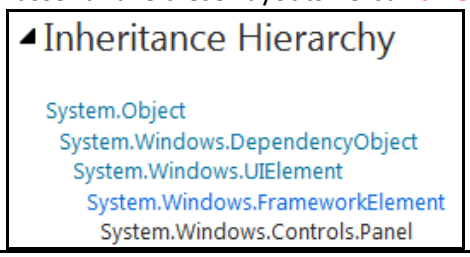
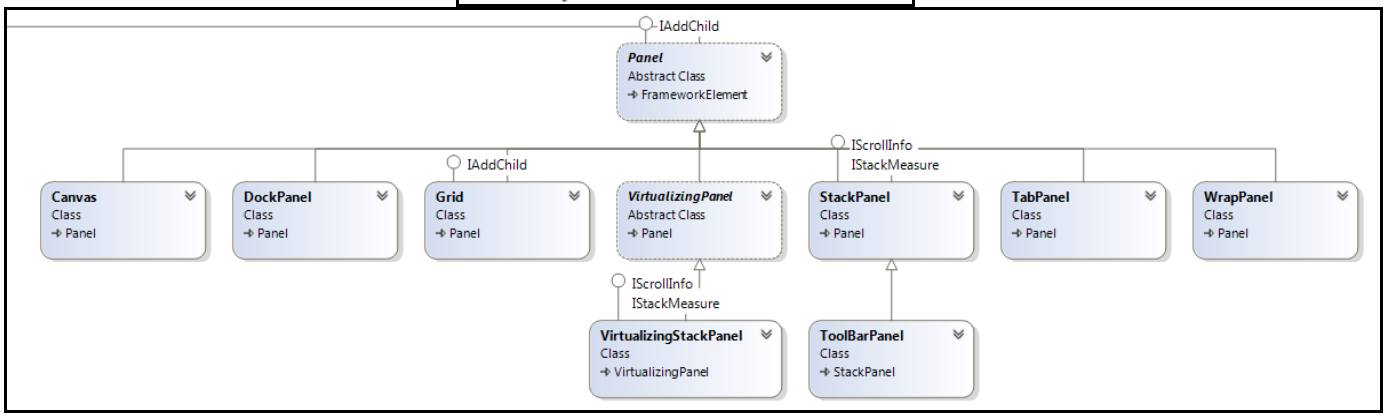
WPF
Layouts

1 Layouts (Panels)

Üblicherweise werden die Controls in WPF nicht absolut im Window positioniert, sondern – ähnlich wie bei Swing – über Layouts angeordnet. Die Basisklasse für alle diese Layouts heißt **`Panel`**:





Ein Panel ist also hauptsächlich ein Container für Controls, die ja von der Klasse **`FrameworkElement`** und daher auch von **`UIElement`** abgeleitet sind (wie auch Panel selbst). Die zugehörige Property heißt **`Children`** (die sinnvollerweise auch die **`ContentProperty`** ist) und kann eben UIElement-Objekte aufnehmen:

```
[ContentProperty("Children")]
public abstract class Panel : FrameworkElement, IAddChild
```

```
/// <summary>
/// Returns a UIElementCollection of children for user to add/remove children manually
/// Returns read-only collection if Panel is data-bound (no manual control of children is possible,
/// the associated ItemsControl completely overrides children)
/// Note: the derived Panel classes should never use this collection for
/// internal purposes like in their MeasureOverride or ArrangeOverride.
/// They should use InternalChildren instead, because InternalChildren
/// is always present and either is a mirror of public Children collection (in case of Direct Panel)
/// or is generated from data binding.
/// </summary>
[DesignerSerializationVisibility(DesignerSerializationVisibility.Content)]
public UIElementCollection Children...
```

In weiterer Folge ergibt sich daraus, dass man Panels auch schachteln kann (**`Children`** ist vom Typ **`UIElementCollection`** und sowohl Controls als auch Layouts sind UIElements).

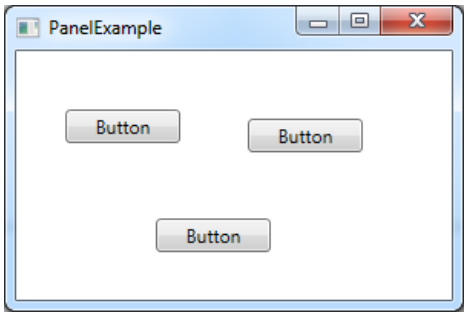
Neben der Property **`Children`** hat Panel nur noch 3 weitere Properties

- **`Background`**: setzt den Hintergrund in Form eines **`Brush`** (Default ist **`null`**)
- **`ZIndex`**: wird den Kindelementen zugewiesen um festzulegen, welche davon im Vordergrund liegen. Je kleiner der Werte, desto weiter im Hintergrund.
- **`IsItemsHost`**: wichtig für Styles, Triggers u. Templates

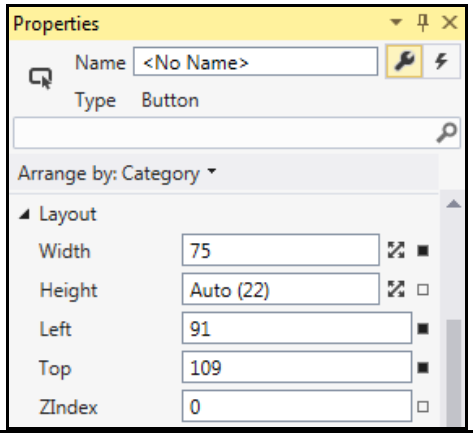
Die wichtigsten Panels sollen jetzt kurz besprochen werden.

1.1 Canvas

Das ist das einfachste Panel und sollte eigentlich nur zum Zeichnen verwendet werden. Es ordnet die Kinder **absolut** an.



Das erfolgt über die Attached Property **`Canvas.Left`**, **`Canvas.Bottom`**, **`Canvas.Right`** u. **`Canvas.Top`**, die auf die Kindelemente gesetzt wird.

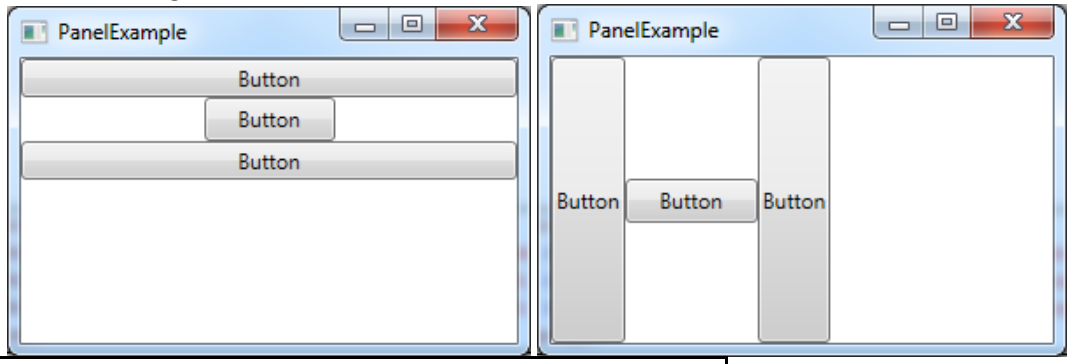



```
<Canvas>
    <Button Content="Button" Canvas.Left="151" Canvas.Top="44" Width="75"/>
    <Button Content="Button" Canvas.Left="32" Canvas.Top="38" Width="75"/>
    <Button Content="Button" Canvas.Left="91" Canvas.Top="109" Width="75"/>
</Canvas>
```

Hinweis: Man beachte, dass die Kindelemente direkt angegeben werden können und der Tag <Canvas.Children> entfallen kann, weil Children die ContentProperty aller Panels ist!

1.2 StackPanel

Dieses Layout ordnet die Controls neben- oder untereinander an. Diese Richtung wird passenderweise über die Property **`Orientation`** gesteuert.




```
<StackPanel Orientation="Horizontal">
    <Button Content="Button"/>
    <Button Content="Button" Width="75" Height="25"/>
    <Button Content="Button"/>
</StackPanel>
```

Die Controls werden dabei gestreckt, außer man gibt eine Width-Property an.

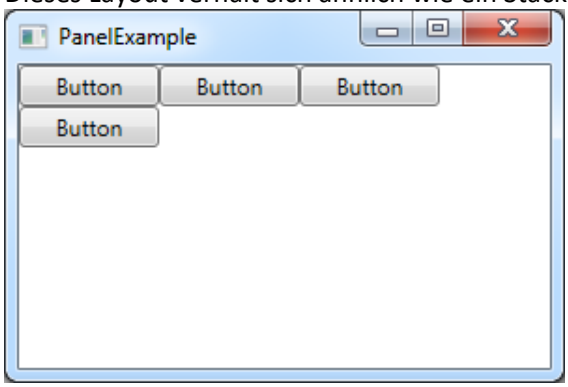
1.2.1 Code

Mit dem Wissen, dass im XAML das Element <Children> aufgrund der ContentProperty fehlen darf, könnte man mit C# obiges StackPanel so aufbauen:

```
var panel = new StackPanel { Orientation = Orientation.Horizontal };
panel.Children.Add(new Button { Content = "Button" });
panel.Children.Add(new Button { Content = "Button", Width = 75, Height = 25 });
panel.Children.Add(new Button { Content = "Button" });
this.Content = panel;
```

1.3 WrapPanel

Dieses Layout verhält sich ähnlich wie ein StackPanel, das jedoch am Ende automatisch umbricht.



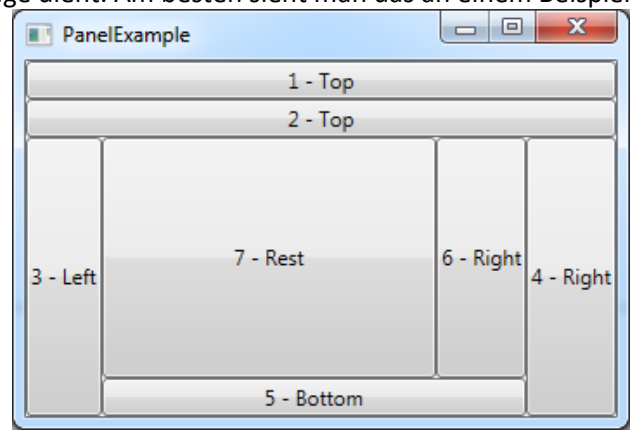

```
<WrapPanel>
    <Button Content="Button" Width="75"/>
    <Button Content="Button" Width="75"/>
    <Button Content="Button" Width="75"/>
    <Button Content="Button" Width="75"/>
</WrapPanel>
```

Die Breite bzw. Höhe für alle Kind-Elemente kann man dabei direkt im Panel einstellen. Obiges Beispiel könnte man daher auch so schreiben:

```
<WrapPanel ItemWidth="75">
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
    <Button Content="Button"/>
</WrapPanel>
```

1.4 DockPanel

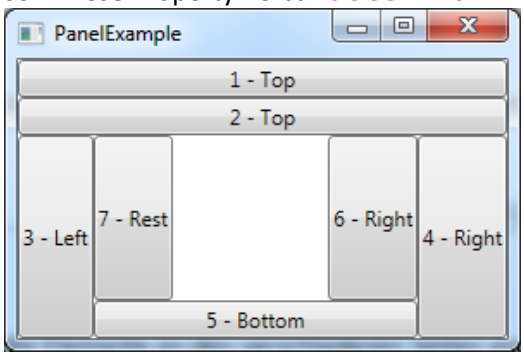
Mit diesem Layout kann man die Elemente an den verschiedenen Seiten andocken, wobei jeweils der verbleibende Platz als Grundlage dient. Am besten sieht man das an einem Beispiel:



Ähnlich wie bei Canvas wird das mit der Attached Property **Dock** gelöst:

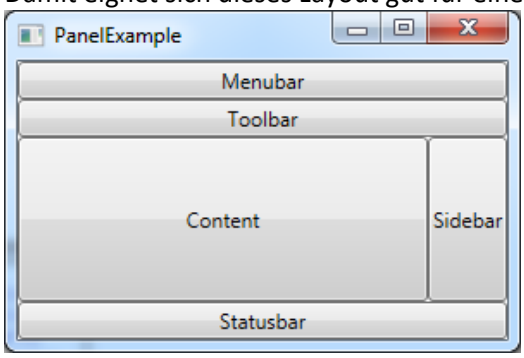
```
<DockPanel>
    <Button Content="1 - Top" DockPanel.Dock="Top"/>
    <Button Content="2 - Top" DockPanel.Dock="Top"/>
    <Button Content="3 - Left" DockPanel.Dock="Left"/>
    <Button Content="4 - Right" DockPanel.Dock="Right"/>
    <Button Content="5 - Bottom" DockPanel.Dock="Bottom"/>
    <Button Content="6 - Right" DockPanel.Dock="Right"/>
    <Button Content="7 - Rest"/>
</DockPanel>
```

Eine Property ist eventuell noch interessant – nämlich, ob das letzte Element den verbleibenden Platz ausfüllen soll. Diese Property heißt **`LastChildFill`** (Default-Wert ist **`true`**):




```
<DockPanel LastChildFill="false">
    <Button Content="1 - Top" DockPanel.Dock="Top"/>
```

Damit eignet sich dieses Layout gut für eine App mit Toolbar, MenuBar u. Statusbar:



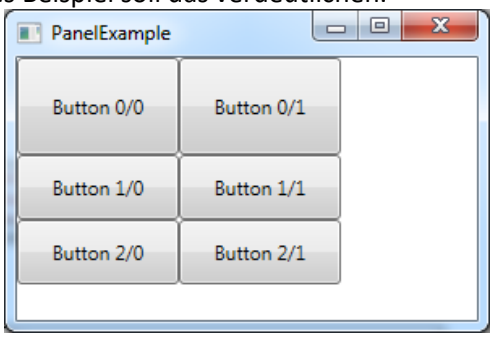

```
<DockPanel>
    <Button Content="Menubar" DockPanel.Dock="Top"/>
    <Button Content="Toolbar" DockPanel.Dock="Top"/>
    <Button Content="Statusbar" DockPanel.Dock="Bottom"/>
    <Button Content="Sidebar" DockPanel.Dock="Right"/>
    <Button Content="Content" DockPanel.Dock="Right"/>
</DockPanel>
```

1.5 Grid

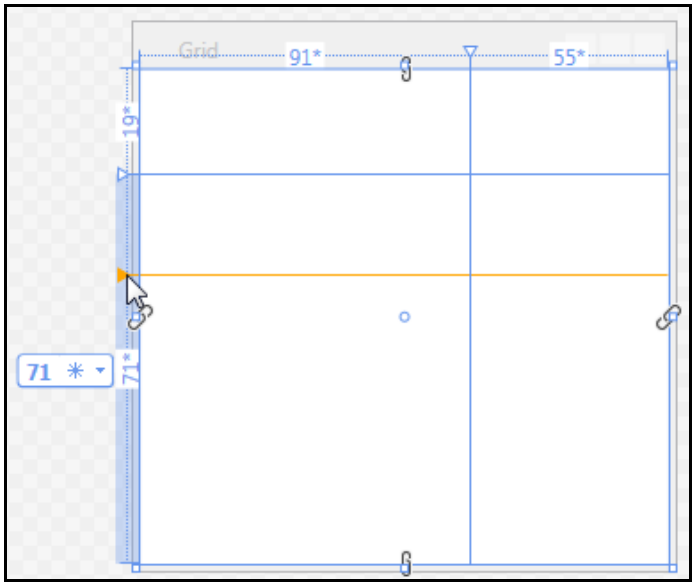
Dieses Panel ist am vielseitigsten einsetzbar und ist auch der Default beim Erzeugen eines neuen Window-Elements im Visual Studio. Wie der Name vermuten lässt, werden die Kind-Elemente in einem Gitter mit **`Grid.Row`** und **`Grid.Column`** angeordnet. Diese werden, wie schon bei den vorherigen Layouts gesehen, über Attached Properties den Kind-Elementen zugewiesen.

1.5.1 Row/ColumnDefinitions

Jedoch muss hier noch vorher definiert werden, wie viele Zeilen und Spalten das Grid haben soll. Das erfolgt über die Tags <**`Grid.RowDefinitions`**> u. <**`Grid.ColumnDefinitions`**>, in der man jede Zeile u. Spalte mit Höhe bzw. Breite definiert. Folgendes Beispiel soll das verdeutlichen.



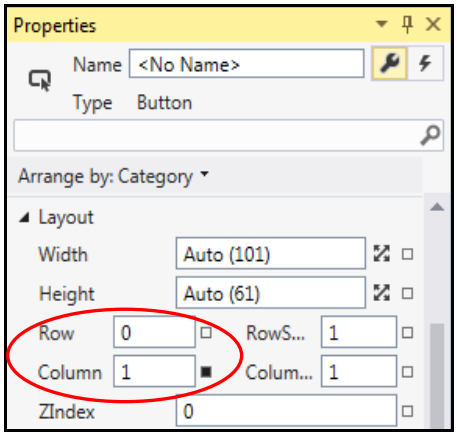
Die Definition kann man entweder direkt ins XAML schreiben oder auch graphisch im Designer generieren lassen.




```
<Grid.RowDefinitions>
  <RowDefinition Height="19*"/>
  <RowDefinition Height="71*"/>
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
  <ColumnDefinition Width="91*"/>
  <ColumnDefinition Width="55*"/>
</Grid.ColumnDefinitions>
```

1.5.2 Row/Column

Die Zuordnung im Grid kann man entweder direkt im XAML notieren, oder über die Properties zuweisen (es wird erkannt, dass sich der Button in einem Grid befindet):



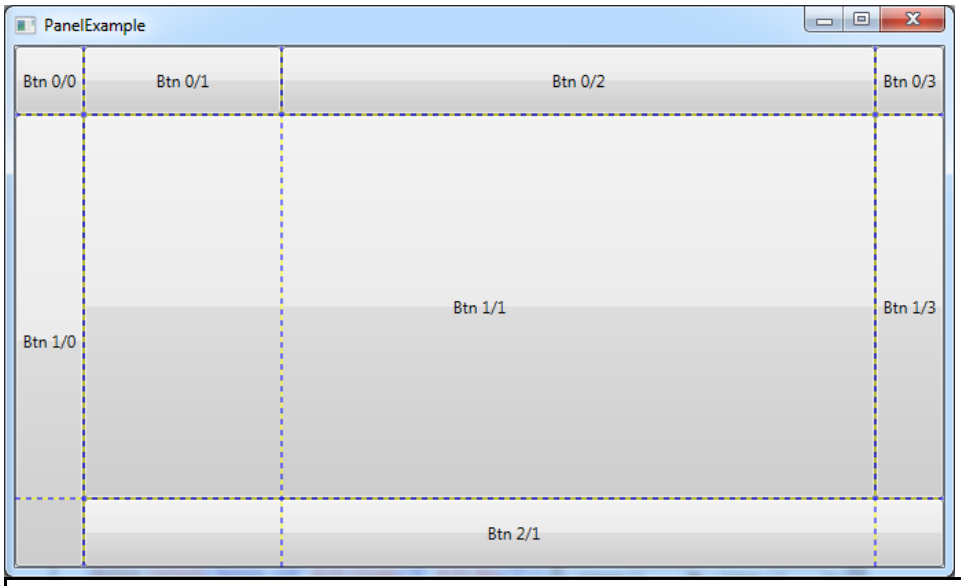

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="60"/>
        <RowDefinition Height="40"/>
        <RowDefinition Height="40"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100"/>
        <ColumnDefinition Width="100"/>
    </Grid.ColumnDefinitions>
    <Button Content="Button 0/0"/>
    <Button Content="Button 0/1" Grid.Column="1"/>
    <Button Content="Button 1/0" Grid.Column="0" Grid.Row="1"/>
    <Button Content="Button 1/1" Grid.Column="1" Grid.Row="1"/>
    <Button Content="Button 2/0" Grid.Column="0" Grid.Row="2"/>
    <Button Content="Button 2/1" Grid.Column="1" Grid.Row="2"/>
</Grid>
```

Der Default für Grid.Column und Grid.Row ist jeweils 0. Elemente, die in dieselbe Zelle eingetragen sind, werden einfach **übereinander** gezeichnet.

Bemerkungen:

- mit **`ShowGridLines`** kann man die Gitterlinien anzeigen lassen. Das ist vor allem während des Entwickelns hilfreich
- Soll eine Zelle über mehrere Zeilen oder Spalten gehen, kann man im Element mit **`Grid.ColumnSpan`** bzw. **`Grid.RowSpan`** einstellen. Das funktioniert also praktisch wie in HTML.
- Eine Breite/Höhe von **`"*"`** bedeutet, dass der restliche Platz ausgefüllt werden soll. Dabei kann man auch Gewichtungen angeben, wenn sich mehrere Elemente den restlichen Platz aufteilen wollen. Aber das kennt man ja bereits von Android.

Beispiel:




```
<Grid ShowGridLines="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="50"/>
        <RowDefinition Height="*"/>
        <RowDefinition Height="50"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="50"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="3*"/>
        <ColumnDefinition Width="50"/>
    </Grid.ColumnDefinitions>
    <Button Content="Btn 0/0" Grid.Column="0" Grid.Row="0"/>
    <Button Content="Btn 0/1" Grid.Column="1" Grid.Row="0"/>
    <Button Content="Btn 0/2" Grid.Column="2" Grid.Row="0"/>
    <Button Content="Btn 0/3" Grid.Column="3" Grid.Row="0"/>
    <Button Content="Btn 1/0" Grid.Column="0" Grid.Row="1" Grid.RowSpan="2"/>
    <Button Content="Btn 1/1" Grid.Column="1" Grid.Row="1" Grid.ColumnSpan="2"/>
    <Button Content="Btn 1/3" Grid.Column="3" Grid.Row="1"/>
    <Button Content="Btn 2/1" Grid.Column="1" Grid.Row="2"/>
    <Button Content="Btn 2/1" Grid.Column="1" Grid.Row="2"  Grid.ColumnSpan="3"/>
</Grid>
```

Hier teilen sich die 2. u. 3. Spalte die restliche Breite im Verhältnis 1:3, also ¼ bzw. ¾.

1.6 Layout-Eigenschaften

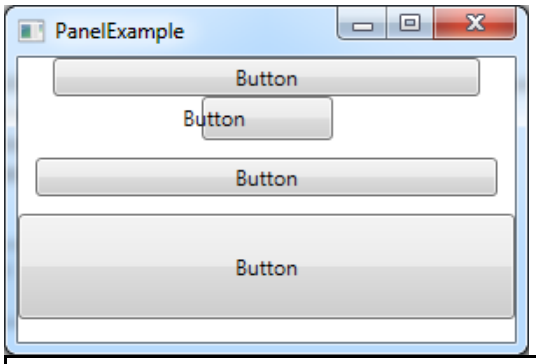
Die wichtigsten Layout-Eigenschaften der Controls, die das Layout beeinflussen, werden hier kurz vorgestellt.

1.6.1 Margin/Padding

Margins bestimmen den **äußeren** Rand, **Padding** den **inneren** Rand eines Controls. Für diese gilt jeweils:

- Können für jede Seite extra angegeben werden.
- Wird nur ein Wert angegeben, gilt dieser für alle vier Seiten.
- Kann man im Property-Fenster einstellen
- Werte können negativ sein.

Beispiel:




```
<StackPanel Orientation="Vertical">
    <Button Content="Button" Margin="20,0,20,0"/>
    <Button Content="Button" Width="75" Height="25" Padding="-60,0,0,0"/>
    <Button Content="Button" Margin="10"/>
    <Button Content="Button" Padding="20"/>
</StackPanel>
```

Property-Window:

1.6.2 Visibility

Die Sichtbarkeit funktioniert ähnlich wie bei Android und wird über die Property **`Visibility`** eingestellt – es gibt drei Zustände:

- **`Visible`**: Control ist sichtbar
- **`Hidden`**: Control wird nicht angezeigt, der Platz bleibt aber reserviert
- **`Collapsed`**: Control wird nicht angezeigt und wird beim Layoutprozess auch nicht berücksichtigt

1.6.3 Alignment

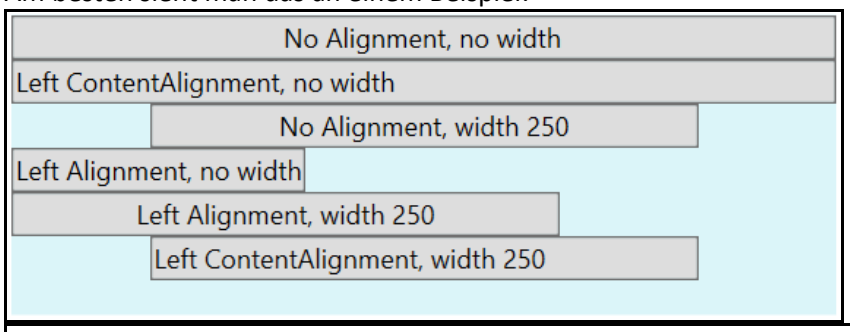
Hat ein Control mehr Platz zur Verfügung, als es braucht, kann man es horizontal oder vertikal ausrichten. Ähnlich wie bei Margin u. Padding gibt es dazu Properties für die Ausrichtung

- nach außen, **`HorizontalAlignment`** und **`VerticalAligment`**
- nach innen, **`HorizontalContentAlignment`** und **`VerticalContentAligment`**.

Mögliche Werte sind dabei die selbsterklärenden Angaben

- horizontal: **`Left`**, **`Center`**, **`Right`**, **`Stretch`**
- vertikal: **`Top`**, **`Center`**, **`Bottom`**, **`Stretch`**

Am besten sieht man das an einem Beispiel:




```
<StackPanel  Height="136" Width="376" Background="#FFDBF5F9">
  <Button Content="No Alignment, no width" />
  <Button Content="Left ContentAlignment, no width" HorizontalContentAlignment="Left" />
  <Button Content="No Alignment, width 250" Width="250" />
  <Button Content="Left Alignment, no width" HorizontalAlignment="Left"/>
  <Button Content="Left Alignment, width 250" HorizontalAlignment="Left" Width="250" />
  <Button Content="Left ContentAlignment, width 250" HorizontalContentAlignment="Left" Width="250" />
</StackPanel>
```

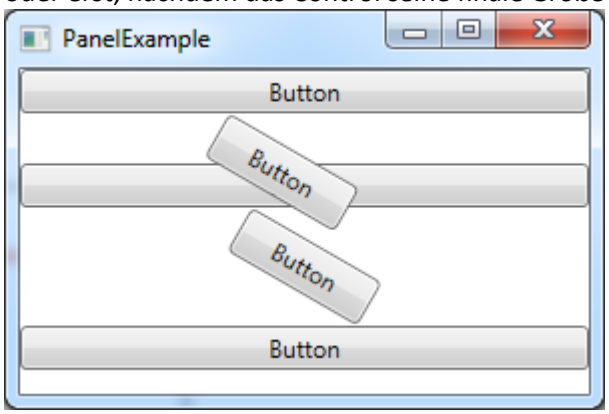
2 Zusatzinfos

2.1 Transformationen

Sowohl vor Measure als auch vor Arrange hat man die Möglichkeit, Controls noch in ihrer Lage zu verändern. Das erfolgt über eine Matrix. Es gibt aber auch sprechende Methoden, die sehr einfach ohne Matrix-Kenntnisse zu verwenden sind:

- **RotateTransform**: dreht
- **ScaleTransform**: ändert die Größe
- **TranslateTransform**: verschiebt
- **MatrixTransform**: benutzt eine Matrix

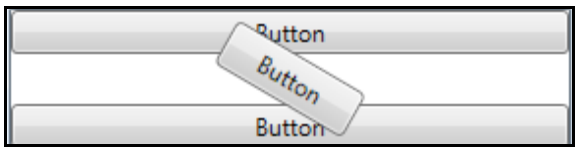
Dabei hat man z.B. bei einer Drehung die Möglichkeit, vorher zu drehen und somit mehr Platz zu beanspruchen, oder erst, nachdem das Control seine finale Größe vom Layoutprozess zugewiesen bekommen hat:



Daher gibt es zwei Tags bzw. Klassen: **RenderTransform** bzw. **LayoutTransform**:

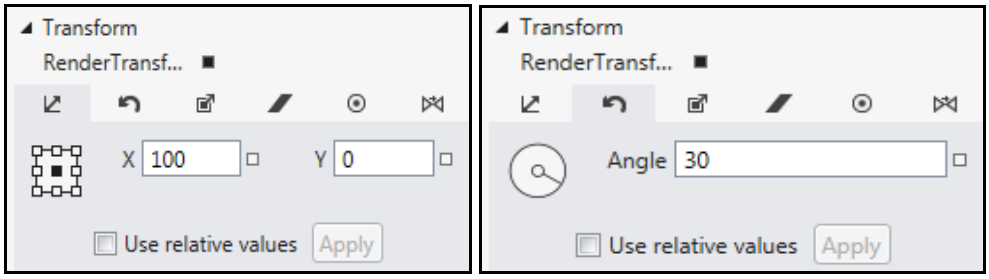
```
<StackPanel Orientation="Vertical">
    <Button Content="Button" />
    <Button Content="Button" Width="75" Height="25" Panel.ZIndex="1">
      <Button.RenderTransform>
            <RotateTransform Angle="30" />
        </Button.RenderTransform>
    </Button>
    <Button Content="Button" />
    <Button Content="Button" Width="75" Height="25">
      <Button.LayoutTransform>
            <RotateTransform Angle="30" />
        </Button.LayoutTransform>
    </Button>
    <Button Content="Button" />
</StackPanel>
```

Dabei kann man bei RotateTransform noch einstellen, um welchen Punkt gedreht werden soll. Dazu gibt man mit **RenderTransformOrigin** zwei Werte zw. 0 u. 1 an, die die relative horizontale u. vertikale Position angeben:




```
<Button Content="Button" Width="75" Height="25"
        RenderTransformOrigin="0.5,0.5"
        Panel.ZIndex="1">
    <Button.RenderTransform>
        <RotateTransform Angle="30" />
    </Button.RenderTransform>
</Button>
```

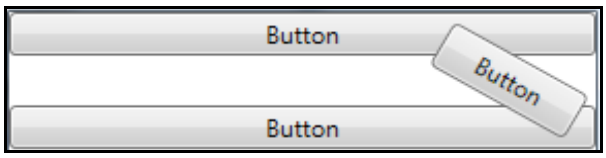
Man wird auch im Properties-Fenster unterstützt:



Speziellere Transformationen kann man eben mit **`MatrixTransform`** bzw. der Property **`RenderTransform`** erreichen. So könnte man obige Drehung mit folgender Matrix erreichen:

```
<Button Content="Button" Width="75" Height="25"
        RenderTransformOrigin="0.5,0.5"
        RenderTransform="0.866,0.5,-0.5,0.866,0,0"
        Panel.ZIndex="1">
</Button>
```

Mit einer **`TransformGroup`** kann man mehrere Transformationen kombinieren:



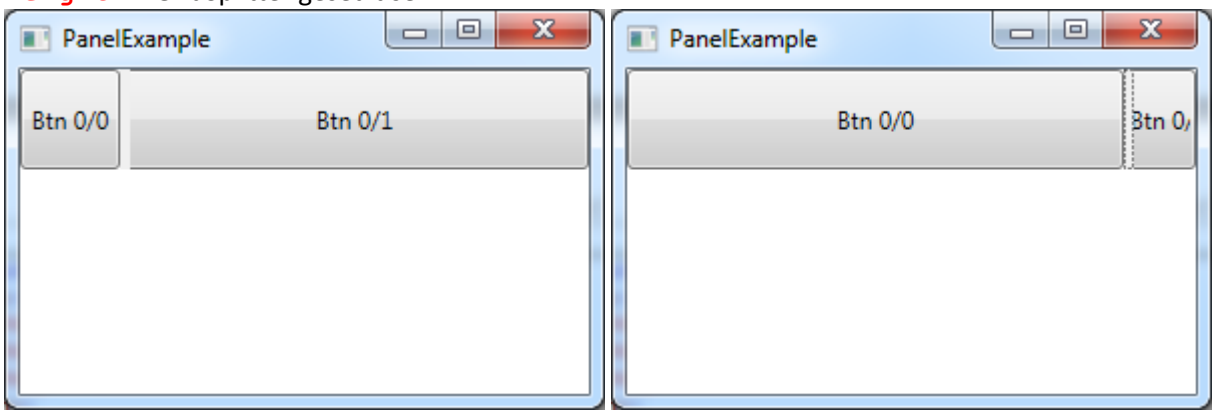

```
<Button Content="Button" Width="75" Height="25"
        RenderTransformOrigin="0.5,0.5"
        Panel.ZIndex="1">
    <Button.RenderTransform>
        <TransformGroup>
            <RotateTransform Angle="30" />
            <TranslateTransform X="100"/>
        </TransformGroup>
    </Button.RenderTransform>
</Button>
```

Dabei ist die Reihenfolge natürlich wichtig!

2.2 Grid

2.2.1 GridSplitter

Mit einem **`GridSplitter`** kann man zur Laufzeit die Breite/Höhe von Zeilen oder Spalten ändern. Dazu wird dieser einfach als Kindelement dem Grid hinzugefügt. Damit er sichtbar ist, muss entweder die **`Width`** oder die **`Height`** im GridSplitter gesetzt sein.



```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="50"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="50"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Button Content="Btn 0/0" Grid.Column="0" Grid.Row="0"/>
    <Button Content="Btn 0/1" Grid.Column="1" Grid.Row="0"/>
    <GridSplitter Grid.Column="1"
                  HorizontalAlignment="Left"
                  Width="5"/>
</Grid>
```

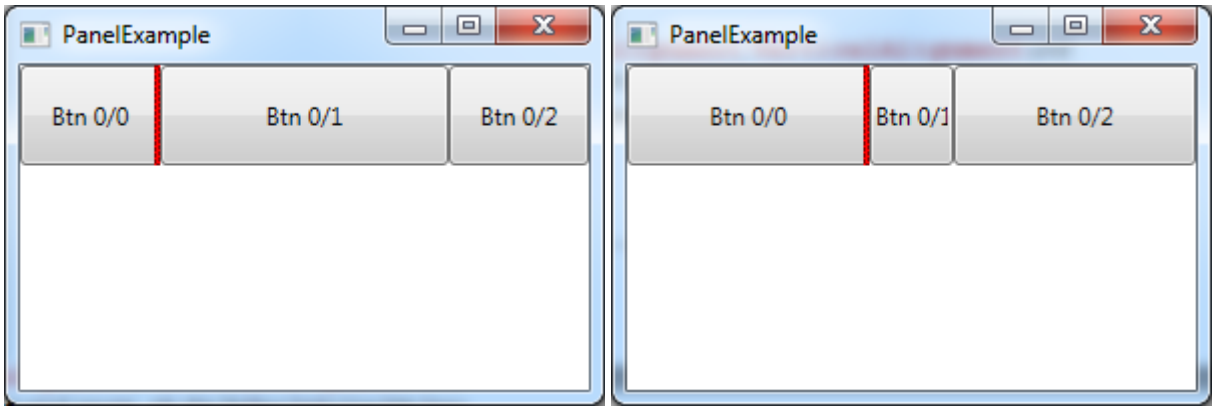
Man kann beim GridSplitter über die Properties **`HorizontalAlignment`**, **`VerticalAlignment`** und **`ResizeBehavior`** einstellen, ob er links/rechts/mittig platziert wird sowie, ob die Zellen links/rechts bzw. oben/unten beeinflusst werden. Bei Bedarf ausprobieren bzw. nachschlagen.

2.2.2 SharedSizeGroup

Mit der Property **`SharedSizeGroup`** kann man Zellen Spalten/Zeilen gleichzeitig in Breite/Höhe ändern. Diese Property verlangt einen String. Alle Spalten/Zeilen mit derselben Gruppe erhalten dann die gleiche Breite/Höhe. Üblicherweise gibt man dann für alle außer der ersten Zelle der Gruppe die Breite/Höhe mit **`"Auto"`** an (kann aber auch auf einen anderen Wert gesetzt werden, wie man unten sieht).
Zusätzlich muss für das Grid noch die Property **`IsSharedSizeScope="True"`** gesetzt werden.

```
<Grid IsSharedSizeScope="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="50"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="50" SharedSizeGroup="quaxi"/>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="70" SharedSizeGroup="quaxi"/>
    </Grid.ColumnDefinitions>
    <Button Content="Btn 0/0" Grid.Column="0" Grid.Row="0"/>
    <Button Content="Btn 0/1" Grid.Column="1" Grid.Row="0"/>
    <Button Content="Btn 0/2" Grid.Column="2" Grid.Row="0"/>
    <GridSplitter Width="3"
                  Grid.Column="0"
                  HorizontalAlignment="Right"
                  Background="Red"/>
</Grid>
```

Und so sieht's aus:



2.3 Layoutprozess

Ganz kurz möchte ich noch auf die Schritte eingehen, die ein Panel bei der Anordnung der Kindelemente durchführt. Diese sind (in dieser Reihenfolge):

1. **`Measure`**: wie groß möchten die Kindelemente gerne sein?
2. **`Arrange`**: die Kinder bekommen den tatsächlich verfügbaren Platz
3. **`OnRender`**: die Kinder werden gezeichnet (bzw. zeichnen sich selbst)

Ändert sich bei einem eigenen UserControl etwa die Größe, werden alle drei Schritte von WPF durchgeführt, und zwar in der Methode **`LayoutManager.UpdateLayout`**. Diese Methode sieht so aus (ich habe alle unwesentlichen Teile gelöscht):

```
internal void UpdateLayout()
{
  //...
  UIElement currentElement = null;
  while (hasDirtiness || _firePostLayoutEvents)
  {
    _isInUpdateLayout = true;
    while (true)
    {
      currentElement = MeasureQueue.GetTopMost();
      if (currentElement == null) break; //exit if no more Measure candidates
      currentElement.Measure(currentElement.PreviousConstraint);
    }
    while (MeasureQueue.IsEmpty)
    {
      currentElement = ArrangeQueue.GetTopMost();
      if (currentElement == null) break; //exit if no more Measure candidates
      Rect finalRect = getProperArrangeRect(currentElement);
      currentElement.Arrange(finalRect);
    }
    if (!MeasureQueue.IsEmpty) continue;
    _isInUpdateLayout = false;
    //...
  }
}
```

Alle Controls des WPF überschreiben daher die folgenden zwei Methoden:

- **`Size MeasureOverride(Size availableSize)`**: berechnet die gewünschte Größe des Elements und gibt die tatsächlich benötigte Größe zurück
- **`Size ArrangeOverride(Size finalSize)`**: hier werden die Kindelemente angeordnet. Die tatsächliche Größe wird in der Property **`RenderSize`** gespeichert (in der internen WPF-Methode Arrange).

Die tatsächliche Größe eines Controls kann man immer über die Properties **`ActualWidth`** u. **`ActualHeight`** abfragen, die genau den Werten von **`RenderSize`** entsprechen.