WPF

Grafika i multimedia

Krzysztof Mossakowski
Wydział Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
http://www.mini.pw.edu.pl/~mossakow

Możliwości grafiki w WPF

- Zalety wykorzystania grafiki w WPF:
  - niezależność od rozdzielczości i urządzenia
  - zaawansowane możliwości graficzne
  - animacja
  - wykorzystanie wspomagania sprzętowego
- Dostępne elementy graficzne:
  - kształty dwuwymiarowe
  - geometrie dwuwymiarowe
  - efekty dwuwymiarowe
  - rysowanie grafiki trójwymiarowej
  - animacja
  - multimedia – obrazki, filmy, dźwięki

Renderowanie grafiki

- **Visual** jest abstrakcyjną klasą, z której dziedziczą wszystkie obiekty **FrameworkElement**
  - to jest jedna z podstawowych klas WPF odpowiadająca za renderowanie używanych elementów
  - jest odpowiedzialna za rysowania elementów, transformacje, obcinanie, sprawdzanie trafienia w element i obliczanie otaczającego prostokąta
  - nie wspomaga w żaden sposób obsługi zdarzeń, układu elementów, stylów, wiązania danych i globalizacji
  - obiekt **Visual** przechowuje dane do renderowania w postaci listy instrukcji grafiki wektorowej
    - wykorzystywane są cztery typy danych: grafika wektorowa, obrazek, kształt znaku w czcionce (*glyph*) i film

Renderowanie grafiki c.d.

- **DrawingVisual** – wykorzystywany do renderowania kształtów, obrazków i tekstu
  - może być wykorzystany do stworzenia własnego obiektu wizualnego
- **Viewport3DVisual** – połączenie między dwu- i trójwymiarową grafiką
  - konieczne jest zdefiniowanie obiektów klasy **Camera** i **Viewport**
- **ContainerVisual** – kontener dla obiektów klasy **Visual**
- **Drawing** – abstrakcyjna klasa podstawowa dla rysowania grafiki dwuwymiarowej
  - klasa bazowa dla **DrawingGroup**, **GeometryDrawing**, **GlyphRunDrawing**, **ImageDrawing** i **VideoDrawing**

Renderowanie grafiki c.d.

- **DrawingGroup** – operacje graficzne stosowane przed w następującej kolejności:
  1. **OpacityMask**
  2. **Opacity**
  3. **BitmapEffect**
  4. **ClipGeometry**
  5. **GuidelineSet**
  6. **Transform**
- **DrawingContext** – wypełnia zawartością graficzną obiekty **Visual**
  - można go otrzymać za pomocą metod **DrawingGroup.Open** i **DrawingVisual.RenderOpen**

Renderowanie

- Wyświetlane obiekty zawierają zbiór zserializowanych danych umożliwiających ich narysowanie
  - system (a nie aplikacja) jest odpowiedzialny za odpowiadanie na wszystkie żądania odrysowania zawartości okien
  - WPF jest w stanie wydajnie zoptymalizować odrysowywanie
- WPF używa grafiki wektorowej do renderowania
- WPF używa współrzędnych logicznych i automatycznie przeskalowuje scenę stosownie do rozdzielczości urządzenia, na którym następuje rysowanie
  - "logiczne piksele" mają wartość 1/96 cala, czyli są zgodne ze standardową rozdzielczością w Windows
  - wszystkie elementy graficzne i teksty zostaną automatycznie przeskalowane

Transformacje

- **Transform** to macierz 3x3**:**

M11	**M12**	0.0
M21	**M22**	0.0
OffsetX	**OffsetY**	0.1

- **RotateTransform, ScaleTransform, SkewTransform, TranslateTransform**
- Klasa **Transform** dziedziczy z **Animatable**, czyli może być animowana
- Wiele transformacji można zastosować do jednego elementu za pomocą **TransformGroup**

```
<Button Content="A Button"
        RenderTransformOrigin="0.5,0.5">
  <Button.RenderTransform>
    <RotateTransform
         x:Name="AnimatedRotateTransform"
         Angle="0" />
  </Button.RenderTransform>
  <Button.Triggers>
    <EventTrigger
          RoutedEvent="Button.Click">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation
              Storyboard.TargetName=
                 "AnimatedRotateTransform"
              Storyboard.TargetProperty=
                 "Angle"
              To="360" Duration="0:0:1"
              FillBehavior="Stop" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

Efekty bitmapowe

- Efekt bitmapowy pobiera **BitmapSource** i tworzy dla niego nowy **BitmapSource** z modyfikacją wynikającą z pożądanego efektu
- W WPF efekty mogą być określane w obiektach **Visual** jako właściwości
  - w takim przypadku **Visual** jest automatycznie konwertowany do **BitmapSource**
- Dostępne są następujące efekty:
  - **BlurBitmapEffect**
  - **OuterGlowBitmapEffect**
  - **DropShadowBitmapEffect**
  - **BevelBitmapEffect**
  - **EmbossBitmapEffect**
- Można także definiować własne efekty

Beveled Button

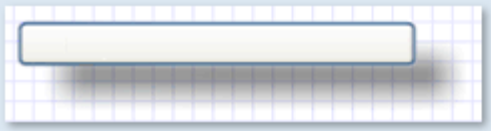
```
...
<BevelBitmapEffect BevelWidth="15"
EdgeProfile="CurvedIn" LightAngle="320"
Relief="0.4" Smoothness="0.4" />
...
```

Blurred Button

```
...
<BlurBitmapEffect Radius="10"
KernelType="Box" />
...
```

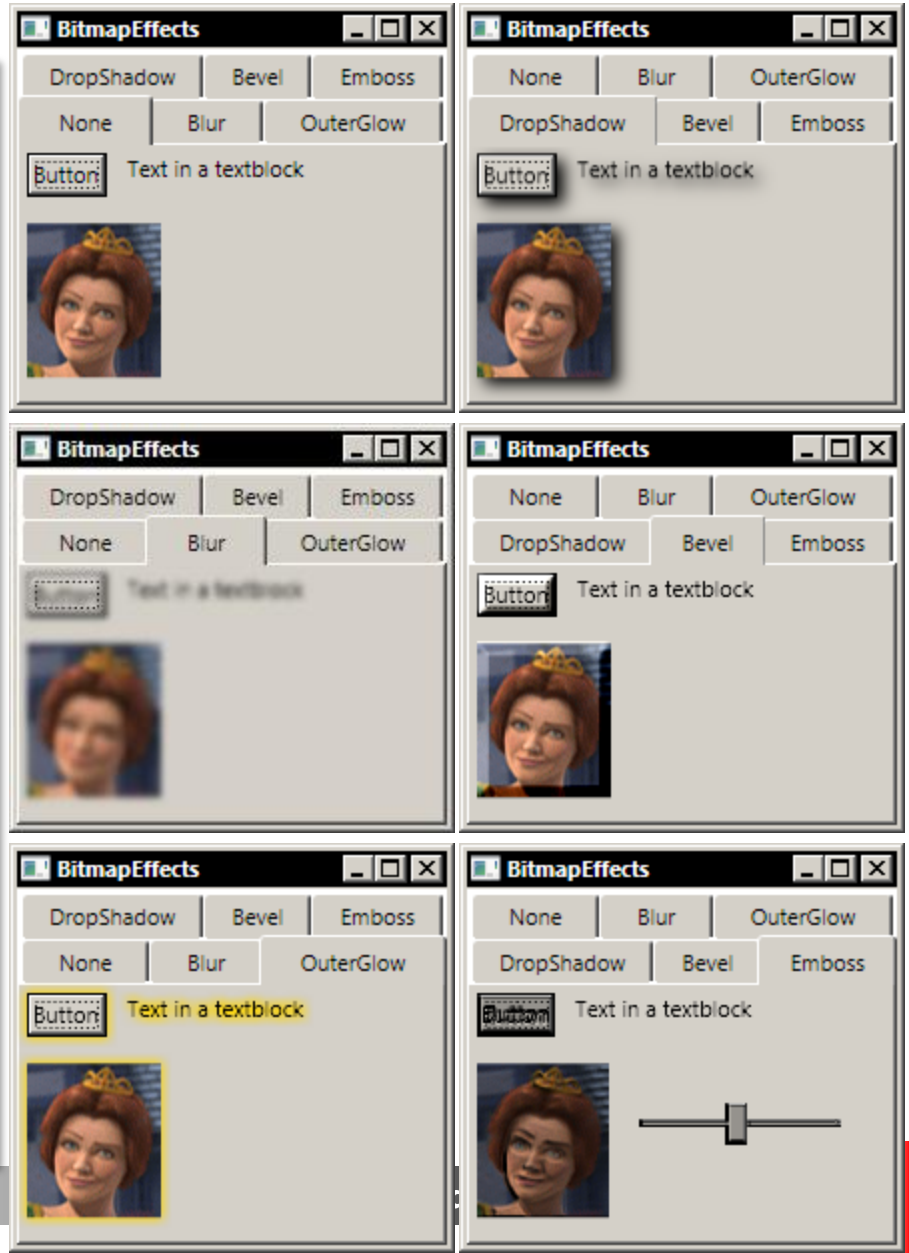
```
...
<OuterGlowBitmapEffect GlowColor="Blue"
Opacity="0.4" GlowSize="30" Noise="1" />
...
```

```
...
<DropShadowBitmapEffect Color="Black"
Direction="320" ShadowDepth="25" Softness="1"
Opacity="0.5" />
...
```



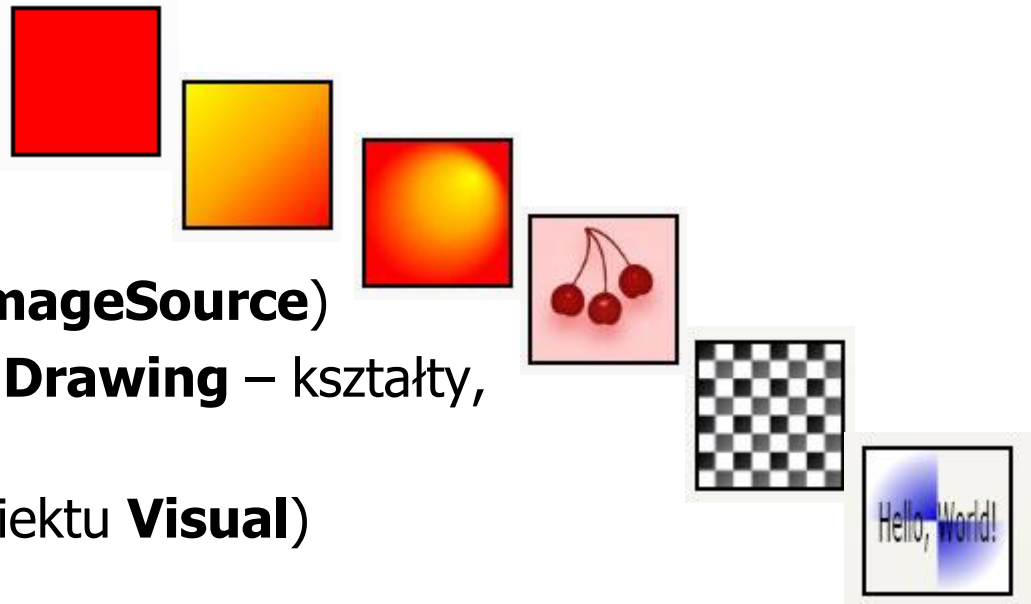
Przykład efektów

```
<TabControl>
  <TabItem Header="None">
    <Canvas>
      <Button>Button</Button>
      <TextBlock Canvas.Left="50">
        Text in a textblock</TextBlock>
      <Image Canvas.Top="35"
        Source="/Images/Fiona_67x77.gif"/>
    </Canvas>
  </TabItem>
  <TabItem Header="Blur">
    <Canvas>
      <Canvas.BitmapEffect>
        <BlurBitmapEffect/>
      </Canvas.BitmapEffect>
      <Button>Button</Button>
      <TextBlock Canvas.Left="50">
        Text in a textblock</TextBlock>
      <Image Canvas.Top="35"
        Source="/Images/Fiona_67x77.gif"/>
    </Canvas>
  </TabItem>
  [...]
</TabControl>
```



Pędzle

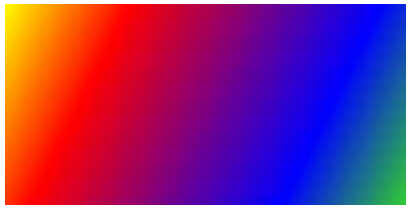
- Wszystkie to, co jest widoczne, zostało narysowane za pomocą pędzla
- Dostępne rodzaje pędzli:
  - **SolidColorBrush**
  - **LinearGradientBrush**
  - **RadialGradientBrush**
  - **ImageBrush** (używa **ImageSource**)
  - **DrawingBrush** (używa **Drawing** – kształty, obrazki, teksty, filmy)
  - **VisualBrush** (używa obiektu **Visual**)



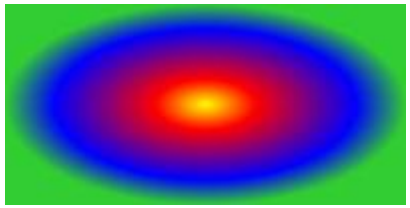
- Wspólne cechy pędzli:
  - **Opacity**
  - **Transform**, **RelativeTransform**
  - mogą być animowane
  - mogą być "zamrożone" (tzn. niemożliwa będzie ich modyfikacja)

Przykłady pędzli gradientowych

```
<Rectangle Width="200" Height="100">
  <Rectangle.Fill>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
      <GradientStop Color="Yellow" Offset="0.0" />
      <GradientStop Color="Red" Offset="0.25" />
      <GradientStop Color="Blue" Offset="0.75" />
      <GradientStop Color="LimeGreen" Offset="1.0" />
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

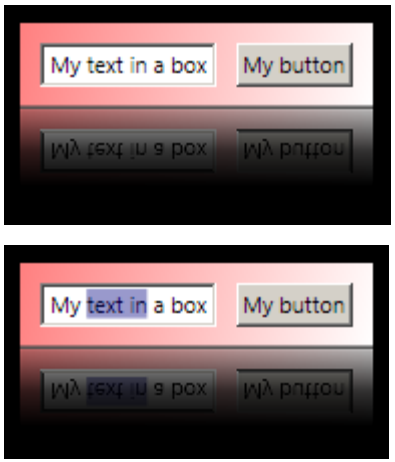


```
<Rectangle Width="200" Height="100">
  <Rectangle.Fill>
    <RadialGradientBrush GradientOrigin="0.5,0.5"
         Center="0.5,0.5" RadiusX="0.5" RadiusY="0.5">
      <GradientStop Color="Yellow" Offset="0" />
      <GradientStop Color="Red" Offset="0.25" />
      <GradientStop Color="Blue" Offset="0.75" />
      <GradientStop Color="LimeGreen" Offset="1" />
    </RadialGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```



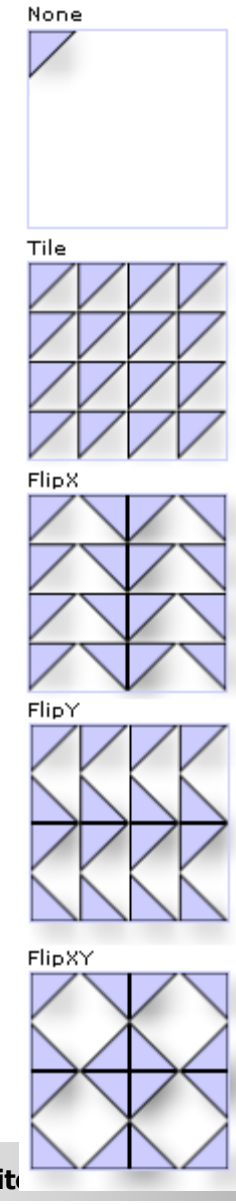
```
<StackPanel Canvas.Top="300" Canvas.Left="50">
  <Border Name="ReflectedVisual">[...]</Border>

  <Rectangle Height="1" Fill="Gray" HorizontalAlignment="Stretch" />
  <Rectangle  Height="{Binding Path=ActualHeight, ElementName=ReflectedVisual}"
              Width="{Binding Path=ActualWidth, ElementName=ReflectedVisual}">
    <Rectangle.Fill>
      <VisualBrush Opacity="0.75" Stretch="None"
           Visual="{Binding ElementName=ReflectedVisual}">
        <VisualBrush.RelativeTransform>
          <TransformGroup>
            <ScaleTransform ScaleX="1" ScaleY="-1" />
            <TranslateTransform Y="1" />
          </TransformGroup>
        </VisualBrush.RelativeTransform>
      </VisualBrush>
    </Rectangle.Fill>
    <Rectangle.OpacityMask>
      <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
        <GradientStop Color="#FF000000" Offset="0.0" />
        <GradientStop Color="#44000000" Offset="0.5" />
        <GradientStop Color="#00000000" Offset="0.9" />
      </LinearGradientBrush>
    </Rectangle.OpacityMask>
    <Rectangle.BitmapEffect>
      <BlurBitmapEffect Radius="1.5" />
    </Rectangle.BitmapEffect>
  </Rectangle>
</StackPanel>
```

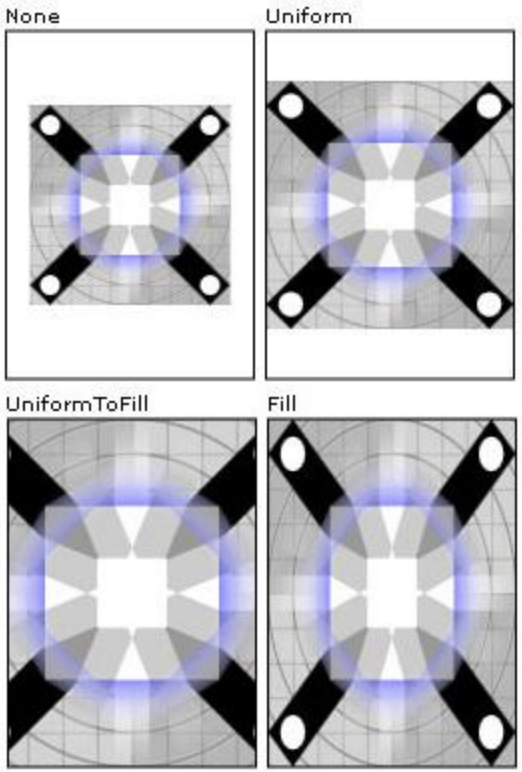


Przykłady TileBrush

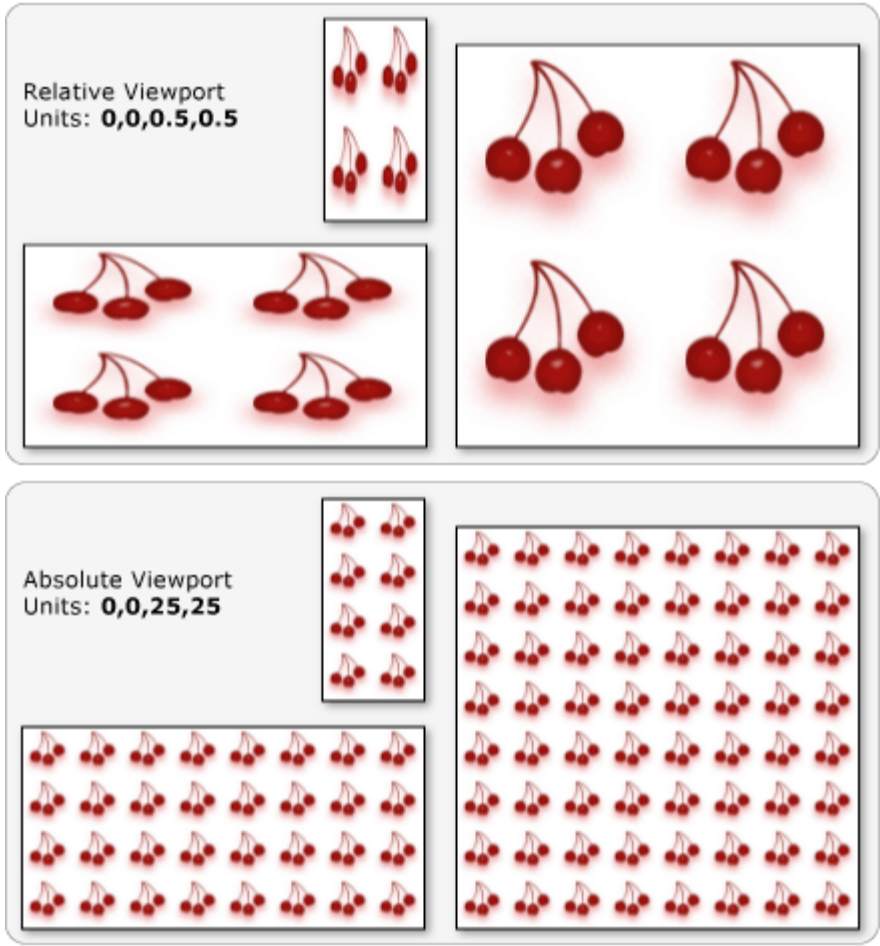
TileMode



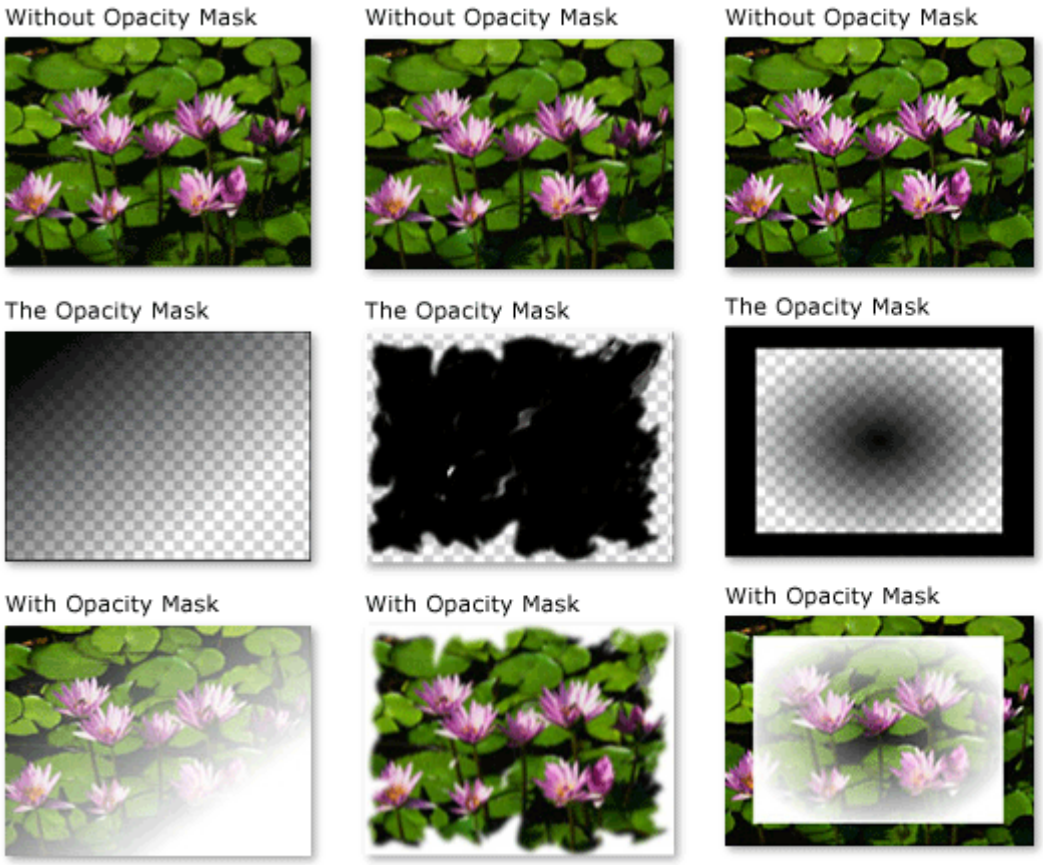
Stretch



Viewport, ViewportUnits

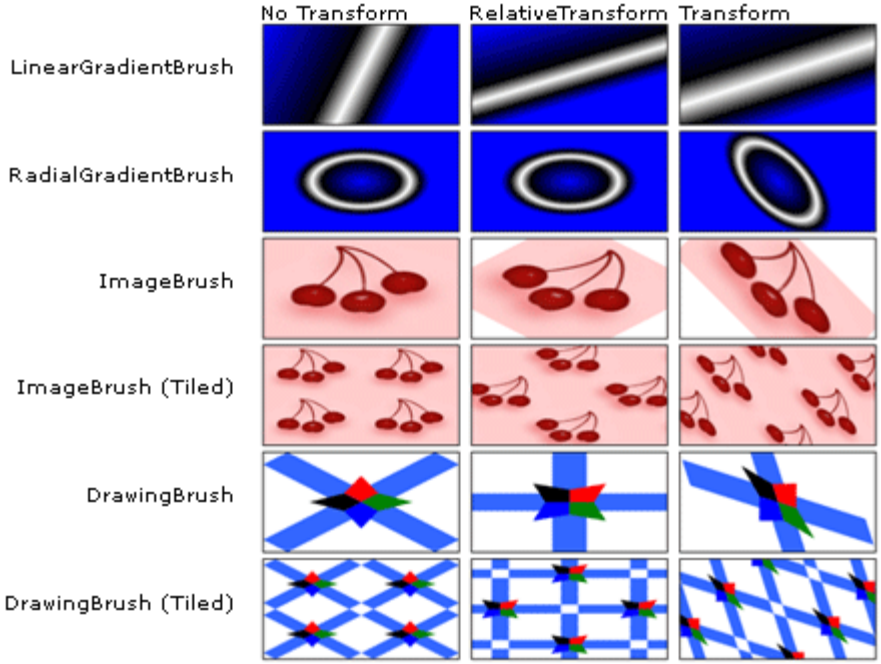


Maski przezroczystości



Transformacje pędzli

- W WPF wykorzystywany jest następujący algorytm:
  1. Przetworzenie zawartości pędzla
  2. Odwzorowanie pędzla na prostokąt transformacji 1x1
  3. Zastosowanie **RelativeTransform**, o ile jest zdefiniowana
  4. Odwzorowanie zawartości pędzla na docelową powierzchnię do narysowania
  5. Zastosowanie **Transform**, o ile jest zdefiniowana



Efekt powstały przez transformację o kąt 45 stopni z wykorzystaniem właściwości Transform i RelativeTransform

Kształty

- Klasa **Shape** jest klasą bazową dla wszystkich kształtów:
  - **Ellipse**, **Line**, **Path**, **Polygon**, **Polyline**, **Rectangle**
- Wspólne właściwości obiektów kształtów:
  - **Stroke** – sposób rysowania konturu
  - **StrokeThickness** – grubość konturu
  - **Fill** – sposób rysowania wnętrza
  - **Transform**
  - **Stretch**

```
<Polygon
    Points="0,0 0,1 1,1"
    Fill="Blue"
    Width="100"
    Height="100"
    Stretch="Fill"
    Stroke="Black"
    StrokeThickness="2" />
```

```
PointCollection myPointCollection =
    new PointCollection();
myPointCollection.Add(new Point(0, 0));
myPointCollection.Add(new Point(0, 1));
myPointCollection.Add(new Point(1, 1));

Polygon myPolygon = new Polygon();
myPolygon.Points = myPointCollection;
myPolygon.Fill = Brushes.Blue;
myPolygon.Width = 100;
myPolygon.Height = 100;
myPolygon.Stretch = Stretch.Fill;
myPolygon.Stroke = Brushes.Black;
myPolygon.StrokeThickness = 2;
```

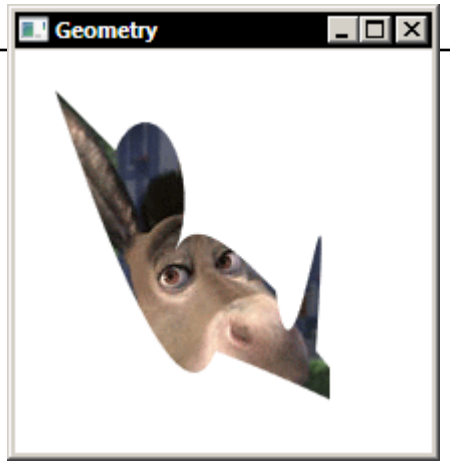
Geometria

- Klasa **Geometry** i jej klasy potomne pozwalają na opisanie geometrii dwuwymiarowych obiektów
- Obiekty **Geometry** są bardziej uniwersalne od obiektów **Shape**
  - **Shape** jest używany do renderowania grafiki dwuwymiarowej
  - **Geometry** może być użyte do zdefiniowania regionu dla grafiki dwuwymiarowej, regionu dla obcinania lub sprawdzania trafienia
- Obiekty klasy **Path** używają **Geometry** do określenia swojej zawartości
  - można to wykorzystać do narysowania obiektu **Geometry** (wykorzystując właściwości **Data**, **Fill** i **Stroke**)

Dostępne geometrie

- Proste geometrie:
  - **LineGeometry**, **RectangleGeometry**, **EllipseGeometry**
- **PathGeometry** zawiera kolekcję obiektów **PathFigure**, każdy taki obiekt może zawierać segmenty następujących typów:
  - **LineSegment**, **PolyLineSegment**, **ArcSegment**, **BezierSegment**, **PolyBezierSegment**, **QuadraticBezierSegment**, **PolyQuadraticBezierSegment**
- **StreamGeometry** definiuje złożony obiekt, który może zawierać krzywe, łuki i linie
  - przy określaniu zawartości **StreamGeometry** nie można wykorzystać wiązania danych ani animacji (można w **PathGeometry**)
- Obiekt **CombinedGeometry** i metoda **Combine** wykonują operację logiczną na dwóch geometriach
- Klasa **GeometryGroup** tworzy złączenie zawieranych przez siebie obiektów **Geometry**

Przykład PathGeometry



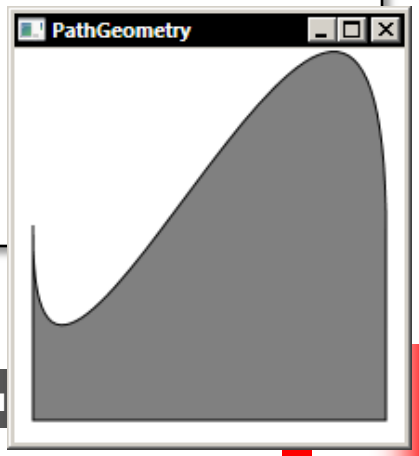

```
  <Image Source="/Images/Osiol_157x200.gif">
    <Image.Clip>
<!--<Path Stroke="Black" StrokeThickness="1" >
    <Path.Data>-->
      <PathGeometry>
        <PathGeometry.Figures>
          <PathFigure StartPoint="20,20">
            <PathFigure.Segments>
              <LineSegment Point="50,50"/>
              <ArcSegment Point="80,100" Size="10,20"
                          SweepDirection="Clockwise"/>
              <BezierSegment Point1="90,80" Point2="110,100" Point3="130,120"/>
              <PolyBezierSegment
                       Points="140,200,160,0,150,150,190,200,180,180,100,150"/>
              <QuadraticBezierSegment Point1="80,200" Point2="20,20"/>
            </PathFigure.Segments>
          </PathFigure>
        </PathGeometry.Figures>
      </PathGeometry>
<!--</Path.Data>
  </Path>-->
    </Image.Clip>
  </Image>
```

Składnia deklaracji ścieżki

```
<Path Stroke="Black" Fill="Gray"
      Data="M 10,100 C 10,300 200,-200 200,100 V 200 H 10 Z" />
```

- M – move
- L – line
- H – horizontal line
- V – vertical line
- C – cubic Bezier curve
- Q – quadratic Bezier curve
- S – smooth Bezier curve
- A – elliptical arc
- Z – close path

```
<Path Stroke="Black"
      Fill="Gray">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="10,100"
                    IsClosed="True">
          <PathFigure.Segments>
            <BezierSegment Point1="10,300"
                           Point2="200,-200"
                           Point3="200,100"/>
            <LineSegment Point="200,200"/>
            <LineSegment Point="10,200"/>
          </PathFigure.Segments>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```



Obrazki

- *WPF Imaging* zawiera kodeki dla formatów: BMP, JPEG, PNG, TIFF, Windows Media Photo, GIF i ICON
  - ICON jest jedynym formatem bez obsługi zapisu
- **BitmapSource** reprezentuje pojedynczy, niezmienny zbiór pikseli o ustalonym rozmiarze i rozdzielczości
  - może to być także klatka animowanego obrazka lub efekt transformacji innego obiektu **BitmapSource**
- **BitmapFrame** jest wykorzystywany do przechowywania faktycznych danych rastrowych
- Za pomocą właściwości **Metadata** obiektu **BitmapSource** możliwy jest dostęp do metadanych obrazka

Grafika trójwymiarowa

- Trójwymiarowa grafika w WPF jest umieszczana w elementach **Viewport3D**, które mogą należeć do drzewa wyświetlania
  - **Viewport3D** jest powierzchnią, na którą następuje projekcja sceny
  - dwu- i trójwymiarowe obiekty na siebie oddziaływać w obrębie **Viewport3D**
- Układy współrzędnych w dwu- i trójwymiarowej grafice są różne:

Figure 1
2D Coordinate System

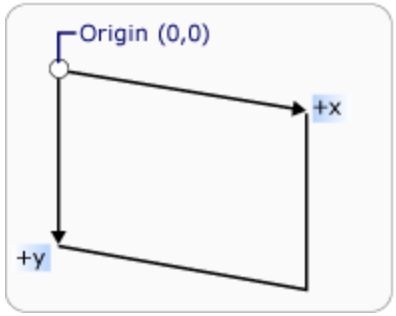
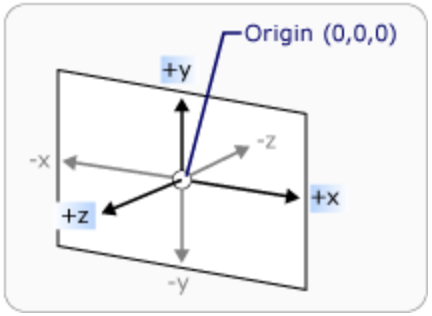


Figure 2
3D Coordinate System

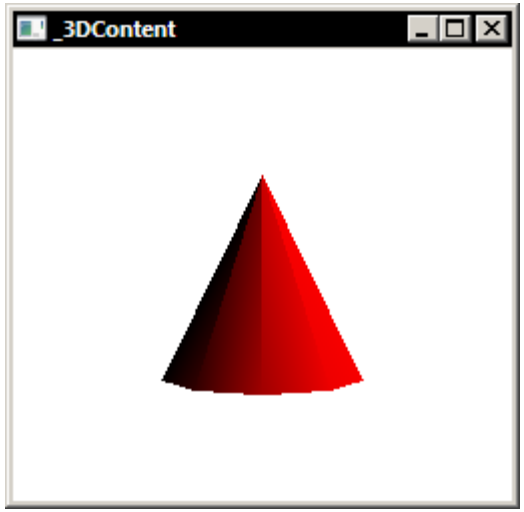


- Dostępne są 3 predefiniowane transformacje trójwymiarowe (dziedziczące z abstrakcyjnej klasy **Transform3D**): **TranslateTransform3D**, **ScaleTransform3D**, **RotateTransform3D**

Grafika trójwymiarowa c.d.

- **Camera** pozwala na zdefiniowanie punktu widzenia obserwatora
  - **ProjectionCamera** daje możliwość zastosowania różnych projekcji i ich właściwości
    - **PerspectiveCamera** uwzględnia standardową projekcję perspektywiczną
- **Model3D** jest klasą abstrakcyjną reprezentującą trójwymiarowy obiekt
  - **GeometryModel3D** tworzy trójwymiarowy model zawierający **MeshGeometry3D** i **Material**
- Do oświetlenia tworzonej sceny można wykorzystać następujące źródła światła: **AmbientLight**, **DirectionalLight**, **PointLight**, **SpotLight**

```
<!-- Viewport3D is the rendering surface. -->
<Viewport3D Name="myViewport" >
  <!-- Add a camera. -->
  <Viewport3D.Camera>
    <PerspectiveCamera FarPlaneDistance="20"
          LookDirection="0,0,1" FieldOfView="45"
          UpDirection="0,1,0" NearPlaneDistance="1"
          Position="0,0,-3" />
  </Viewport3D.Camera>
  <!-- Add models. -->
  <Viewport3D.Children>
    <ModelVisual3D>
      <ModelVisual3D.Content>
        <Model3DGroup >
          <Model3DGroup.Children>
            <!-- Lights, MeshGeometry3D and DiffuseMaterial -->
            <DirectionalLight Color="#FFFFFFFF" Direction="3,-4,5" />
            <!-- Define a red cone. -->
            <GeometryModel3D>
              <GeometryModel3D.Geometry>
                <MeshGeometry3D [...] />
              </GeometryModel3D.Geometry>
              <GeometryModel3D.Material>
                <DiffuseMaterial>
                  <DiffuseMaterial.Brush>
                    <SolidColorBrush Color="Red" Opacity="1.0"/>
                  </DiffuseMaterial.Brush>
                </DiffuseMaterial>
              </GeometryModel3D.Material>
[...]
```



Animacja

- WPF zawiera wydajny system odliczania czasu
  - jest dostępny zarówno z kodu źródłowego jak i z kodu XAML
  - może zostać wykorzystany do animacji kontrolek i innych obiektów
- W WPF obiekty są animowane poprzez "animację" wartości ich właściwości, pod warunkiem, że:
  - są to tzw. *dependency properties*
  - właściwości są obiektami klasy dziedziczącej z **DependencyObject** i implementującej interfejs **IAnimatable**
  - istnieje animacja kompatybilna z typem właściwości
- Animacje mogą być wykorzystywane niemal wszędzie, włączając definicje stylów i wzorców kontrolek
  - animacje nie są ograniczone do widocznych efektów

Typy animacji

- ***<Type>*Animation** (np. **DoubleAnimation** lub **ColorAnimation**)
  - "animuje" pomiędzy początkową (**From**) i końcową (**To**) wartością lub przez dodanie wartości kroku (**By**) do aktualnej wartości (począwszy od początkowej)
- ***<Type>*AnimationUsingKeyFrames**
  - można zdefiniować dowolną liczbę punktów animacji i sposób interpolacji pomiędzy nimi
- ***<Type>*AnimationUsingPath**
  - wykorzystuje ścieżkę geometryczną
- ***<Type>*AnimationBase**
  - abstrakcyjna klasa, którą można wykorzystać do stworzenia własnych animacji

Klasa Timeline

- Wszystkie typy animacji dziedziczą z klasy **Timeline**
- **Timeline** definiuje segment czasu i pozwala na określenie następujących właściwości:
  - **Duration** (domyślnie 1 sekunda)
  - **AutoReverse** – czy po zakończeniu odtworzyć animację w przeciwnym kierunku
  - **RepeatBehavior** – ile razy powtórzyć animację (domyślnie 1)
  - **FillBehavior** – co zrobić z wartością animowanej właściwości po zakończeniu animacji
- Za kontrolę czasu podczas animacji odpowiedzialny jest obiekt **Clock**
  - **Clock** udostępnia następujące informacje: **CurrentTime**, **CurrentProgress** i **CurrentState**

Przykład animacji - zmiana przezroczystości

1. Dodać element **DoubleAnimation** (pozwala na zmianę wartości w zadanym zakresie)
2. Dodać element **Storyboard** (we właściwości **TargetName** podać nazwę obiektu, a w **TargetProperty** nazwę jego animowanej właściwości)
3. Przyporządkować **Storyboard** do triggera

```
<Rectangle Name="MyRectangle" Width="100" Height="100" Fill="Blue">
  <Rectangle.Triggers>
    <EventTrigger RoutedEvent="Rectangle.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation Storyboard.TargetName="MyRectangle"
             Storyboard.TargetProperty="Opacity" From="1.0" To="0.0"
             Duration="0:0:5" AutoReverse="True" RepeatBehavior="Forever" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Rectangle.Triggers>
</Rectangle>
```

Animacja typu *Key-Frame*

- Udostępnia trzy rodzaje interpolacji wartości pomiędzy węzłami:
  - liniowa – stały współczynnik zmiany wartości
  - dyskretna – skoki do kolejnych zdefiniowanych wartości
  - oparta na krzywej – współczynnik zmiany wartości jest oparty na krzywej Beziera
    - dwa punkty krzywej są stałe ((0.0, 0.0), (1.0, 1.0)), pozostałe dwa są definiowane za pomocą właściwości **KeySpline**
- Można używać różnych interpolacji w jednej animacji
- Właściwość **KeyTime** określa czas końca segmentu animacji
  - można określać wartość tej właściwości procentowo
  - specjalna wartość "Paced" może być użyta dla określenia, że współczynnik interpolacji ma być stały

Przykłady animacji typu *Key-Frame*

```
<Storyboard>
  <DoubleAnimationUsingKeyFrames
       Storyboard.TargetName="MyAnimatedTranslateTransform"
       Storyboard.TargetProperty="X"
       Duration="0:0:10">
    <LinearDoubleKeyFrame Value="0" KeyTime="0:0:0" />
    <LinearDoubleKeyFrame Value="350" KeyTime="0:0:2" />
    <LinearDoubleKeyFrame Value="50" KeyTime="0:0:7" />
    <LinearDoubleKeyFrame Value="200" KeyTime="0:0:8" />
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

```
<Storyboard>
  <DoubleAnimationUsingKeyFrames
       Storyboard.TargetName="ComboAnimatedTranslateTransform"
       Storyboard.TargetProperty="X"
       Duration="0:0:15"
       RepeatBehavior="Forever">
    <DiscreteDoubleKeyFrame Value="500" KeyTime="0:0:7" />
    <LinearDoubleKeyFrame Value="200" KeyTime="0:0:10" />
    <SplineDoubleKeyFrame Value="350" KeyTime="0:0:15"
       KeySpline="0.25,0.5 0.75,1" />
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

Animacja oparta na ścieżce

- Ten typ animacji wykorzystuje obiekt **PathGeometry**
  - podczas trwania animacji współrzędne x i y oraz kąt są odczytywane ze ścieżki i wykorzystywane podczas animacji
- Zdefiniowane są trzy typy animacji opartych na ścieżkach:
  - **MatrixAnimationUsingPath** generuje obiekty **Matrix**
    - sposobem na przesuwanie obiektu po wskazanej drodze jest wykorzystanie **MatrixTransform** i **MatrixAnimationUsingPath**
    - właściwość **DoesRotateWithTangent** może być wykorzystana do określenia transformacji obrotu przesuwanego obiektu
  - **PointAnimationUsingPath** generuje obiekty klasy **Point** ze współrzędnych **PathGeometry**
  - **DoubleAnimationUsingPath** generuje wartości **Double** z **PathGeometry**

Przykład animacji opartej na ścieżce

```
<Button MinWidth="100" Content="A Button">
  <Button.RenderTransform>
    <MatrixTransform x:Name="ButtonMatrixTransform"/>
  </Button.RenderTransform>
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <MatrixAnimationUsingPath
                Storyboard.TargetName="ButtonMatrixTransform"
                Storyboard.TargetProperty="Matrix"
                DoesRotateWithTangent="True"
                Duration="0:0:5"
                RepeatBehavior="Forever" >
            <MatrixAnimationUsingPath.PathGeometry>
              <PathGeometry
                Figures="M 10,100 C 35,0 135,0 160,100 180,190 285,200 310,100"
                PresentationOptions:Freeze="True" />
            </MatrixAnimationUsingPath.PathGeometry>
          </MatrixAnimationUsingPath>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

Możliwości animacji wartości właściwości

- Istnieją 4 możliwości "animacji" wartości właściwości obiektu:
  - *Storyboard animation*
    - pozwala na zdefiniowanie i zastosowanie animacji w kodzie XAML, interaktywną kontrolę animacji po jej starcie, tworzenie złożonego drzewa animacji lub animację właściwości **Style**, **ControlTemplate** i **DataTemplate**
  - *Local animation*
    - wygodny sposób na "animację" wartości *dependency property* dowolnego obiektu **Animatable**
  - *Clock animation*
    - pozwala na zdefiniowanie złożonych zależności czasowych i na interaktywną kontrolę animacje po jej uruchomieniu
  - *Per-frame animation*
    - użyteczne, gdy trzeba niemal w całości podmienić system animacji wbudowany w WPF (np. animacje procesów fizycznych)

Storyboard

- **Storyboard** jest kontenerem dla obiektów *timeline*
- Może być wykorzystany do animacji właściwości *(dependency property)* obiektu dowolnej klasy dziedziczącej z **Animatable**
- **Storyboard** może rozpocząć swoje działanie przez wywołanie z *triggera* lub z metody
  - w kodzie XAML można wykorzystać element **BeginStoryboard** z jednym z rodzajów *triggerów*: **EventTrigger**, **Trigger** lub **DataTrigger**
  - w kodzie wystarczy wywołać metodę **Begin**
- **Storyboard** może być kontrolowany z poziomu kodu XAML
  - dostępne operacje: *pause, resume, seek, skipToFill, stop, remove*

Timing Events

- Klasy **Timeline** i **Clock** udostępniają 5 zdarzeń związanych z upływem czasu:
  - **Completed, CurrentGlobalSpeedInvalidated, CurrentStateInvalidated, CurrentTimeInvalidated, RemoveRequested**

```
myStoryboard.Stop(myRectangle);

// This statement might execute before the storyboard has stopped.
myRectangle.Fill = Brushes.Blue;
```

```
// Register for the CurrentStateInvalidated timing event.
myStoryboard.CurrentStateInvalidated +=
    new EventHandler(myStoryboard_CurrentStateInvalidated);
```

```
// Change the rectangle's color after the storyboard stops.
void myStoryboard_CurrentStateInvalidated(object sender, EventArgs e) {
    Clock myStoryboardClock = (Clock)sender;
    if (myStoryboardClock.CurrentState == ClockState.Stopped) {
        myRectangle.Fill = Brushes.Blue;
    }
}
```

Klasa Freezable

- Klasa **Freezable** określa dla swoich obiektów jeden z dwóch stanów: zamrożony i niezamrożony (*unfrozen* i *frozen*)
  - w stanie zamrożonym obiekt nie może być modyfikowany
- Zamrożenie obiektu zwiększa wydajność – brak jest powiadomień związanych z modyfikacją obiektu
  - zamrożony obiekt może być jednocześnie używany przez różne wątki (to jest niemożliwe dla obiektu niezamrożonego)
- Metoda **Freeze** zamraża obiekt
  - właściwość **CanFreeze** określa, czy obiekt może zostać zamrożony
  - **InvalidOperationException** jest rzucany w przypadku nieudanej próby zamrożenia
  - właściwość **IsFrozen** sprawdza, czy obiekt jest zamrożony

Multimedia

- Istnieją dwie klasy, które mogą być wykorzystane do prezentacji treści multimedialnych (obie wymagają Microsoft Windows Media Player 10 OCX):
  - **MediaElement** jest obiektem **UIElement**, który można dodawać jako zawartość do kontrolek w XAML i w kodzie
  - **MediaPlayer** jest przeznaczony dla obiektów **Drawing**
    - media wczytane za pomocą **MediaPlayer** mogą być prezentaowane tylko za pomocą **VideoDrawing** lub przez bezpośrednią interakcję z **DrawingContext**
    - nie może być wykorzystany w XAML
- Obsługiwane są dwa tryby
  - Independent mode – media są odpowiedzialne za odtwarzanie
  - Clock mode – **MediaTimeline** odpowiada za odtwarzanie

Klasa MediaElement

- Do odtworzenia treści multimedialnej wystarczy dodać kontrolkę **MediaElement** i podać adres **Uri** treści (np. filmu)
- Właściwości **LoadedBehavior** albo **UnloadedBehavior** odpowiadają za zachowanie obiektu **MediaElement** odpowiednio gdy **IsLoaded** jest *true* albo *false*
- **MediaElement** jest widoczny, gdy jest dostępna treść, którą ma odtworzyć
  - do chwili załadowania treści właściwości **ActualWidth** i **ActualHeight** mają wartości 0
  - dla treści będącej filmem **ActualWidth** i **ActualHeight** zostaną ustawione stosownie do rozmiaru przed zdarzeniem **MediaOpened**
  - ustawienie właściwości **Width** i **Height** spowoduje przeskalowanie filmu; dla zachowania stosunku wysokości do szerokości należy ustawiać tylko jedną z tych właściwości

```
<StackPanel Name="MainPanel" Background="Black">
  <MediaElement Name="myMediaElement" MediaOpened="Element_MediaOpened" />
  <StackPanel HorizontalAlignment="Center" Width="260" Orientation="Horizontal">
    <Button Name="PlayButton" Margin="30,10,10,10">Play</Button>
    <Button Name="PauseButton" Margin="10">Pause</Button>
    <Button Name="ResumeButton" Margin="10">Resume</Button>
    <Button Name="StopButton" Margin="10">Stop</Button>
  </StackPanel>
  <Slider Name="timelineSlider" Margin="5" HorizontalAlignment="Center" />
  <StackPanel.Triggers>
    <EventTrigger RoutedEvent="Button.Click" SourceName="PlayButton">
      <EventTrigger.Actions>
        <BeginStoryboard Name= "myBegin">
          <Storyboard SlipBehavior="Slip">
            <MediaTimeline Source="Movies\TestMovie.wmv"
                           Storyboard.TargetName="myMediaElement"
                           CurrentTimeInvalidated="MediaTimeChanged" />
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.Click" SourceName="PauseButton">
      <EventTrigger.Actions>
        <PauseStoryboard BeginStoryboardName="myBegin" />
      </EventTrigger.Actions>
    </EventTrigger>
    [...]
  </StackPanel.Triggers>
</StackPanel>
```

MediaPlayer

```
MediaTimeline timeline = new MediaTimeline(
    new Uri(@"Movies\TestMovie.wmv", UriKind.Relative));
timeline.RepeatBehavior = RepeatBehavior.Forever;

// Create the clock, which is shared with the MediaPlayer
MediaClock clock = timeline.CreateClock();
MediaPlayer player = new MediaPlayer();
player.Clock = clock;
// Create the VideoDrawing
VideoDrawing videoDrawing = new VideoDrawing();
videoDrawing.Rect = new Rect(0, 0, 1, 1);
videoDrawing.Player = player;

// Assign the DrawingBrush
DrawingBrush brush = new DrawingBrush(videoDrawing);
brush.Viewport = new Rect(0, 0, 0.5, 0.25);
brush.TileMode = TileMode.Tile;

// Start the timeline
clock.Controller.Begin();

MainCanvas.Background = brush;
```

```
<Canvas Name="MainCanvas" Background="Pink" />
```