HOCHSCHULE HAMM-LIPPSTADT

Praktische Informatik

Vorlesung 07

2D Grafik

Zuletzt haben wir gelernt…

- Wie man Fenster erstellt und auf verschiedene Arten anzeigt.
- Das bereits fertige Dialogklassen existieren.
- Wie man Steuerelemente mit Hilfe von Layoutcontainern positioniert.
- Wie das StackPanel und das WrapPanel funktionieren.
- Wie man mit dem DockPanel und dem Grid umgeht.
- Wozu das Canvas benutzt werden kann.
- Wie man Layoutcontainern ineinander verschachtelt, um komplexe Layouts zu erzeugen.

Inhalt heute

- Shapes
- Bilder
- Pinsel
- Shapes bewegen
- Pong
- Transformationen

Shapes

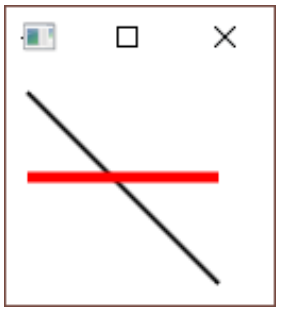
- In WPF ist es sehr einfach, grafische Elemente zu verwenden.
  - Grafische Elemente können überall in Layout Containern eingesetzt werden.
  - Meist wird ein Canvas benutzt, um die Elemente absolut zu positionieren.

- Die Klasse **Shape** ist für viele grafische Elemente die Basis.
  - Sie definiert folgende gemeinsamen Eigenschaften:
  - **Stroke**: Eine Angabe, wie der Rand gezeichnet werden soll.
  - **StrokeThikness**: Die Dicke des Randes.
  - **Fill**: Die Füllung des Elements.

- Es existieren die folgenden Subklassen von Shape:

Klasse	Realisiert
Line	Eine einfache Linie.
Ellipse, Rectangle	Ellipsen und Rechtecke
Polygone, Polyline	Ein geschlossener/offener Polygonzug.
Path	Eine komplexe Figur.

Line

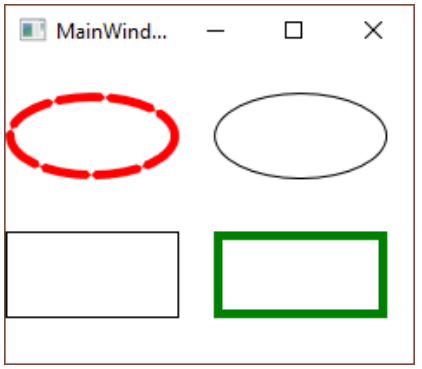


- Eine Linie wird über die Klasse Line abgebildet.
  - Dabei werden Start- und Endpunkt über die Positionen X1/Y1 und X2/Y2 im übergeordneten Container festgelegt.

```
<Canvas>
    <Line X1="10" Y1="10" X2="100" Y2="100" Stroke="Black" StrokeThickness="2" />
    <Line X1="10" Y1="50" X2="100" Y2="50" Stroke="Red" StrokeThickness="5"/>
</Canvas>
```

- Neben den Eigenschaften `Stroke` und `StrokeThikness` existieren viele weitere Eigenschaften, um den Stil der Linie zu beeinflussen, z.B.:
  - `StrokeDashArray`: Eine Liste von Werten, welches das Strichmuster der Linie festlegt.
  - `StrokeDashCap`: Definiert das Ende der Linien beim Strichmuster.

Ellipse und Rechteck



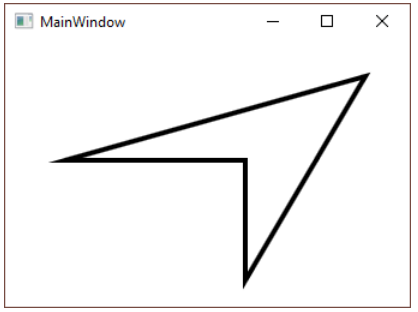
- Auch die Ellipse bzw. ein Kreis ist mit Hilfe der Klasse `Ellipse` sehr einfach erstellt.

```
<Ellipse Canvas.Top="20" Canvas.Left="0" Width="100" Height="50"
   Stroke="Red" StrokeThickness="5" StrokeDashArray="5 1" StrokeDashCap="Triangle" />
<Ellipse Canvas.Top="20" Canvas.Left="120" Width="100" Height="50"
   Stroke="Black" StrokeThickness="1" />
```

- Entsprechendes gilt für das Rechteck, bzw. das Quadrat:

```
<Rectangle Canvas.Top="100" Canvas.Left="0" Width="100" Height="50"
   Stroke="Black" StrokeThickness="1" />
<Rectangle Canvas.Top="100" Canvas.Left="120" Width="100" Height="50"
   Stroke="Green" StrokeThickness="5" />
```

Polygon

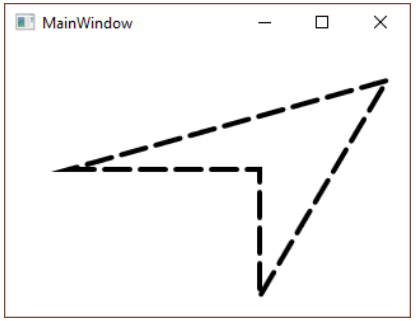


- Ein Polygon ist ebenfalls ein Shape.
  - Es stellt einen geschlossenen Zug von Linien dar.
  - Diese werden durch einzelne Punkte vorgegeben.

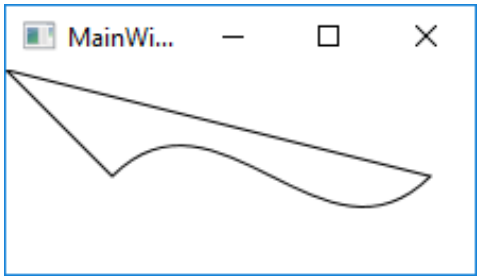
```
<Polygon Points="50,100 200,100 200,200 300,30" Stroke="Black" StrokeThickness="4" />
```

- Auch hier kann der Stil der Linie wieder entsprechend konfiguriert werden.

```
<Polygon Points="50,100 200,100 200,200 300,30"
    Stroke="Black"
    StrokeThickness="4"
    StrokeDashArray="5 2"
    StrokeDashCap="Round"
/>
```



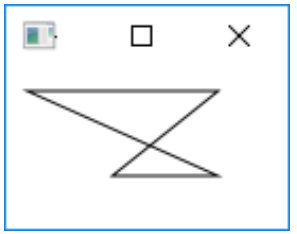
Path



- Mit Objekten vom Typ Path können komplexe Figuren bestehend aus Linien und Kurven gezeichnet werden.
  - Die PathGeometry eines Paths-Objektes kann aus vielen Figuren bestehen.
  - Dort werden dann Linien-Segmente oder Bezier-Kurven hinterlegt.

```
<Path Stroke="Black" >
    <Path.Data>
        <PathGeometry>
            <PathFigure IsClosed="True">
                <LineSegment Point="50, 50" />
                <BezierSegment Point1="100,0" Point2="150,100" Point3="200,50" />
            </PathFigure>
        </PathGeometry>
    </Path.Data>
</Path>
```

Path mini Sprache



- Die Klasse Path erlaubt es auch mit Hilfe einer Art mini Programmiersprache komplexe Figuren zu beschreiben.

```
<Path Stroke="Black" Data="M10,10 L 100,10 50,50 100,50 Z" />
```

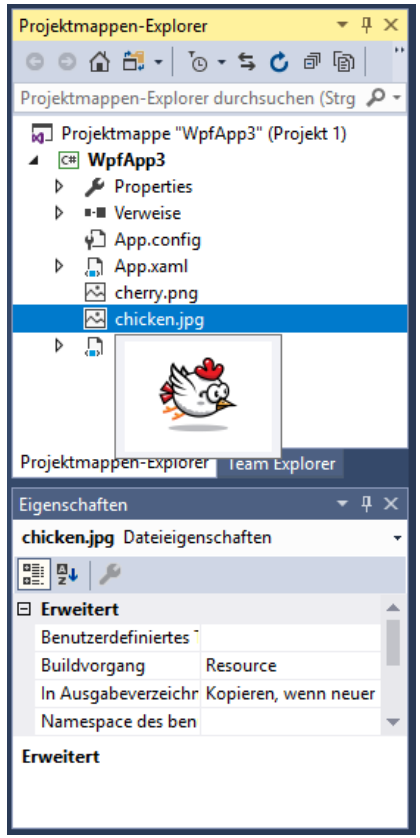
- In dieser Mini-Sprache können z.B. die folgenden Anweisungen benutzt werden:

Anweisung	Bedeutung
M10,10	Bewege den Stift zur Position 10,10 ohne zu malen.
L 100,10	Zeichne eine Linie zum Punkt 100,10.
50,50	Zeichne weiter zu 50,50.
Z	Schließe die Form durch Zeichnen einer Linie zum Ausgangspunkt.

Es existieren noch viele weitere Möglichkeiten. Siehe hier.

Dateien dem Projekt hinzufügen

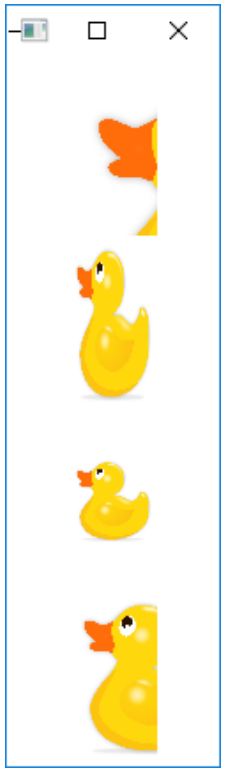
- Mit Hilfe der Klasse `Image` können auch Bilddateien in einem Layout Container benutzt werden.
  - Zunächst müssen aber Bilddateien dem Projekt hinzugefügt werden.
  - Dies kann über den Menüpunkt "Hinzufügen" geschehen.
- Danach sollten im Visual Studio die Eigenschaften der Bilddateien angepasst werden.
  - Es muss mindestens dafür gesorgt werden, dass die Datei beim Erstellen des Projektes in das Ausgabeverzeichnis kopiert wird.
- Die Datei kann aber auch als **Ressource** definiert werden.
  - Dann wird die Datei in die Binärdatei des Projektes hineinkompiliert.
  - Die Datei muss dann nicht mehr separat auf andere Rechner weitergegeben werden.



Image

- Ein Bild aus einer Bilddatei lässt sich mit der Klasse Image sehr einfach in einem Layout Container darstellen.
  - Die Bilddatei muss im lokalen Verzeichnis zu finden, oder als Ressource eingebunden sein.

```
<Image Source="duck.png" />
```



- Dem Image-Objekt kann eine andere Größe vorgegeben, werden, als die Bilddatei selber besitzt.
  - Über die Eigenschaft Stretch kann dann vorgegeben werden, wie sich das Bild an die Größe angepasst.

```
<Image Source="duck.png" Width="50" Height="100" Stretch="None"/>
<Image Source="duck.png" Width="50" Height="100" Stretch="Fill"/>
<Image Source="duck.png" Width="50" Height="100" Stretch="Uniform"/>
<Image Source="duck.png" Width="50" Height="100" Stretch="UniformToFill"/>
```

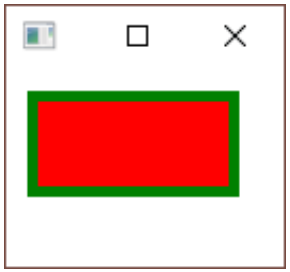
Pinsel

- Alles, was in WPF sichtbar ist, wird mit einem **Pinsel (engl. brush)** gezeichnet.
  - Auch die Shapes benutzen Pinsel, um die Linien zu zeichnen.
- Dazu haben wir gerade die Eigenschaft `Stroke` auf eine einzelne Farbe gesetzt.
  - `Stroke="Black"`
  - Diese Farbe wird automatisch in ein Objekt vom Typ `Brush` umgewandelt.
- Um die Pinselfarbe eines Rechtecks auf Schwarz zu setzen, können wir dies auch wie folgt hinschreiben:

```
<Rectangle StrokeThickness="1">
  <Rectangle.Stroke>
    <SolidColorBrush Color="Black" />
  </Rectangle.Stroke>
</Rectangle>
```

Der SolidColorBrush ist der einfachste Typ von Pinseln.

Füllen mit Pinsel



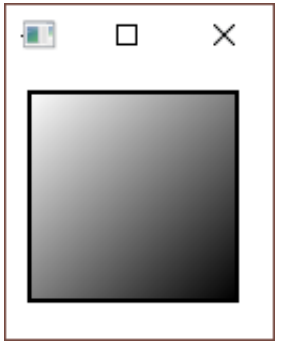
- Ein Pinsel vom Typ `Brush` kann nicht nur für die Linie des `Shapes` benutzt werden.
  - Man kann mit einem Pinsel auch ein grafisches Element füllen.
  - Dazu muss die Eigenschaft Fill mit einem Objekt vom Typ Brush belegt werden.
  - Am einfachsten ist dies mit dem `SolidColorBrush`, der lediglich eine einzige Farbe benutzt.

```
<Rectangle Canvas.Top="10" Canvas.Left="10" Width="100" Height="50" StrokeThickness="5">
    <Rectangle.Stroke>
        <SolidColorBrush Color="Green" />
    </Rectangle.Stroke>
    <Rectangle.Fill>
        <SolidColorBrush Color="Red" />
    </Rectangle.Fill>
</Rectangle>
```

Linearer Farbverlauf

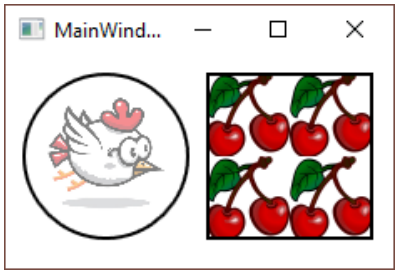
- Neben dem `SolidColorBrush` existieren einige weitere, komplexere Pinsel.
  - Ein Beispiel ist der `LinearGradientBrush`, der einen Farbverlauf auf einer Farbverlaufsachse definieren kann.

```
<Rectangle Canvas.Top="10" Canvas.Left="10" Width="100" Height="100"
StrokeThickness="2" Stroke="Black">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Color="White" Offset="0" />
            <GradientStop Color="Black" Offset="1" />
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```



- Die Farbverlaufsachse reicht im Normalfall von links oben nach rechts unten.
  - Mit Hilfe der Objekte vom Typ `GradientStop` kann dann angegeben werden, ab welcher Position auf der Achse (0 bis 1) entsprechende Farben ineinander übergehen sollen.

ImageBrush



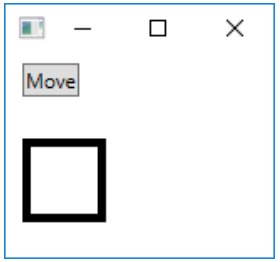
- Mit einem `ImageBrush` können Bilder auch als Pinsel benutzt werden.
  - Normalerweise wird das Bild so verwendet, dass es den gegebenen Platz in beiden Richtungen ausfüllt.
  - Die Art der Streckung/Stauchung kann über die Eigenschaft `Stretch` eingestellt werden.
  - Der Grad der Durchsichtigkeit wird über die `Opacity` definiert.

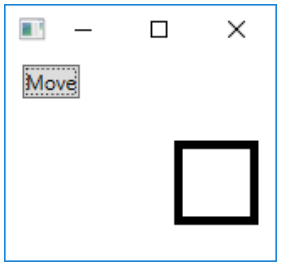
```
<Ellipse Width="100" Height="100" Stroke="Black" StrokeThickness="2">
    <Ellipse.Fill>
        <ImageBrush ImageSource="chicken.jpg" Stretch="UniformToFill" Opacity="0.5"/>
    </Ellipse.Fill>
</ Ellipse>
```

- Bilder können auch wie Kacheln in einem Shape angeordnet werden.
  - Dazu müssen einige weitere Eigenschaften gesetzt werden.

```
<Rectangle Width="100" Height="100" Stroke="Black" StrokeThickness="2">
    <Rectangle.Fill>
        <ImageBrush ImageSource="cherry.png" Stretch="None" TileMode="Tile"
         Viewport="0,0,50,50" ViewportUnits="Absolute"
         AlignmentX="Left" AlignmentY="Top"/>
    </Rectangle.Fill>
</Rectangle>
```

Shapes bewegen



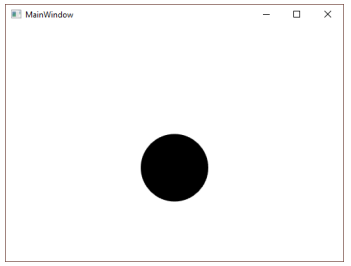


- Auf alle Shapes, die wir im XAML definiert haben, kann man natürlich auch wieder im Code Behind zugreifen.
  - Entsprechend muss die Eigenschaft **x:Name** gesetzt werden.
- Anschließend kann z.B. mit Hilfe der Klasse Canvas die Position im Layout Container verändert werden.
  - Das Element erscheint dann an der entsprechenden Stelle.

```
<Canvas>
    <Rectangle x:Name="rect" Canvas.Top="50" Canvas.Left="10"
       Width="50" Height="50" Stroke="Black" StrokeThickness="5" />
    <Button Canvas.Left="10" Canvas.Top="5" Click="Button_Click">Move</Button>
</Canvas>
```

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Canvas.SetLeft(rect, 100);
}
```

Hüpfender Ball



- Wir wollen ein wenig mit den Möglichkeiten experimentieren.
  - Dazu lassen wir einen Ball hüpfen.
  - Er soll sich wie ein echter Ball der Schwerkraft entsprechend bewegen.
  - Die Bewegung soll starten, wenn der Nutzer die Leertaste drückt.

- Der XAML Code dazu ist recht überschaubar.
  - Es muss lediglich ein Ball gezeichnet werden.
  - Zudem wird ein Event Handler für den Tastendruck definiert.

```
<Window x:Class="WpfApp3.MainWindow"
   Title="MainWindow" SizeToContent="WidthAndHeight" KeyDown="Window_KeyDown">
   <Canvas Height="350" Width="500">
     <Ellipse x:Name="ball" Canvas.Top="250" Canvas.Left="200"
        Width="100" Height="100" Stroke="Black" StrokeThickness="2" Fill="Black"/>
    </Canvas>
</Window>
```

DispatcherTimer

- Im Code Behind nutzen wir die Klasse DispatcherTimer.
  - Ein Objekt der Klasse generiert in festen Zeitintervallen einen Event.

```
private void Window_KeyDown(object sender, KeyEventArgs e)
{
   if (e.Key != Key.Space)
      return;

   if (timer != null)
      timer.Stop();

   t = 0;
   v = 75;

   timer = new DispatcherTimer();
   timer.Tick += Timer_Tick;
   timer.Interval = new TimeSpan(0, 0, 0, 0, 10);
   timer.Start();
}
```

Zeit t und Geschwindigkeit v des Balls als Objektvariablen.

Event Handler

- Durch den DispatchTimer wird die Methode Timer_Tick alle 10 Millisekunden aufgerufen.
  - Hier können wir die eigentliche Bewegung des Balls ablaufen lassen.

```
private void Timer_Tick(object sender, EventArgs e)
{
    t = t + 0.35;
    var h = v * t - 9.81 / 2 * t * t;
    Canvas.SetTop(ball, 250 - h);

    if (h < 0)
    {
        v = v * 0.8;
        t = 0;
    }

    if (v < 1)
    {
        timer.Stop();
    }
}
```

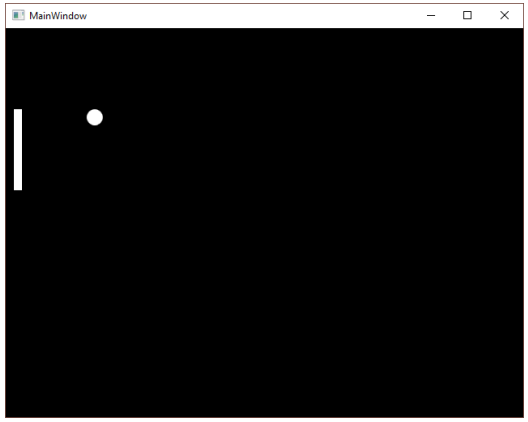
Die Höhe des Balls bestimmen und die Position im Canvas entsprechend setzen.

Pong

- Ähnlich zum hüpfenden Ball können wir auch ein kleine Computerspiel realisieren: PONG.

- Ein Ball prallt von den drei Wänden oben, unten und rechts ab.
  - Links ist ein steuerbares Paddel, mit welchem der Ball zurückgespielt werden muss.
  - Verfehlt der Ball das Paddel, ist das Spiel verloren.

- Alles, was wir zur Realisierung in WPF benötigen, haben wir bereits kennen gelernt.
  - Ellipse, Rectangle, DispatchTimer, ...



XAML von Pong

- Der XAML-Teil von Pong ist recht einfach.
  - Es wird lediglich der Ball und das Paddel erzeugt.
  - Beiden Elementen wird auch ein Name gegeben, so dass im Code Behind darauf zugegriffen werden kann.

```
<Canvas Width="640" Height="480" Background="Black">
   <Rectangle x:Name="paddle" Canvas.Left="10" Canvas.Top="100"
      Width="10" Height="100" Fill="White" />
   <Ellipse x:Name="ball" Canvas.Left="100" Canvas.Top="100"
      Width="20" Height="20" Fill="White" />
</Canvas>
```

Initialisierung von Pong

- In der Klasse `MainWindow` initialisieren wir das Spiel.
  - Wir legen die anfängliche Bewegungsrichtung des Balls mit den Objektvariablen `delta_x` und `delta_y` fest.
  - Zudem erstellen wir ein Objekt vom Typ `DispatchTimer`, der eine Methode `Timer_Tick` alle 10 ms aufruft.

```
private int delta_x = 5;
private int delta_y = 5;

public MainWindow()
{
   InitializeComponent();
   var timer = new DispatcherTimer();
   timer.Tick += Timer_Tick;
   timer.Interval = new TimeSpan(0, 0, 0, 0, 10);
   timer.Start();
}
```

Paddle bewegen

- Um das Paddel zu bewegen, müssen wir auf die Tastaturereignisse reagieren.
  - Dazu haben wir uns im XAML an das Ereignis KeyDown angehängt.
  - Entsprechend können wir die Position des Paddels in den erlaubten Bereichen dynamisch verändern.

```
private void Window_KeyDown(object sender, KeyEventArgs e)
{
    double paddle_y = Canvas.GetTop(paddle);

    if (e.Key == Key.Down && paddle_y + 120 <= 480)
        Canvas.SetTop(paddle, paddle_y + 20);

    if (e.Key == Key.Up == paddle_y - 20 >= 0)
        Canvas.SetTop(paddle, paddle_y - 20);
}
```

Ball bewegen

- Den kompliziertesten Programmcode müssen wir für die Bewegung des Balls schreiben.
- Wenn der Ball eine der drei Wände erreicht hat, muss die Bewegungsrichtung geändert werden.
- Ebenso, wenn er auf das Paddel prallt.
- Erreicht der Ball die linke Wand, ist das Spiel allerdings verloren.

```
private void Timer_Tick(object sender, EventArgs e)
{
    double ball_x = Canvas.GetLeft(ball);
    double ball_y = Canvas.GetTop(ball);
    double paddle_x = Canvas.GetLeft(paddle);
    double paddle_y = Canvas.GetTop(paddle);

    if (ball_x >= 620)
        delta_x *= -1;

    if (ball_y >= 460 || ball_y < 0)
        delta_y *= -1;

    if (ball_x == 20 && ball_y >= paddle_y
        && ball_y <= paddle_y + 100)
        delta_x *= -1;

    if (ball_x <= 0)
    {
        MessageBox.Show("Leider verloren!");
        Close();
    }

    Canvas.SetLeft(ball, ball_x + delta_x);
    Canvas.SetTop(ball, ball_y + delta_y);
}
```

Transformationen

- Viele Elemente der WPF können mit Hilfe von Transformationen in Ihrem Aussehen verändert werden.
  - Dazu gehören die Shapes, aber auch alle Klassen, die von `FrameworkElement` ableiten, wie z.B. Buttons.

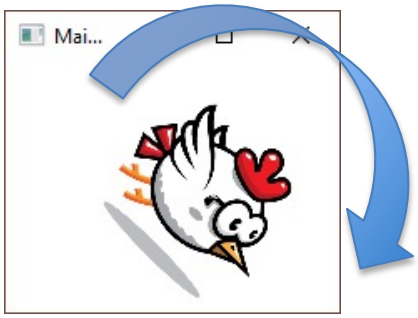
- Einem Element können beliebig viele Transformationen hinzugefügt werden.
  - Darunter z.B. Rotationen, Neigungen, Spiegelungen und Verschiebungen.

Beispiel

- Wie zuvor gesehen, kann ein Bild mit Hilfe der Image-Klasse dargestellt werden.
  - Der Image Klasse können ebenfalls Transformationen hinzugefügt werden.

- Eine mögliche Transformation ist die **Rotation**.
  - Diese wird durch den Winkel und den Mittelpunkt bestimmt.

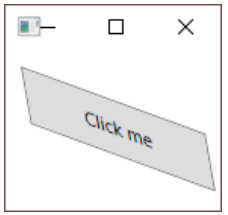
```
<Image Source="chicken.jpg" Width="200">
    <Image.RenderTransform>
        <TransformGroup>
            <RotateTransform Angle="45" CenterX="100" CenterY="100"/>
        </TransformGroup>
    </Image.RenderTransform>
</Image>
```



Weitere Transformationen

- Transformationen können auch auf Elemente, wie z.B. einen `Button` angewandt werden.
  - Mit Hilfe der `SkewTransformation` können wir Neigungen erzeugen.

```
<Button Margin="10" Padding="10">Click me
    <Button.RenderTransform>
        <SkewTransform AngleX="10" AngleY="20" />
    </Button.RenderTransform>
</Button>
```



- Eine ScaleTransformation vergrößert/verkleinert Elemente in X/Y-Richtung.

```
<Button Margin="10" Padding="10">Click me
    <Button.RenderTransform>
        <ScaleTransform ScaleY="3" />
    </Button.RenderTransform>
</Button>
```

Animationen und Trigger

- Die WPF bietet weit mehr Möglichkeiten, als wir in dieser Vorlesung behandeln könnten.
  - Evtl. wollen Sie sich noch zwei weitere Möglichkeiten selber ansehen?

- **Animationen**
  - Mit Hilfe von Animationen können die Werte alle Abhängigkeitseigenschaften animiert werden.
  - So kann z.B. die Farbe, die Position oder die Transformation eines Elements kontinuierlich verändert werden.
  - Solche Animationen können in XAML festgelegt werden, ohne Hilfsmittel wie den DispatchTimer benutzen zu müssen.

- **Trigger**
  - Um Animationen zu starten oder zu stoppen können Trigger definiert werden.
  - So kann in XAML z.B. eine Animation gestartet werden, wenn ein Button geklickt wird, oder eine Eigenschaft einen beliebigen Wert annimmt.

Wir haben heute gelernt…

- Welche Shapes in WPF zum Zeichnen benutzt werden können.
- Wie man Bilder darstellt.
- Wie mit unterschiedlichen Pinsel-Arten die Darstellung von Linien und Füllungen gestaltet werden kann.
- Wie man Shapes dynamisch im Code Behind bewegen kann.
- Wie man damit z.B. das Computerspiel Pong realisieren kann.
- Wie man auf Elemente in WPF grafische Transformationen anwendet.