Shapes, Brushes and Transforms

2D Graphics in WPF

Agenda

- WPF Graphic overview
- Shapes
- Brushes and Pens
- Transforms

Design Goals

- Make it **easy** to use graphics in your application
- Make it easy to exploit the power of your **graphics hardware**
- To integrate all kind of visuals into **one graphical system**

Rendering System

- For a graphics engine there are 2 different ways to provide cooperative rendering (*the process of combining multiple shapes or images together to form the final output*):
  - Clipping
    - Each element gets its own box of space, and all the rendering is confined to that box
    - Is simple and fast but have limitations
  - Composition
    - Allows elements to paint on top of each other – from back to front
    - Support things like transparency and irregular shaped elements but requires more computing power

- GDI/User32 (the “old” windows) uses clipping
- WPF uses composition

WPF's Composition System

- Is very different from how Windows has traditionally worked
  - and it is crucial to enabling the creation of high-quality visuals
- WPF's composition model supports elements of any shape, and allows them to overlap
  - It also allows elements to have any mixture of partially and completely transparent areas
  - This means that any given pixel on-screen may have multiple contributing visible elements
  - WPF uses anti-aliasing around the edges of all shapes
    - This reduces the jagged appearance that simpler drawing techniques can produce on-screen, resulting in a smooth-looking image
    - The composition engine allows any element to have a transformation applied before composition
- WPF's composition engine makes use of the capabilities of modern graphics cards to accelerate the drawing process
  - Internally, it is implemented on top of Direct3D

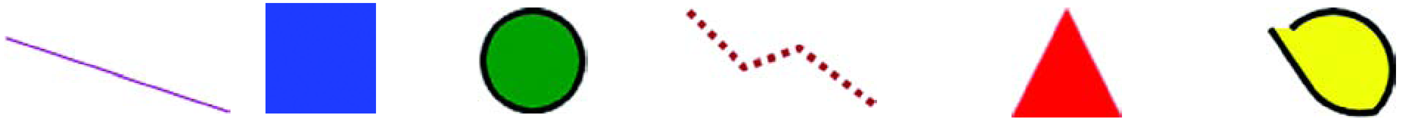
Resolution, coordinates, and "pixels"

- The default units of measurement in a WPF application is **device-independent pixels**
- WPF defines a device-independent pixel as **1/96th of an inch**
- If you specify the width of a shape as 96 pixels, this means that it should be exactly 1 inch wide
  - WPF will use as many physical pixels as are required to fill 1 inch. For example, a high-resolution laptop screens may have a resolution of 150 pixels per inch. So, if you make a shape's width 96 "pixels," WPF will render it 150 physical pixels wide
- This support for scaling graphics means that there is no fixed relationship between the coordinates your application uses and the pixels on-screen
  - A transform may be applied automatically to your whole application if it is running on a high-DPI display
- WPF allows you to use fractional double values when supplying a value in device-independent units

The Basic Building Blocks

The Basic Building Blocks in 2D graphics are:

- Geometries (Shapes)
  - Everything breaks down into a series of geometries that we render



- Brushes / **Fill**
  - Are used to fill the interior of a geometry
- Pens / **Stroke**
  - Are used to draw the outline of a geometry

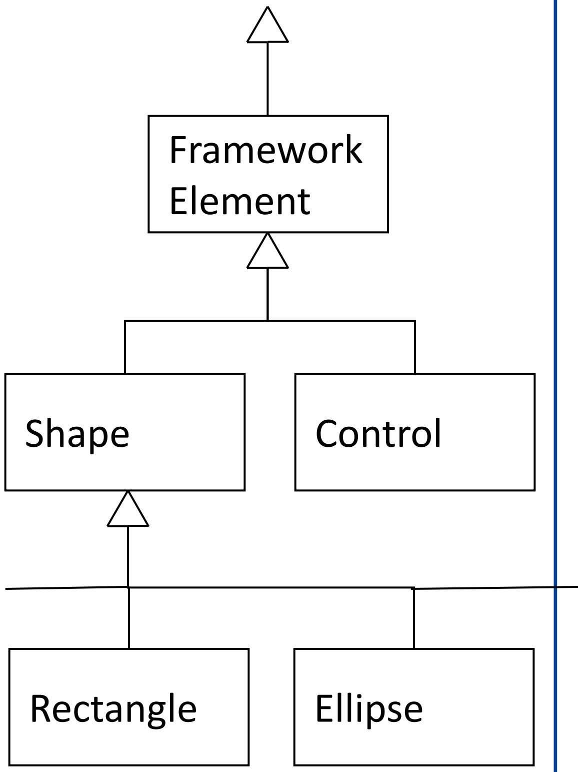
SHAPES

Drawing with Shapes

- Shapes are FrameworkElements that represent drawings
- Shapes add object identity, interactivity and styling to drawings

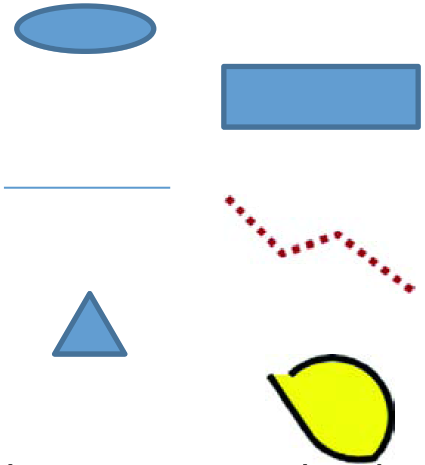
In other words

- The Shape class and its derivations form a set of classes that work much like controls
  - You can define them, set their sizes, locations, colors, and so on, as you would with a TextBox, and they interact with layout as a control does, supporting styles and events, and so on



Shapes

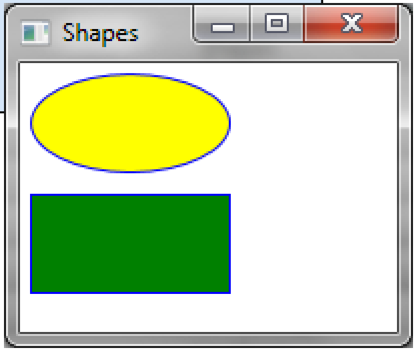
- Shapes are objects in the user interface tree that provide the basic building blocks for high-level drawing
- The different shapes:
  - Ellipse
  - Rectangle
  - Line
  - Polyline
  - Polygon
  - Path
    (the Path class supports both open and closed shapes with any mixture of straight and curved edges)



Rectangle and Ellipse

- Set the familiar Height and Width to define the size of your shape, and then
- Set the Fill or Stroke property (or both) to make the shape visible

```
<StackPanel>
  <Ellipse Fill="Yellow" Stroke="Blue"
           Height="50" Width="100" Margin="5"
           HorizontalAlignment="Left“ />
  <Rectangle Fill= "Green" Stroke="Blue"
             Height="50" Width="100" Margin="5“
             HorizontalAlignment="Left“ />
</StackPanel>
```



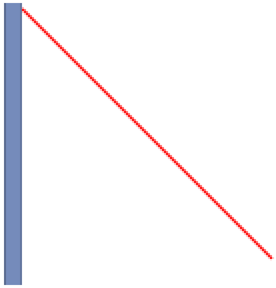
Line

- The Line shape represents a straight line that connects one point to another
- The starting and ending points are set by four properties:

  X1 and Y1 (for the first point)

  X2 and Y2 (for the second point)
- E.g.

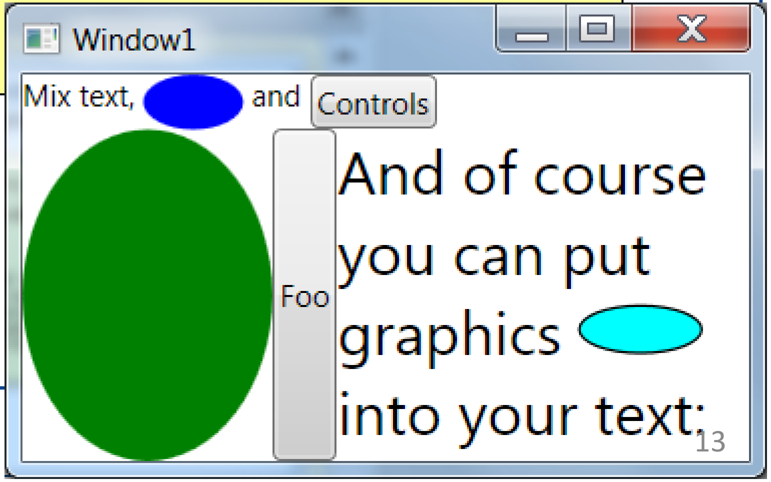
  <Line Stroke= "Red" X1="0" Y1="0" X2="100" Y2="100" />
- The Fill property has no effect for a line
  - You must set the Stroke property



Integration

- Graphical elements can be integrated into any part of your user interface

```
<DockPanel>
  <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
    <TextBlock Text="Mix text, " />
    <Ellipse Fill="Blue" Width="40" />
    <TextBlock Text=" and " />
    <Button>Controls</Button>
  </StackPanel>
  <Ellipse DockPanel.Dock="Left" Fill="Green" Width="100" />
  <Button DockPanel.Dock="Left">Foo</Button>
  <TextBlock FontSize="24" TextWrapping="Wrap">
    And of course you can put graphics <Ellipse Fill="Cyan"
    Width="50" Height="20" /> into your text:
  </TextBlock>
</DockPanel>
```



And of course you can put graphics into your text:

Drawing Object Model

- **You add objects representing graphical shapes to the tree of user interface elements**
- Shape elements are objects in the UI tree like any other
  - so your code can modify them at any time
- **If you change some property that has a visual impact**
  (such as the size, location, or color)
  WPF will automatically update the display

```
<Canvas x:Name="mainCanvas">
    <Ellipse Canvas.Left="10" Canvas.Top="30" Fill="Indigo"
             Width="40" Height="20" />
    <Ellipse Canvas.Left="20" Canvas.Top="40" Fill="Blue"
             Width="40" Height="20" />
    <Ellipse Canvas.Left="30" Canvas.Top="50" Fill="Cyan"
             Width="40" H
    <Ellipse Canvas.Left=
             Width="40" H
    <Ellipse Canvas.Left=
             Width="40" H
  </Canvas>
```

```
public partial class MainWindow : Window {
   public MainWindow() : base(  ) {
      InitializeComponent(  );
      mainCanvas.MouseLeftButtonDown += OnClick;
   }

   private void OnClick(object sender, RoutedEve
      Ellipse r = e.Source as Ellipse;
      if (r != null) {
         r.Width += 10;
```

Demo: 02ChangeItem

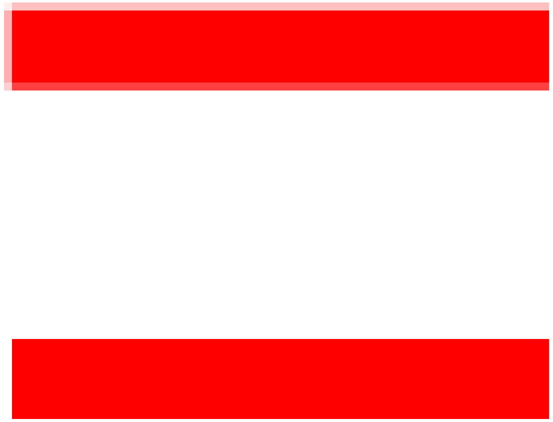
Pixel Snapping

- The ratio of pixels between different DPI settings is rarely a whole number
  - 50 pixels at 96 dpi become 62.4996 pixels on a 120-dpi monitor
- There’s no way to place an edge on a point that’s between pixels
- WPF compensates by using anti-aliasing
  - For example, when drawing a red line that’s 62.4992 pixels long, WPF might fill the first 62 pixels normally and then shade the sixty-third pixel with a value that’s in between the line color (red) and the background
- However, there’s a catch
  - If you’re drawing straight lines, rectangles, or polygons with square corners, this automatic anti-aliasing can introduce a tinge of blurriness at the edges of your shape

Anti-aliasing

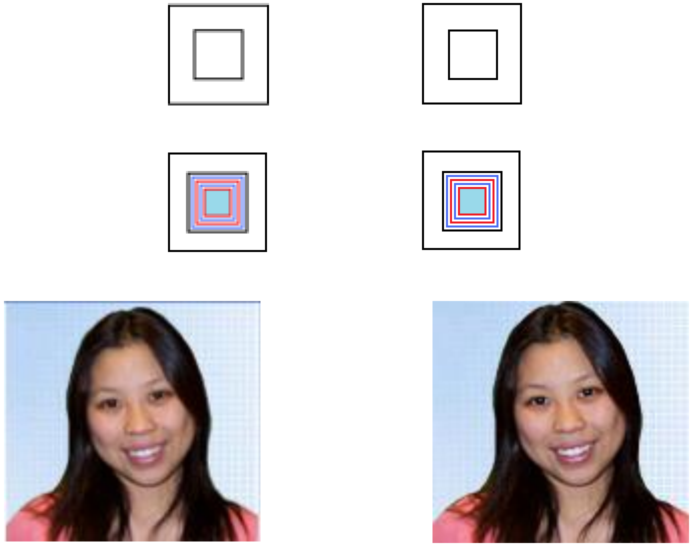
Pixel Snapping

- The fuzzy edge issue isn’t necessarily a problem
  - depending on the type of graphic you’re drawing, it might look quite normal
- If you don’t want this behavior
  - you can tell WPF not to use anti-aliasing for a specific shape
  - WPF will then round the measurement to the nearest device pixel
- You turn on this feature, pixel snapping, by setting the SnapsToDevicePixels property of a UIElement to true



Layout Rounding

- Layout Rounding is an alternative to Pixel Snapping
- Motivation:
  - Sub-pixel position and sizing can cause blurriness (right side images have rounded layout)



Layout Rounding

- Not a graphics feature – a Layout Feature
- How is it different from Pixel Snapping?
  - Layout Rounding changes *both the position and the size* of elements
  - Pixel Snapping is difficult to use and doesn't always work
- OFF by default on WPF
- You enable Layout Rounding on the element
  - `UIElement.UseLayoutRounding = True`

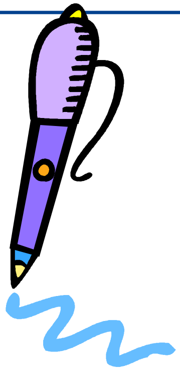
  Or on the Panel (all children will use the parents value)
  - <Grid UseLayoutRounding="True">

BRUSHES AND PENS

Brushes / Fill

- Areas (the inside of a geometry) are painted (Filled) with a brush
- Many brush types are available
  - SolidColorBrush
    - is the single-color Brush - Easy to use
  - LinearGradientBrush and RadialGradientBrush.
    - These allow the color to change over the surface of a shape
  - the ImageBrush
    - create brushes based on images
  - DrawingBrush
    - uses a scalable drawing
  - VisualBrush
    - lets you take any visual tree and use that as a brush to paint some other shape

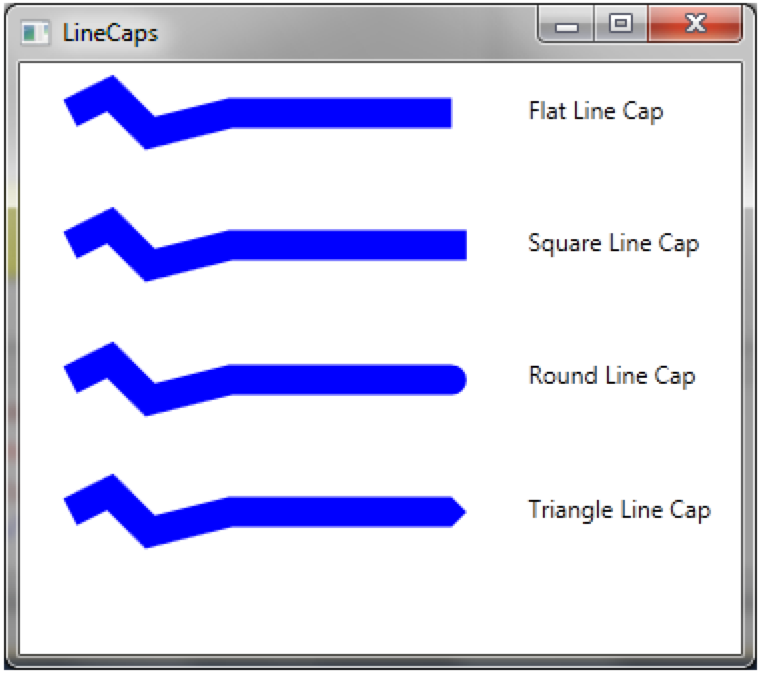
Pens / Stroke

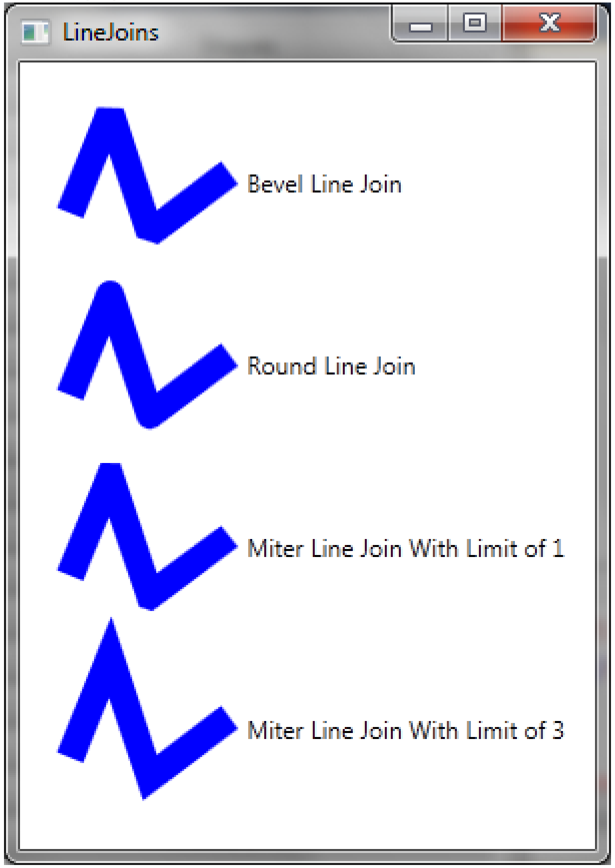


- Pens are used to draw the outline of a geometry
- A pen is really just an augmented brush
  - When you create a Pen object, you give it a Brush to tell it how it should paint onto the screen
  - The Pen class just adds information like line thickness, dash patterns, and endLineCap details
- Note:
  On Shape derived elements we set the pen properties on the containing element – we do not create the Pen explicit

Line Caps and Line Joins

- When drawing with the Line and Polyline shapes, you can choose how the starting and ending edge of the line is drawn using the StartLineCap and EndLineCap properties.





LineJoins
Bevel Line Join
Round Line Join
Miter Line Join With Limit of 1
Miter Line Join With Limit of 3

Dashes

- *A* dashed line is a line that is broken with spaces according to a pattern you specify with the `StrokeDashArray` property e.g.:

```
    <Polyline Stroke="Blue" StrokeThickness="8"
              StrokeDashArray="1 2"
              Points="10,30 60,0 90,40 120,10 350,10">
    </Polyline>
```

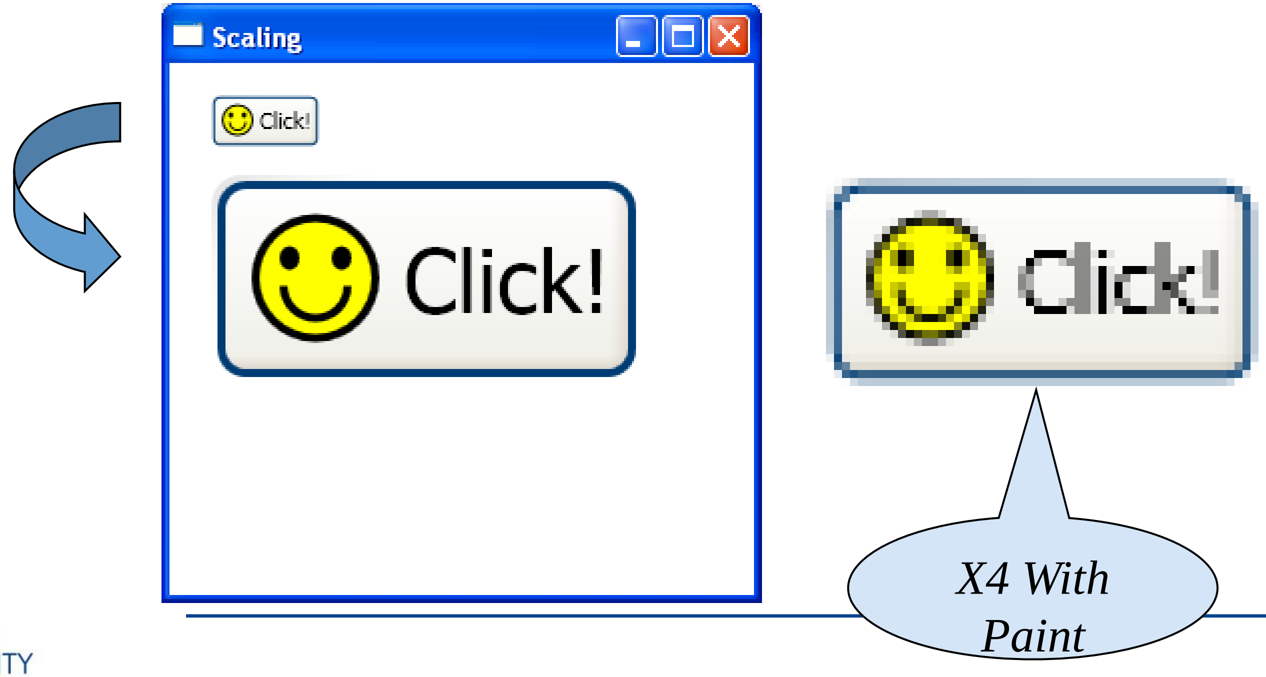
- These values are interpreted relative to the thickness of the line
  - So if the line is 8 units thick:
    - the solid portion is 8 x 1 = 8 units
    - And the blank portion of 8 x 2 = 16 units
- The line repeats this pattern for its entire length

TRANSFORMS

Scaling and rotation

- Everything in the UI can be transformed
  - Because it is built into the underlying composition engine

```
<Button>
  <Button.LayoutTransform>
    <ScaleTransform ScaleX=“4" ScaleY=“4" />
  </Button.LayoutTransform>
  ... as before ...
</Button>
```

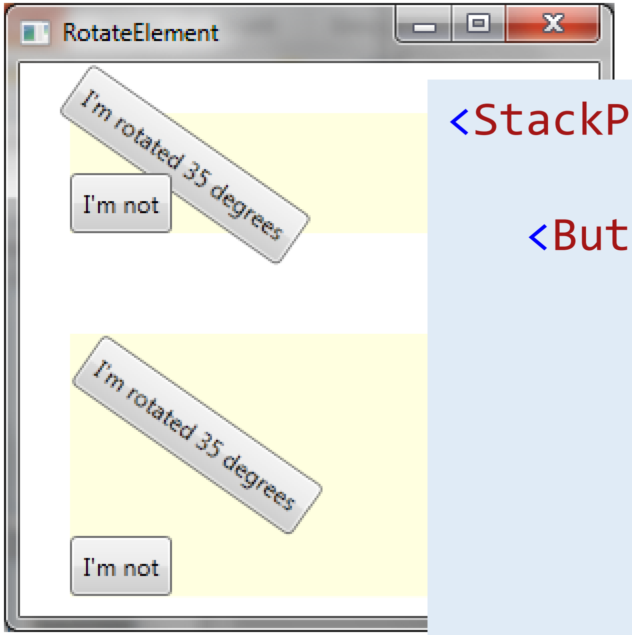


X4 With Paint

Transform Classes

- Alters the way a shape or element is drawn by quietly shifting the coordinate system it uses
- Transforms are represented by classes that derive from the abstract System.Windows.Media.Transform class
- TranslateTransform:
  - Displaces your coordinate system by some amount
- RotateTransform
  - The shapes you draw normally are turned around a center point you choose
- ScaleTransform
  - Scales your coordinate system up or down
- SkewTransform …

RotateTransform



```
<StackPanel  Margin="25"
             Background="LightYellow">
   <Button Padding="5"
           HorizontalAlignment="Left">
       <Button.RenderTransform>
          <RotateTransform Angle="35"
                           CenterX="45”
                           CenterY="5" />
       </Button.RenderTransform>
       <Button.Content>I'm rotated 35 degrees
       </Button.Content>
    </Button>
    <Button Padding="5"
            HorizontalAlignment="Left">
            I'm not
    </Button>
    </StackPanel>
```

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

References & Links

- WPF Graphics on MSDN: https://msdn.microsoft.com/en-us/library/ms742562(v=vs.100).aspx
- Color Model: http://www.easyrgb.com/
  (AU blue: R 3, G 66, B 142)
- Choose color palettes from:
  - http://kuler.adobe.com
  - http://colourlovers.com
- Vischeck simulates colorblind vision
  - http://vischeck.com/