EE 356 **Fall 2018**
Animation in WPF

Animation is done by creating a sequence of images and placing them sequentially on the screen at regular time intervals to give the appearance of a single figure changing in time. In WPF the animation changes the *properties* of objects. The animation timer and sequential image display is done automatically after you set up the specifications. A good place to begin to understand how WPF animation work is Animation Overview from Microsoft. WPF animation can be done using XAML code but for this course and these notes nearly all of the animation will be done in code.

Animating Properties
As a simple example consider the following animation program that animates a blue rectangle when the button is clicked.

XAML

```
<Window x:Class="AnimateButton.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:AnimateButton"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="Button1" Content="Button" HorizontalAlignment="Left"
            Height="23" Margin="178,26,0,0" VerticalAlignment="Top" Width="135"
            Click="Button1_Click"/>
        <Rectangle x:Name="rectangle1" Width="200" Margin="148,194,169,43"
            StrokeThickness="10" Fill="Blue" />
    </Grid>
</Window>
```

Button Click Event

```
        private void Button1_Click(object sender, RoutedEventArgs e)
        {
            DoubleAnimation dblAni = new DoubleAnimation();
            dblAni.From = 0;
            dblAni.To = 360;
            dblAni.Duration = new Duration(TimeSpan.FromSeconds(3));
            dblAni.RepeatBehavior = RepeatBehavior.Forever;
            RotateTransform rt = new RotateTransform();
            rectangle1.RenderTransform = rt;
            rectangle1.RenderTransformOrigin = new Point(.5, .5);
            rt.BeginAnimation(RotateTransform.AngleProperty, dblAni);
        }
```

A rectangle shape is defined in the XAML code and filled with a blue color. The button click event defines a double animation that rotates the rectangle forever about its center.
Notes:
1. The term double animation means that the animation uses double precision variables.
2. The RotateTransform (rt) uses the BeginAnimation method to rotate the angle property.

3. The RenderTransformOrigin is set to 0.5, 0.5 which is the center of the shape. The coordinate system is taken to be from 0 to 1 across the shape and you need not deal with absolution pixel locations.

Here is another example that animates the opacity property of an ellipse.

XMAL

```
<Window x:Class="AnimateOpacity.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:AnimateOpacity"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="btnAnimate" Content="Animate" HorizontalAlignment="Left"
            Height="26" Margin="169,270,0,0" VerticalAlignment="Top" Width="179"
            RenderTransformOrigin="0.319,0.455" Click="btnAnimate_Click"/>
        <Ellipse x:Name="ell1" Fill="#FF00FF00" HorizontalAlignment="Left" Height="214"
            Margin="74,28,0,0" Stroke="Black" VerticalAlignment="Top" Width="370"/>
    </Grid>
</Window>
```

Button Click Event

```
        public MainWindow()
        {
            InitializeComponent();
            this.RegisterName(ell1.Name, ell1);
        }
        private Storyboard sBrd = new Storyboard();
        private void btnAnimate_Click(object sender, RoutedEventArgs e)
        {
            DoubleAnimation dblAni = new DoubleAnimation();
            dblAni.From = 1.0;
            dblAni.To = 0.0;
            dblAni.Duration = new Duration(TimeSpan.FromSeconds(3));
            dblAni.AutoReverse = true;
            dblAni.RepeatBehavior = RepeatBehavior.Forever;
            sBrd.Children.Add(dblAni);
            Storyboard.SetTargetName(dblAni, ell1.Name);
            Storyboard.SetTargetProperty(dblAni,
                     new PropertyPath(Ellipse.OpacityProperty));
            sBrd.Begin(this);
        }
```

The ellipse is initially green. The animation changes its opacity so that it gradually fades to the background.

Notes:

1. In the XAML code the fill color for the ellipse is set by `Fill="#FF00FF00"` where the first FF is the opacity number followed by the Red, Green, and Blue number where 00 is full off and FF is full on.

Animating Movement on a Path

Animating an object to move along a path is very similar to animating a property – in effect you are animating the x and y location properties for the object. To do so you need to create a path which is stored in the StoryBoard that the animation can follow. There are many ways to define a path but a straight forward method uses a *Bezier* curve which fits a polynomial or a set of polynomials through a given set of points to form a smooth path. In WPF you can make use of a class called `PolyBezierSegment`. To create a path you create an instance of `PolyBezierSegment` and load it with points. C# creates a polynomial to fit these points and that polynomial is used to find the x and y values of the path as needed for each movement of the object being animated.

In the following example an ellipse (circle) is created on a canvas and a path is defined from the top left of the canvas to the bottom right. When you click on the start button the circle moves along the path and repeats this behavior forever.

XAML

```
<Window x:Class="AnimationPathEx1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:AnimationPathEx1"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Border BorderThickness="3" BorderBrush="Black" HorizontalAlignment="Left"
                Height="215" Margin="22,23,0,0" VerticalAlignment="Top" Width="475">
            <Canvas Name="cnv1" ></Canvas>
        </Border>
        <Button x:Name="btnStart" Content="Button" HorizontalAlignment="Left"
            Height="43" Margin="181,255,0,0" VerticalAlignment="Top" Width="148"
            Click="btnStart_Click"/>
    </Grid>
</Window>
```

Note that the canvas (cnv1) is enclosed in a Border element which determines its size. The canvas allows us to reset the coordinate system so that 0,0 is the top left corner of the canvas - not the top left corner of the window.

The ellipse could be defined in XAML code but in this example it is defined and C# code. It is added as a child to the canvas. The canvas clip property is set to true so that the ellipse is clipped off if it goes beyond the boundary of the canvas.

The transform that we are using is the translate transform and we will apply it to both the x and y properties of the ellipse.

Main Window Setup Code

```
      public MainWindow()
        {
            InitializeComponent();
            cnv1.ClipToBounds = true;
            SolidColorBrush bluBrush = new SolidColorBrush(Color.FromRgb(0, 0, 255));
            Ellipse ell1 = new Ellipse();
            ell1.Height = 25;ell1.Width = 25;
            ell1.Fill = bluBrush;
            ell1.Margin = new Thickness(0, 0, 0, 0);
            cnv1.Children.Add(ell1);
            // The translate transform moves the ellipse.
            TranslateTransform animatedTranslateTransform =
                new TranslateTransform();
            // Register the transform's name with the page
            // so that it can be targeted by a Storyboard.
            cnv1.RegisterName("AnimatedTranslateTransform",
                                       animatedTranslateTransform);
            ell1.RenderTransform = animatedTranslateTransform;
        }
```

Button Click Event Code

```
        private void btnStart_Click(object sender, RoutedEventArgs e)
        {
            // Create the animation path.
            PathGeometry animationPath = new PathGeometry();
            PathFigure pFigure = new PathFigure();
            pFigure.StartPoint = new Point(0, 0);
            PolyBezierSegment pBezierSegment = new PolyBezierSegment();
            Point start = new Point(0, 0);
            Point end = new Point(445, 185);
            LoadPathPoints(pBezierSegment, start, end);
            pFigure.Segments.Add(pBezierSegment);
            animationPath.Figures.Add(pFigure);
            animationPath.Freeze();// Freeze the PathGeometry for better performance.
            // Create a DoubleAnimationUsingPath to move the ellipse horizontally along
            //   the path by animating its TranslateTransform.
            DoubleAnimationUsingPath translateXAnimation =
                new DoubleAnimationUsingPath();
            translateXAnimation.PathGeometry = animationPath;
            translateXAnimation.Duration = TimeSpan.FromSeconds(5);
            // Set the Source property to X.
            translateXAnimation.Source = PathAnimationSource.X;
            // Set the animation to target the X property
            Storyboard.SetTargetName(translateXAnimation,
                                     "AnimatedTranslateTransform");
            Storyboard.SetTargetProperty(translateXAnimation,
                new PropertyPath(TranslateTransform.XProperty));
            // Repeat the above procedure for y and vertical motion
            DoubleAnimationUsingPath translateYAnimation =
                new DoubleAnimationUsingPath();
            translateYAnimation.PathGeometry = animationPath;
            translateYAnimation.Duration = TimeSpan.FromSeconds(5);
            translateYAnimation.Source = PathAnimationSource.Y;
            Storyboard.SetTargetName(translateYAnimation,
                                       "AnimatedTranslateTransform");
            Storyboard.SetTargetProperty(translateYAnimation,
                new PropertyPath(TranslateTransform.YProperty));

            // Create a Storyboard to contain and apply both animations.
            Storyboard pathAnimationStoryboard = new Storyboard();
            pathAnimationStoryboard.RepeatBehavior = RepeatBehavior.Forever;
            pathAnimationStoryboard.Children.Add(translateXAnimation);
            pathAnimationStoryboard.Children.Add(translateYAnimation);
            // Start the animations.
            pathAnimationStoryboard.Begin(cnv1);

        }
```

Load the Bezier Points function

```
private void LoadPathPoints(PolyBezierSegment pBezierSegment, Point start, Point end)
   {
       double incrx = (end.X - start.X)/5;
       double incry = (end.Y - start.Y)/5;
       int i;
       double x, y;
       x = start.X;y = start.Y;
       for(i=0;i<6;i++)
           {
               pBezierSegment.Points.Add(new Point(x, y));
               x += incrx;
               y += incry;
           }
       }
   }
```

This function loads six points on a straight line into a Bezier segment

Key Frame Animation

Animations that we have looked at so far are limited to two target values and we have little control over the animations interpolation process. Key Frame animations can be used to overcome these obstacles. In Key Frame animation the target values are described using key frame objects and these are added together to form a KeyFrame collection. A Key Frame Animation transitions between key frame objects in the collection. This is somewhat analogous to a StoryBoard containing multiple StoryBoards. For those interested in KeyFrame animation the overview document *Key-Frame Animations Overview* is a good place to start.