
            </CombinedGeometry.Geometry1>
            <CombinedGeometry.Geometry2>
                <EllipseGeometry Center="1.0,0.5"
                    RadiusX="0.5" RadiusY="0.5" />
            </CombinedGeometry.Geometry2>
        </CombinedGeometry>
    </Path.Data>
</Path>
```

Geometry not only can be presented visually but can be used to clip visual elements. Every UI element has a **Clip** property to assign a single geometry. All parts of the visual element that falls outside of the filled area of the geometry is not rendered.

Using geometry as a clipping region: Paths3\MainWindow.xaml

```
<Grid Background="Black">
    <Viewbox Stretch="Uniform" Margin="8">
        <MediaElement Stretch="Fill" Width="300" Height="300"
            Source="C:/WPFDEV/RSC/Paper Airplane.mp4">
            <MediaElement.Clip>
                <GeometryGroup>
                    <EllipseGeometry Center="150,150"
                        RadiusX="150" RadiusY="150" />
                    <EllipseGeometry Center="150,150"
                        RadiusX="50" RadiusY="50" />
                </GeometryGroup>
            </MediaElement.Clip>
        </MediaElement>
    </Viewbox>
</Grid>
```

2 Text & Documents

2.1 TextBlock

In the previous examples, you only used **TextBlock** to display a simple line of text. A TextBlock has an **Inlines** collection that can contain a different inline elements. When you assign a string to a TextBlock it will be created as a **Run** inline element. Following shows how to add multiple **Run** inline elements into a TextBlock.

The following shows multiple runs of text: Text1\MainWindow.xaml

```
<StackPanel>
     <TextBlock Margin="4" TextTrimming="WordEllipsis">
        <Run>This is a run of text!</Run>
        <Run>This is another run of text!</Run>
        <Run>This is the last run of text!</Run>
    </TextBlock>
</StackPanel>
```

The reason why we need multiple runs is that each run can be separately styled but still displayed as one block of text. You can first set the default style for the TextBlock. This would then be inherited by all inline elements if they do not set those properties explicitly. You can then decide the styling for each of the runs and set the properties appropriately. The following examples show mixing of different fonts, font sizes and style in the same block of text.

Default styling for TextBlock

```
<TextBlock Margin="4" FontFamily="Verdana"
    FontSize="12" TextWrapping="Wrap" TextAlignment="Justify">
    <Run>This is a run of text!</Run>
    <Run FontSize="16">This is another run of text!</Run>
    <Run FontFamily="Times new Roman" FontSize="22" FontStyle="Italic">
        This is the final run of text!
    </Run>
</TextBlock>
```

You can see that the **FontStyle** to **Italic** in the last run. While this is fine for a block of text, it would be a bit tedious to create a Run just to change a single word to italic. That is why we have **Bold**, **Italic** and **Underline** elements. However you cannot use these elements inside a Run since a run cannot contain other inline elements. This is why we have a **Span** that can contain any number of inline elements including other runs and spans. You can also use **LineBreak** inline element to insert line breaks in the text content.

Using spans and other inline elements

```
<TextBlock TextWrapping="Wrap"
     TextAlignment="Right" FontSize="14">
    <Span FontFamily="Georgia">
        This is a <Bold>formatted</Bold> span of text.
        I can use any <Underline>inline</Underline>
        elements in a <Italic>Span</Italic>.
        <LineBreak />
        <LineBreak />
    </Span>
    <Span FontFamily="Verdana">
        You can have multiple lines as well by
        inserting a <Bold><Italic>LineBreak</Italic></Bold>
        element.
    </Span>
</TextBlock>
```

Since **Bold**, **Italic** and **Underline** can also contain inline elements, you can create a combined style by nesting the elements as shown above. Use **Hyperlink** to add a hyperlink inline element. Note that it is only hosting the hyperlink so you need to still handle the Click event and open the web browser.

Creating a hyperlink in a text block

```
<TextBlock
    Margin="8" Width="160" FontSize="14"
    TextWrapping="Wrap" TextAlignment="Center">
    Click <Hyperlink NavigateUri=
           "http://www.windowsclient.net">here</Hyperlink>
        to visit the Microsoft WPF site.
</TextBlock>
```

If you want to want to contain other UI elements as an inline, you can then use **InlineUIContainer** inline element. This is like a decorator control that has only one **Child** which can be assigned any UI element. The following shows how we can insert an **Image** control as part of the text. Each inline is a class so you can create and access all inline elements from program code during runtime. All these elements will behave and function as they would when placed into any container in the window.

Inserting UI elements

```
<InlineUIContainer>
    <Image Width="32" Source="C:\WPFDEV\RSC\Icons\World.ico" />
</InlineUIContainer>
```

TextBlock is a lightweight control for displaying text content and should only be used for few lines of text. If you want to display entire documents, use document viewers that can display flow documents or XPS documents.

2.2 FlowDocument

Unlike **TextBlock**, a **FlowDocument** contains **Blocks** rather than **Inlines**. **Section**, **Paragraph**, **List** and **Table** are blocks that you can assign to a FlowDocument. A **Section** is used to grouping multiple blocks. A **Paragraph** is used for inline elements, **List** to create a list and **Table** to make a table. You need to use a document viewer control to view a FlowDocument since it is not a UI element but a resource. Use a **FlowDocumentScrollViewer** to contain the **FlowDocument**.

Viewing an embedded FlowDocument: Text2\MainWindow.xaml

```
<FlowDocumentScrollViewer
    x:Name="docViewer"
     Margin="4"
     BorderThickness="2"
     BorderBrush="SteelBlue">
    <FlowDocument
        IsOptimalParagraphEnabled="True"
        IsHyphenationEnabled="True">
        <Paragraph>
            A <Bold>FlowDocument</Bold> unlike a
            <Bold>TextBlock</Bold> can contain a list of
            <Italic>block elements</Italic> rather than
            a list of <Italic>inline elements</Italic>.
        </Paragraph>
        <Paragraph>
            To assign inline text elements, you must use
            The <Bold>Paragraph</Bold> block element. There
            are also other block elements that you can use
            to make a list or table.
        </Paragraph>
        <Paragraph>
            You can use a <Bold>Section</Bold> block to
            group a list of other blocks. Use the <Bold>List</Bold>
            block to create a list and <Bold>Table</Bold>
            block to make a table.
        </Paragraph>
        <List>
            <ListItem Foreground="Green">
                <Paragraph>This is item 1</Paragraph>
            </ListItem>
            <ListItem Foreground="Blue">
                <Paragraph>This is item 2</Paragraph>
            </ListItem>
            <ListItem Foreground="Red">
                <Paragraph>This is item 3</Paragraph>
            </ListItem>
        </List>
                    :
```

```
        <Table BorderThickness="1" BorderBrush="LightSteelBlue"
            TextAlignment="Center" CellSpacing="16" Padding="4">
            <TableRowGroup>
                <TableRow Background="SkyBlue">
                <TableCell><Paragraph>1</Paragraph></TableCell>
                <TableCell><Paragraph>2</Paragraph></TableCell>
                <TableCell><Paragraph>3</Paragraph></TableCell>
                </TableRow>
                <TableRow Background="Beige">
                <TableCell><Paragraph>4</Paragraph></TableCell>
                <TableCell><Paragraph>5</Paragraph></TableCell>
                <TableCell><Paragraph>6</Paragraph></TableCell>
                </TableRow>
            </TableRowGroup>
        </Table>
    </FlowDocument>
</FlowDocumentScrollViewer>
```

The previous viewer provides scrolling through the entire document but no paging. If you want paging use **FlowDocumentPageViewer** instead or use the more complete **FlowDocumentReader** that has a search text feature. To create multiple columns set the **BreakColumnBefore** property on any block to **true**. To force a new page, set the **BreakPageBefore** property to **true**.

Insert a column break

```
<Paragraph BreakColumnBefore="True">
    You can use a <Bold>Section</Bold> block to
    group a list of other blocks. Use the <Bold>List</Bold>
    block to create a list and <Bold>Table</Bold>
    block to make a table.
</Paragraph>
```

Inside inline content you can assign special blocks such as **Figure** and **Floater**. You can control the exact position of a **Figure** but it would not appear if there is not enough room to show it. Inline content will flow around the figure. You will not be able to set the exact position of a **Floater** but it always appears and will be part of the document flow.

Inserting a figure in a paragraph

```
<Paragraph>
        :
      <Figure Background="Azure"
        BorderThickness="1" BorderBrush="SteelBlue"
        Width="0.25 column" Height="0.25 page"
        HorizontalOffset="100" HorizontalAnchor="PageLeft">
        <Paragraph Foreground="Red">Here I Am!</Paragraph>
    </Figure>
```

Code to Print the FlowDocument

```
PrintDialog pd = new PrintDialog();
if (pd.ShowDialog() != true) return;
document.PageHeight = pd.PrintableAreaHeight;
document.PageWidth = pd.PrintableAreaWidth;
document.ColumnWidth = pd.PrintableAreaWidth;
document.PagePadding = new Thickness(50);
document.ColumnGap = 0;
IDocumentPaginatorSource dps = document;
pd.PrintDocument(dps.DocumentPaginator, "Text2");
```

There is a problem if you print the same document that you are viewing since you need to change the properties. One method is to save the properties to be changed to variables so that you can restore them back later on. Another way is to save the document and load it back into a new document for printing only. In this way the original document is not affected. You can convert FlowDocument to a FixedDocument easily by printing to the XPS printer driver.

2.3 XPS Document

XPS document is **FixedDocument** rather than **FlowDocument** since the pagination is already performed and what you see will be what you get when printed. To display an XPS document use **DocumentViewer** instead that was built for FixedDocument.

The FixedDocument viewer: Text3\MainWindow.xaml

```
<DocumentViewer Name="docViewer" />
```

To use XPS documents you need to add reference to **ReachFramework** library. Create an **XpsDocument** instance and pass in the instance and access mode to open the document. Note that the document should be kept open while it is being used. Call **Close** to close the document when you know you do not need it anymore.

Opening an XPS document

```
protected void Window_Loaded(object sender, EventArgs e) {
    XpsDocument document = new XpsDocument(
        @"C:\WPFDEV\RSC\01.xps", FileAccess.Read);
    docViewer.Document = document.GetFixedDocumentSequence();
}
```

3 Transformation & Animation

3.1 Transformations

UI elements support transformations. You can assign transformations to the element **LayoutTransform** or **RenderTransform** property. **RotateTransform** can be used to rotation by setting the **Angle** property. The element will then be rotated based on a pivot point by setting **RenderTransformOrigin**. RenderTransform can transform element out of allocated space since it is transform during render and not layout. Use **TransformGroup** to set multiple transformations to an element.

Rotate element within the layout: Transform1\MainWindow.xaml

```
<TextBlock Text="This is some text!"
    FontFamily="Georgia" FontSize="20" Margin="8"
    HorizontalAlignment="Center">
    <TextBlock.LayoutTransform>
        <RotateTransform Angle="30" />
    </TextBlock.LayoutTransform>
</TextBlock>
```

Rotating an element out of layout

```
<TextBlock Text="This is some text!" FontFamily="Georgia"
    FontSize="20" Margin="8" HorizontalAlignment="Center"
    RenderTransformOrigin="0.5,0">
        <TextBlock.RenderTransform>
        <RotateTransform Angle="30" />
    </TextBlock.RenderTransform>
</TextBlock>
```

Applying a group of transformations

```
<Button x:Name="button1" Content="Button1" HorizontalAlignment="Center">
    <Button.RenderTransform>
        <TransformGroup>
            <SkewTransform AngleX="10" AngleY="10" />
            <ScaleTransform ScaleX="2" ScaleY="2" />
            <TranslateTransform X="-40" Y="40" />
            <RotateTransform Angle="30" />
        </TransformGroup>
    </Button.RenderTransform>
</Button>
```

It will be tedious in code to work with a **TransformGroup** containing four transforms. Instead you should use **MatrixTransform** instead. This transform can be assigned a **Matrix** to apply all the transformation. **MatrixTransform** is more suited to be used from code because the values has to be calculated by calling methods rather than assigned. Always use an **Identity** static property to obtain a default matrix without transformations. Then call methods provided to generate new values.

Setting up a matrix and transform in code

```
var matrix = Matrix.Identity;
matrix.Skew(10, 10);
matrix.Scale(2, 2);
matrix.Translate(40, 40);
matrix.Rotate(30);
var tr = new MatrixTransform(matrix);
button2.RenderTransform = tr;
```

3.2 Storyboards

Elements are animated based on properties. We have different types of properties, so there are different types of animations. To animate a color you use **ColorAnimation** and **DoubleAnimation** to animate double values and **PointAnimation** to animate **Point** values. Everything else that does not have the correct type of animation like **enum** use **ObjectAnimation** instead. Only dependency properties can be animated. All WPF element properties are dependency properties. A storyboard can be used to store, manage and assign many animations for the same or different elements.

Animation example: Animation1\MainWindow.xaml

```
<Window x:Class="Animation1.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Loaded="Window_Loaded">
    <Window.Resources>
        <Storyboard x:Key="sb1"></Storyboard>
    </Window.Resources>
    <Grid x:Name="LayoutRoot" Background="White">
   </Grid>
</Window>
```

Basic animations have **From** and **To** property to specify the start and end value. Use **Duration** to represent the length of animation. Set **RepeatBehavior** property if you want to repeat an animation. Normally an animation stops or repeats from start but you can set the **AutoReverse** to reverse the animation from the end back to start before it stops or repeats. The storyboard helps to attach the animations to the correct element and property using the **TargetName** and **TargetProperty** attached properties.

TargetProperty is a path to the property starting from the **TargetName** element. Note you are not animating the color of the Grid but **Color** property of the brush assigned to the **Background** of the Grid. Since **Color** property is only available on a **SolidColorBrush**, you will need to specify the exact type of brush that the property belongs to by casting the property.

Animating the color of a brush on an element

```
<Storyboard x:Key="sb1">
    <ColorAnimation Storyboard.TargetName="LayoutRoot"
        Storyboard.TargetProperty="Background.(SolidColorBrush.Color)"
        Duration="00:00:04" From="Red" To="Blue"
        AutoReverse="True" RepeatBehavior="Forever" />
</Storyboard>
```

You can now write code to fetch the storyboard and use it to start the animation in the **Loaded** event handler. You can access it from the resource dictionary by using the key and then call **Begin** method to start all the animations inside the storyboard. Run the application to see the results.

Starting a storyboard: MainWindow.xaml.cs

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
    var sb1 = (Storyboard)FindResource("sb1");
    sb1.Begin();
}
```

You can now try another animation. Put the following button inside the Grid. It has a **RotateTransform** assigned to the **RenderTransform** property that has an **Angle** property that we wish to animate.

Element containing a transform to animate

```
<Button x:Name="btnOK" Width="96" Height="24" Content="OK"
    RenderTransformOrigin="0.5, 0.5">
    <Button.RenderTransform><RotateTransform /></Button.RenderTransform>
</Button>
```

Animating the Angle of a RenderTransform assigned to an element

```
<DoubleAnimation Storyboard.TargetName="btnOK"
    Storyboard.TargetProperty="RenderTransform.(RotateTransform.Angle)"
    Duration="00:00:04" From="0" To="360" RepeatBehavior="Forever" />
```

Animations do not stop even if you do not repeat them forever. By default at the end of an animation the animation would hold on to the end of the animated value. You can change this by setting **FillBehavior** property from **HoldEnd** to **Stop**. However this won't stop the animation if it is still repeating.

To really stop an ongoing animation or storyboard, you have to call the **Stop** method. Attach an event handler to OK button and write the code shown below to stop the storyboard. When animations are stopped the values are reset back.

Stopping a storyboard

```
private void btnOK_Click(object sender, RoutedEventArgs e) {
    var sb1 = (Storyboard)FindResources("sb1");
    sb1.Stop();
}
```

The previous property paths are a bit complex simply because the actual object that contains the dependency property that is being animated does not have a name. So you will need to go through the parent or ancestor that has a name to access the property. To simplify property path, name the exact object that contains the property to animate. You can then target those objects directly.

Naming non-UI elements

```
<Grid x:Name="LayoutRoot">
    <Grid.Background><SolidColorBrush
        x:Name="brush1" Color="White" /></Grid.Background>
    <Button x:Name="btnOK" Width="96" Height="24" Content="OK"
        RenderTransformOrigin="0.5, 0.5" Click="btnOK_Click">
        <Button.RenderTransform><RotateTransform
            x:Name="transform1" Angle="0" /></Button.RenderTransform>
    </Button>
</Grid>
```

Simple direct property paths

```
<ColorAnimation Storyboard.TargetName="brush1"
    Storyboard.TargetProperty="Color"
    Duration="00:00:04" From="Red" To="Blue"
    AutoReverse="True" RepeatBehavior="Forever" />
<DoubleAnimationStoryboard.TargetName="transform1"
    Storyboard.TargetProperty="Angle"
    Duration="00:00:04" From="0" To="360"
    RepeatBehavior="Forever" />
```

3.3 Event Triggers

Even though we did not write code to create any of the animations but we still had to write code to activate the storyboard. However we can start a storyboard from XAML by using event trigger. All UI elements has **Triggers** property that can be assigned a collection of triggers. A trigger contains a **TriggerAction** collection representing a list of actions to perform when the trigger is activated. **EventTrigger** is a type of trigger that can be activated by events. Assign the event to **RoutedEvent** property.

BeginStoryBoard is an action to activate storyboards. Assign the storyboard to the **Storyboard** property. You can now comment out code to access and activate the storyboard since this is now done in XAML by using the following elements. At the moment you can only control storyboards and play a sound file using event triggers. You cannot write custom actions because **TriggerAction** is not extensible.

Activate a storyboard when Window is loaded

```
<Window.Triggers>
    <EventTrigger RoutedEvent="Window.Loaded">
        <BeginStoryboard Storyboard="{StaticResource sb1}" />
    </EventTrigger>
</Window.Triggers>
```

Use **SoundPlayerAction** to play a sound file when an EventTrigger is activated by clicking on the button. Add any sound file from the Windows Media directory into the project. You can create a **Media** or **Sounds** folder in the project to add and organize your media files. Make sure that **Build Action** on the sound file is set to **Resource**.

Play a sound when button is clicked

```
<Button x:Name="btnOK"
     Width="96" Height="24" Content="OK">
    <Button.Triggers>
        <EventTrigger RoutedEvent="Button.Click">
            <SoundPlayerAction Source="Sounds/tada.wav" />
        </EventTrigger>
    </Button.Triggers>
        :
```

To stop or suspend a storyboard **BeginStoryboard** action must have a name. You can then use **PauseStoryboard** action or the **StopStoryboard** action by setting the **BeginStoryboardName** property. Use **ResumeStoryboard** to resume a storyboard that has been paused. Following example shows how to stop the storyboard when the user presses and releases the mouse button on the window.

Controlling a storyboard

```
<EventTrigger RoutedEvent="Window.Loaded">
    <BeginStoryboard x:Name="sbAction1"
        Storyboard="{StaticResource sb1}" />
</EventTrigger>
<EventTrigger RoutedEvent="Window.MouseUp">
    <StopStoryboard BeginStoryboardName="sbAction1" />
</EventTrigger>
```

Animation can target only one object and property but you can use a **Style** to assign animations to multiple objects. A Style can also contain resources and triggers which can then be applied to elements that binds to that style.

Storyboards and triggers in Style: Animation2\MainWindow.xaml

```
<Style x:Key="FlashStyle" TargetType="Control">
    <Style.Resources>
        <Storyboard x:Key="Flashing">
            <DoubleAnimation Storyboard.TargetProperty="Opacity"
                From="1" To="0.25" Duration="00:00:00.500"
                RepeatBehavior="Forever" AutoReverse="True" />
            </Storyboard>
    </Style.Resources>
    <Style.Triggers>
        <EventTrigger RoutedEvent="MouseEnter">
            <BeginStoryboard x:Name="BeginFlashing"
                Storyboard="{StaticResource Flashing}" />
        </EventTrigger>
            <EventTrigger RoutedEvent="MouseLeave">
                <StopStoryboard BeginStoryboardName="BeginFlashing"/>
        </EventTrigger>
    </Style.Triggers>
</Style>
```

Assigning style to different element types

```
<StackPanel>
     <Label Style="{StaticResource FlashStyle}"
        Content="Here I Am!" Margin="4" FontSize="20"
        HorizontalAlignment="Center"/>
    <Button Style="{StaticResource FlashStyle}"
        Margin="4"  Width="96" Height="24" Content="Button1" />
    <Button Style="{StaticResource FlashStyle}"
        Margin="4"  Width="96" Height="24" Content="Button2" />
</StackPanel>
```

3.4 Cascading Animations

Sometimes you need to start an animation but only after another animation has been completed. In this case, you need to create two separate storyboards and assign a handler to **Completed** event. You can then start off the second storyboard when the first storyboard completes.

Elements to be animated: Animation3\MainWindow.xaml

```
<Canvas>
    <Ellipse x:Name="shape1" Canvas.Top="0"
         Width="32" Height="32" Fill="Red" Margin="4"/>
    <Ellipse x:Name="shape2" Canvas.Top="40"
         Width="32" Height="32" Fill="Blue" Margin="4"/>
</Canvas>
```

Separate storyboards & Completed event

```
<Window.Resources>
    <Storyboard x:Key="sb1"
        Completed="sb1_Completed">
        <DoubleAnimation Storyboard.TargetName="shape1"
            Storyboard.TargetProperty="(Canvas.Left)"
            Duration="00:00:03"
            From="0" To="200"
            RepeatBehavior="2x"
            AutoReverse="True" />
    </Storyboard>
    <Storyboard x:Key="sb2"
        Completed="sb2_Completed">
        <DoubleAnimation Storyboard.TargetName="shape2"
            Storyboard.TargetProperty="(Canvas.Left)"
            Duration="00:00:03"
            From="0" To="200"
            RepeatBehavior="2x"
            AutoReverse="True" />
    </Storyboard>
</Window.Resources>
```

Event handlers to start and stop storyboards

```
Storyboard sb1;
Storyboard sb2;

private void Window_Loaded(object sender, RoutedEventArgs e) {
    sb1 = (Storyboard)FindResource("sb1");
    sb2 = (Storyboard)FindResource("sb2");
    sb1.Begin();
}

private void sb1_Completed(object sender, EventArgs e) {
    sb1.Stop(); sb2.Begin();
}

private void sb2_Completed(object sender, EventArgs e) {
    sb2.Stop(); sb1.Begin();
}
```

But if the previous animation is what you are trying to achieve then you could have done the animations using one storyboard and there is no need to have a **Completed** event handler. An animation does not have to begin immediately when you start the storyboard. There is a **BeginTime** property where you can delay the animation to a specific time after storyboard has started. And if you wish to stop animations when they are completed, change **FillBehavior** to **Stop**.

Using BeginTime and FillBehavior

```
<Storyboard x:Key="sb1">
    <DoubleAnimation
        Storyboard.TargetName="shape1"
        Storyboard.TargetProperty="(Canvas.Left)"
        Duration="00:00:03"
        From="0" To="200"
        RepeatBehavior="2x"
        AutoReverse="True" />
    <DoubleAnimation
        Storyboard.TargetName="shape2"
        Storyboard.TargetProperty="(Canvas.Left)"
        BeginTime="00:00:12"
        Duration="00:00:03"
        From="0" To="200"
        RepeatBehavior="2x"
        AutoReverse="True" />
</Storyboard>
```

A storyboard also has **BeginTime**, **Duration**, **RepeatBehavior** and **FillBehavior** properties. In the following example, storyboard will only start animating 4 seconds after it is started. The animations will run for 16 seconds but the storyboard will run for 28 seconds, so there will be another 4 seconds before it repeats and the entire animations are repeated forever.

Setting animation properties on Storyboard

```
<Storyboard x:Key="sb1"
    BeginTime="00:00:04"
    Duration="00:00:28"
    RepeatBehavior="Forever">
            :
</Storyboard>
```

You may leave out the **From** if you wish to use the current value rather than a fixed value. This allows different element to use the same animation but having a different starting value. However if you use **To** then even though the starting value can be different the ending value is always the same unless you replace it with a **By** instead. A **By** animation value is always relative from the start value and not a fixed value. Linear animations are also very boring since direction and animation speed remains the same throughout. To create fluid animations, you may attach easing objects to the **EasingFunction** property.

Using easing objects

```
<DoubleAnimation
    Storyboard.TargetName="shape1"
    Storyboard.TargetProperty="(Canvas.Left)"
    Duration="00:00:03" By="200" RepeatBehavior="2x" AutoReverse="True">
    <DoubleAnimation.EasingFunction>
        <BounceEase EasingMode="EaseOut"
            Bounciness="2" Bounces="5"/>
    </DoubleAnimation.EasingFunction>
</DoubleAnimation>
<DoubleAnimation
    Storyboard.TargetName="shape2"
    Storyboard.TargetProperty="(Canvas.Left)"
    BeginTime="00:00:12" By="200" Duration="00:00:03" RepeatBehavior="2x"
    AutoReverse="True">
    <DoubleAnimation.EasingFunction>
        <QuadraticEase EasingMode="EaseOut" />
    </DoubleAnimation.EasingFunction>
</DoubleAnimation>
```

3.5 Using Timers

You can also perform automated operations based on time. A **DispatcherTimer** has **Tick** event where you can assign a handler that will be called repeatedly based on an **Interval** unless it is disabled or stopped. The event handler will be executed using UI thread so you can update the UI in the handler. Alternatively you can use a separate thread to perform operations so that they do not affect the UI by using a system **Timer** object instead. However you cannot update UI elements from a worker thread. UI elements can only be updated using a dispatcher thread that is accessible from all WPF elements. You can then use the **Dispatcher.Invoke** method to pass a delegate that will be executed by the dispatcher thread.

Using DispatcherTimer: Timers1\MainWindow.xaml.cs

```
int counter = 0;
DispatcherTimer timer;
public MainWindow() {
    InitializeComponent();
    timer = new DispatcherTimer();
    timer.Interval = TimeSpan.FromSeconds(0.25);
    timer.Tick += timer_Tick; timer.Start();
}

void timer_Tick(object sender, EventArgs e) {
    if (++counter == 100) timer.Stop();
    txtCounter.Text = counter.ToString();
}
```

Timer using worker thread: Timers2\MainWindow.xaml.cs

```
private Timer timer;
private int counter;
private Action updateAction;

public MainWindow() {
    InitializeComponent();
    int startTime = 0;  // start immediately
    int timerTick = 250; // 0.25 seconds
    timer = new Timer(Timer_Tick, null, startTime, timerTick);
    updateAction = UpdateCounter()
}

public void UpdateCounter() {
    txtCounter.Text = counter.ToString();
}

public void Timer_Tick(object state) {
    if (++counter == 100) timer.Dispose();
    Dispatcher.Invoke(UpdateAction);
}
```