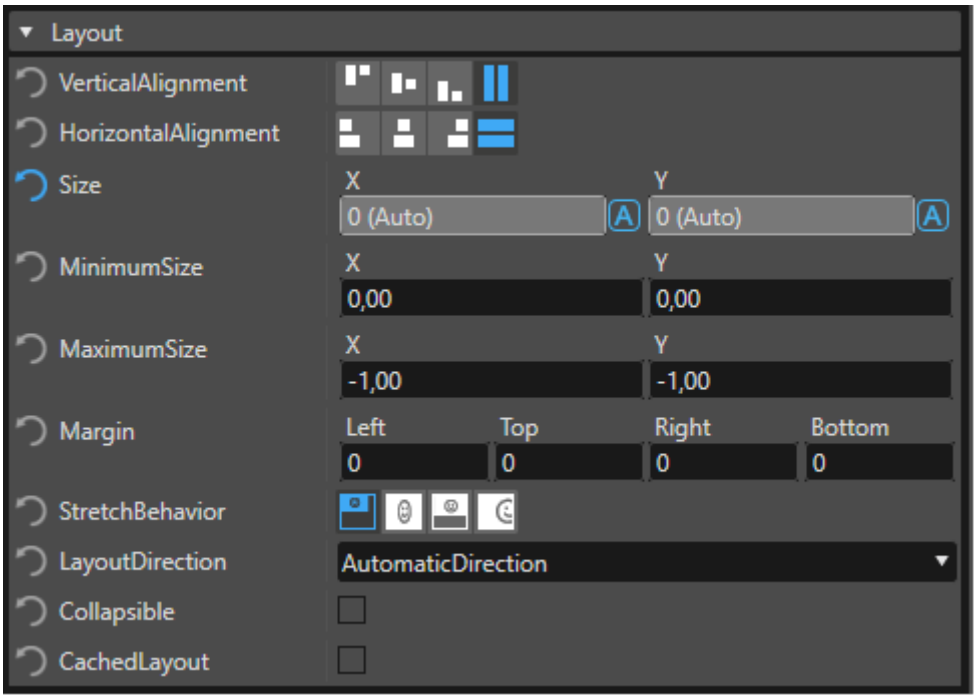
Functional Overview

- Basic Layouter Properties
- Canvas Text - Size Property
- Types of Layouters
- Culture Dependent Layout
- Automatic Resizing Layouter - CGI Studio vs. WPF

Basic Layouter Properties

Description

CGI-Studio provides various layouters with certain properties, which can be configured. Properties that apply to all of the CGI-Studio layouters are described below.

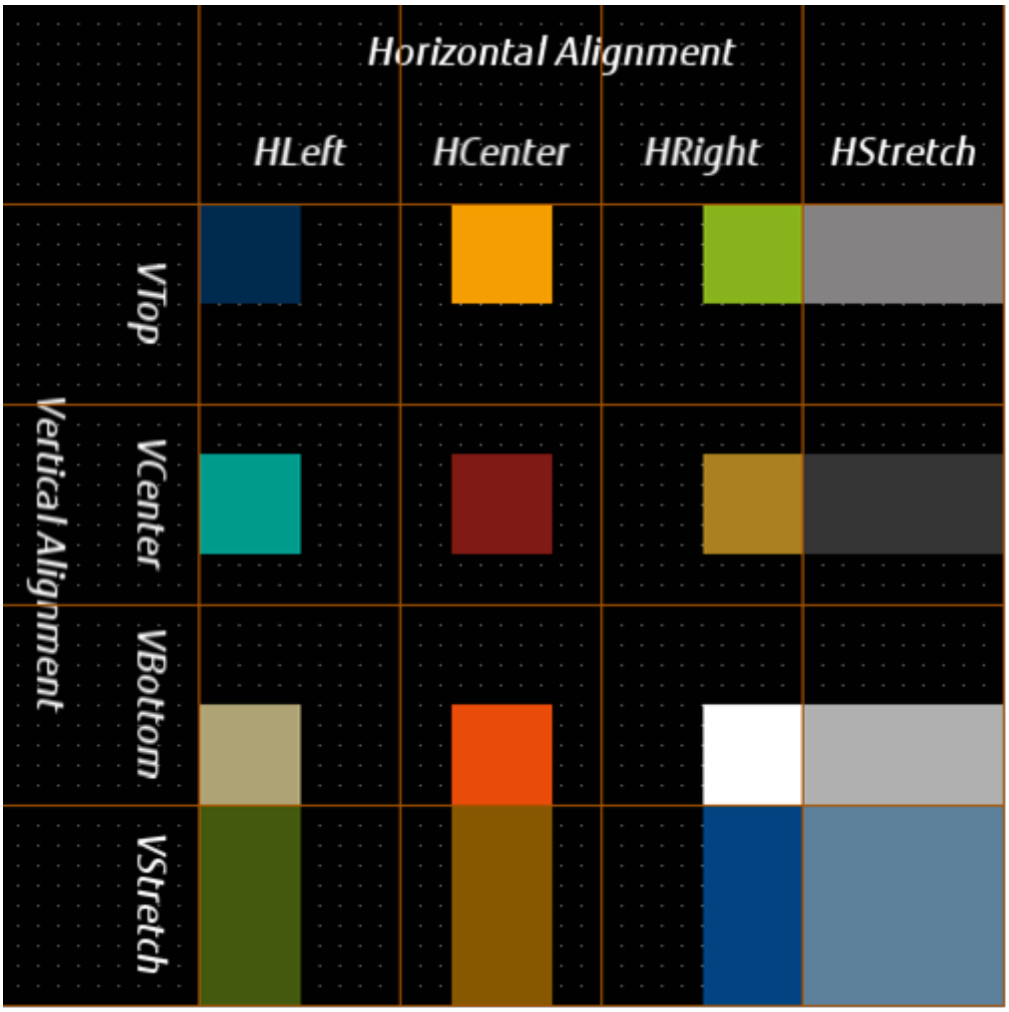


Vertical Alignment and Horizontal Alignment

Vertical and Horizontal Alignment define how the layouter should align the nodes within their child layout area. The size and position of the child layout area depends on the chosen layouter. But, the alignment within this area is defined by these two properties.

Open "Import" dialog to import an FBX file via menu or toolbar.

If nothing is specified for your use case, you should use the default values VStretch and HStretch, because these settings will delegate the child layout area to the child layouter. When choosing VStretch or HStretch please check the setting of the additional property *Stretch Behavior*.



Size

The Size property allows defining a fixed size for this child node. The value defined in the Size property overrules the value provided by the layout area. This means if a layouter only provides an area with a width of 10 (due to size setting on a parent node) a size of (20,-1) will force the layouter to provide a child layout area with a width of 20 for this child node. Use this property if you want a layouter to arrange its child elements freely within a defined range. The layouter has to use exactly this area (if not overruled by the Minimum Size or Maximum Size property).

The value -1 indicates that this value is not set for this node and has to be determined by other properties or the child layout area.

Minimum Size

The Minimum Size property defines the minimum size of the child node. This property overrules the Maximum Size property, the Size property and the layout area. So, if you define a Size or Maximum Size that is smaller than the Minimum Size then the Size is limited to the Minimum Size (if still not overruled, the layout area will still be applied). Use this property if you want a layouter to arrange its child elements freely within a limited range (e.g. if there is a specification for an area that has to consume a minimal size but is allowed to be bigger then use this specification as Minimum Size).

Maximum Size

The Maximum Size property defines the maximum size of the child node. This property overrules the Size property and the layout area. So, if you define a Size that is larger than the Maximum Size then the Size is limited to the Maximum Size (if still not overruled the layout area will still be applied). Use this property if you want a layouter to arrange its child elements freely within a limited range (e.g. if there is a specification for an area which everything has to fit into but should use minimum space then use this specification as Maximum Size).

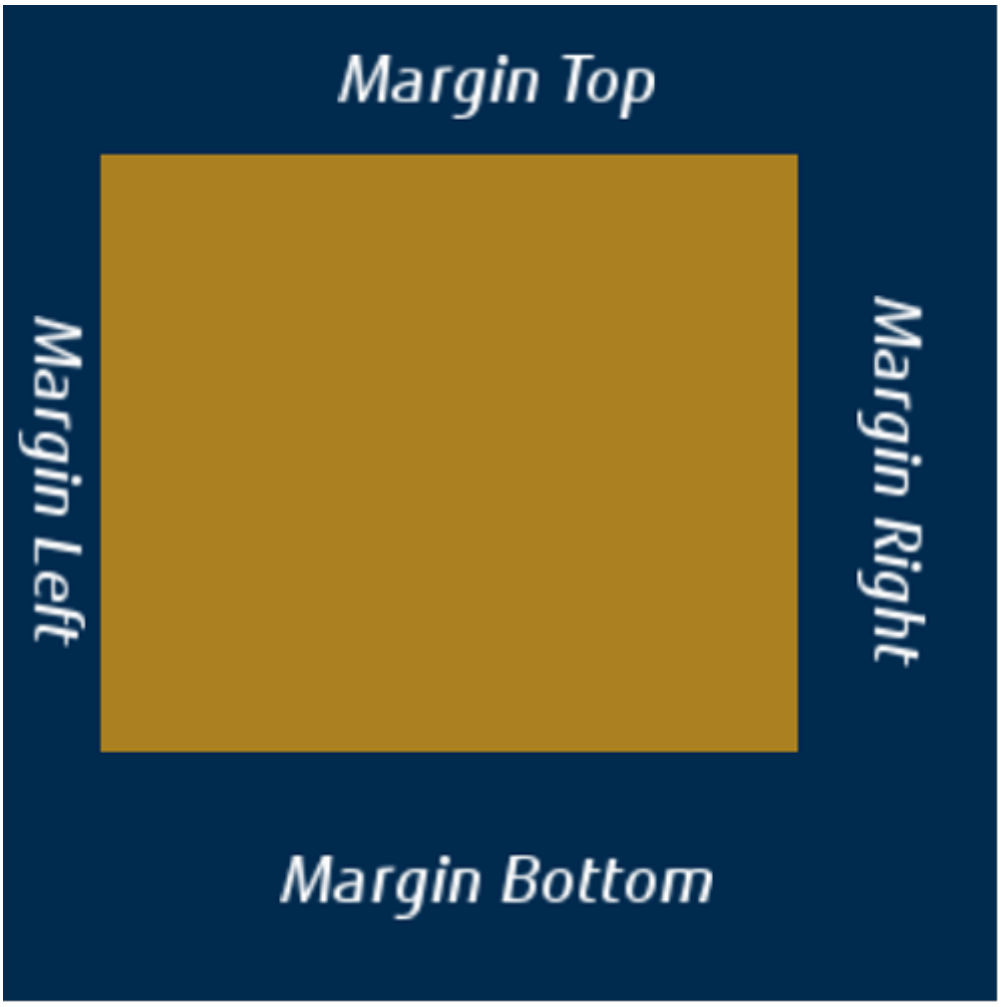
Similar to the Size property -1 can be used to indicate that the value is not set.

Margin

The Margin property is used to define the offset on the top, bottom, left and right that have to be considered by the layouter when arranging its child elements. You can implement relative positioning within the layout area after the alignment is applied. For example if you want to layout a node right and top aligned but position it 10 pixel down from top and 20 pixel more left than on the right border then you set the top Margin to 10 and the right Margin to 20.

> Setting the position manually is impossible with layouters, because the configured position will be overwritten by the layouter.

The values for Margin can be set separately for left, top, right and bottom so the margin can be asymmetric (see image below).



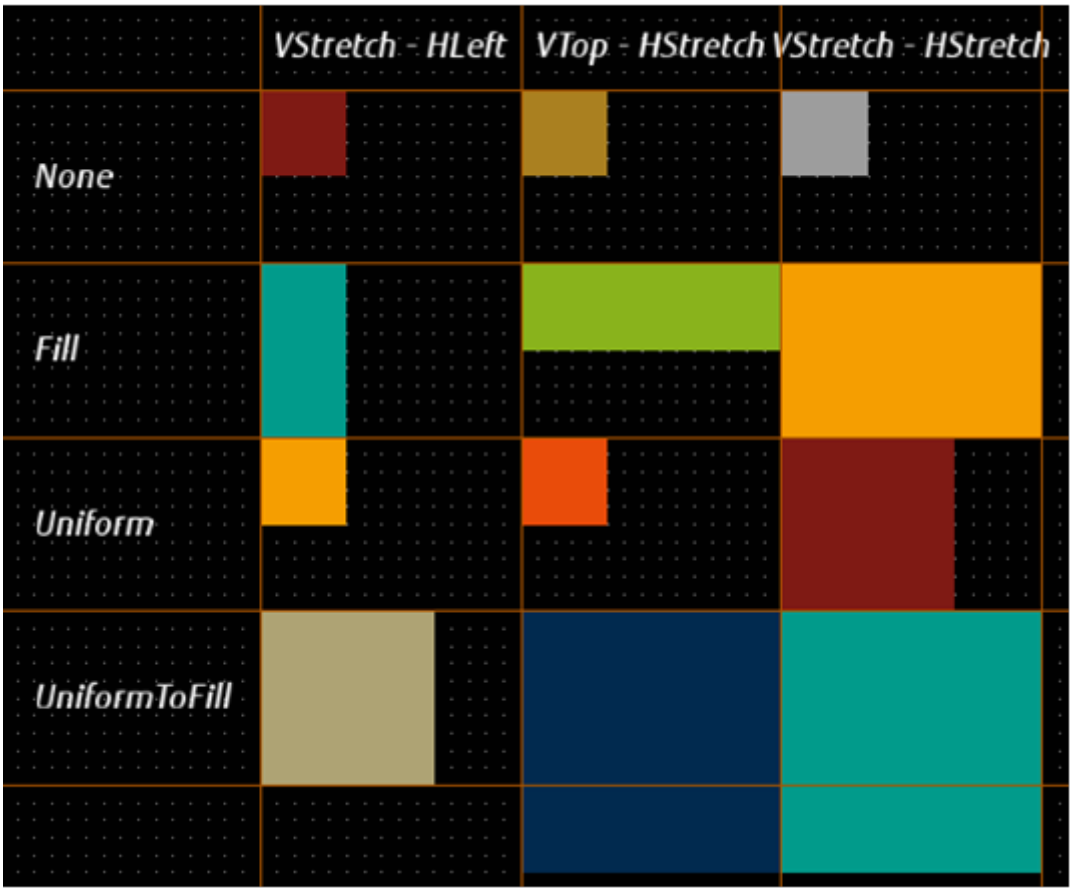
Stretch Behavior

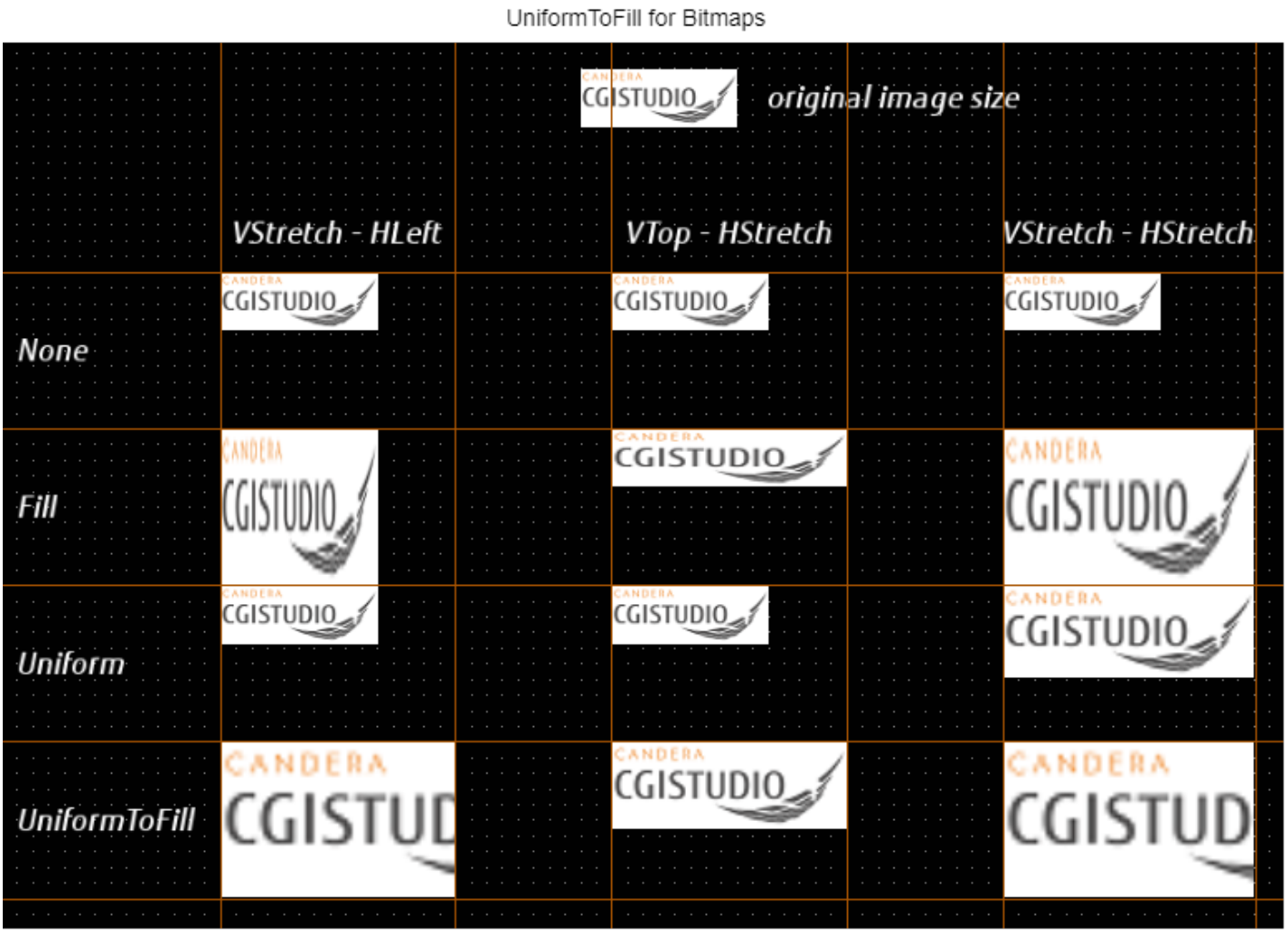
The Stretch Behavior property is only useful on RenderNodes because it defines how they have to be adjusted to the nodes' Scale property to fit into the layout area. Use None if the node has to keep its normal size. Use Fill if you want to stretch it in the axis which is configured for stretching. Use Uniform to fit into the layout area but keep its original proportion. Use UniformToFill to fill the axis that is configured for stretching but keep its original proportion.

> If a stretch behavior is set for a node, it will consume the area accordingly. So, if the size is not limited there may be no space left for other content. Therefore, it should be limited with one of the size properties either directly or indirectly by one of the node's parents

Using the option UniformToFill the RenderNode can exceed its layout area (see image below).

UniformToFill for SolidColorNodes





Layout Direction

The Layout Direction property defines how the alignment is handled.

- **LeftToRightDirection**: The alignment is fixed to left-to-right
- **Right-To-LeftDirection**: The alignment is fixed to right-to-left
- **AutomaticDirection**: This is the default layout direction. The default interpretation of AutomaticDirection is to inherit the parent's value (InheritDirection). If no value is set on any parent, and therefore nothing can be inherited, the culture layout direction is used (CultureDirection).
  You can change the interpretation of the AutomaticDirection by calling

  ```
  Layouter::SetAutomaticDirectionBehavior()
  ```

- **InheritDirection**: Alignment is inherited from the parent layout direction. If no layout direction is set on any parent, the direction of the current culture is used.
- **CultureDirection**: The alignment depends on the culture and is independent of the parent layout direction.
  - Right-to-left for Arabic and Hebrew
  - Left-to-right for most languages (e.g. English, German...)

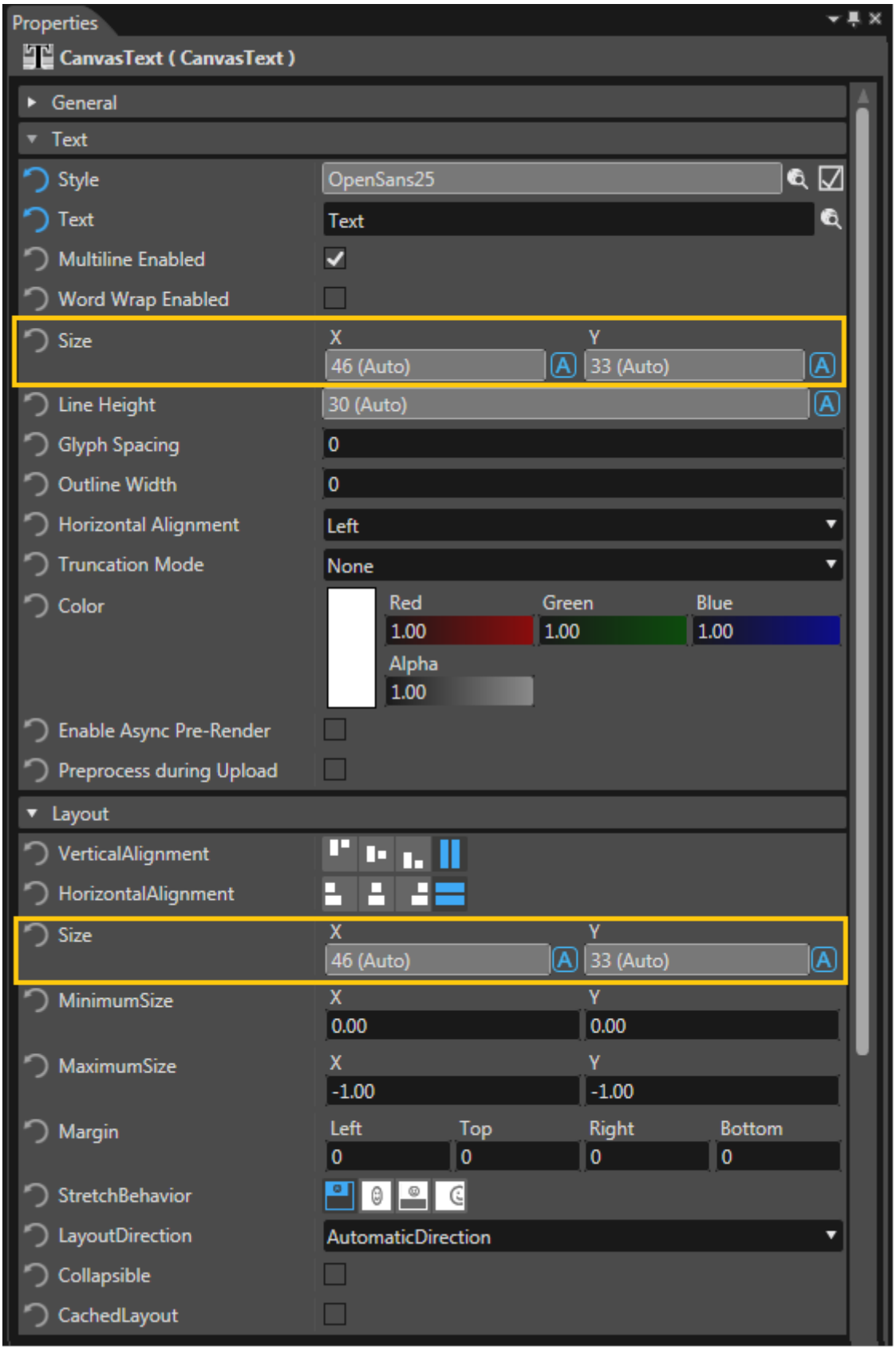
Collapsible

A collapsible node won't be considered during the calculation of the layout if its rendering is disabled. Its rendering can be disabled by unchecking the property Enable Rendering. The property *Collapsible* will ignore calculations for position and size in the layout. It activates if effective rendering of the node is disabled and applies to all child nodes.

Canvas Text - Size Property

The Canvas Text provides a huge range of configuration possibilities. Some of them offer a great opportunity for customization but also increase the complexity level.
This section will look into one of the more complex features of the text: The ‘Size’ property and gives a short insight into the combination of size and other properties as the behavior will adjust accordingly.
In total there are up to three different size properties on each text:

1. The Text::Size property
2. The Layout::Size property
3. And an additional layout size (e.g. the column width or row height when using a grid layout)



The Text::Size (1.) property itself is mainly used to layout the text (text measuring) and provides text alignment, word wrapping and other text features to applications which have not set layout to enabled via CMake (CANDERA_LAYOUT_ENABLED is not set).
Aside from that, this is the most basic way to define the dimensions in which text is rendered.

For the two layout properties (2. and 3.) it is possible to set only one of them which will result in the same output.
E.g. setting the grid column width (3.) to 100 px is equal to setting the Layout::Size (2.) to 100 px. However, in the first variant the layout size is bound to the underlying layout, whereas in the second variant it is bound to the text.
If both are set, both will have impact on the layout. The grid column/row size (3.) will have higher priority when calculating the complete layout but the Layout::Size (2.) will still have some impact.

The Text::Size (1.) is the normal way to define a text's dimensions. This is comparable to setting the size of a SolidColorBrush effect. The reference image here shows exactly that: A SolidColorBrush effect (green) which has a greater width than the layout size of the grid (grey). Therefore, the second element overlaps the first one as it is laid out based on the layout dimensions and not the size dimensions. This effect is only visible when clipping is disabled. Otherwise, the SolidColorBrush will be clipped at the layout's size.

For the text, it will be rendered similarly into this area and then this block will be laid out. So overlapping might occur. Use cases:

1. Text is rendered ignoring layout settings – only using its own settings
2. Text should be within the layout boundaries as first requirement. And if Text::Size smaller than Layout::Size, it should be placed within Text::Size
3. Text should be positioned within the Layout::Size.

SceneComposer UI

The SceneComposer supports the user who is adjusting the size of the Layouter by providing a visual helper - an orange rectangle that makes it possible to visualize the configured dimensions.

Impact of size on different properties:

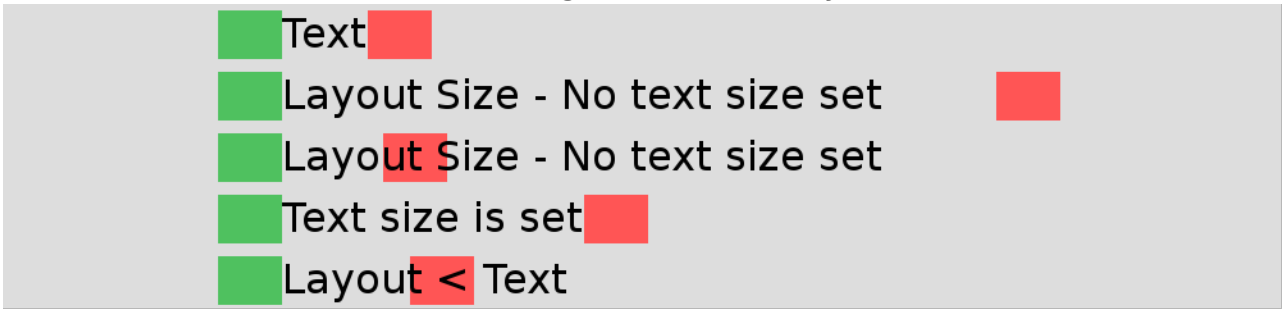
Alignment: The text has a property for horizontal alignment. This property aligns the text within Text::Size. If no Text::Size is set, the Layout::Size is used. The layout has a horizontal and vertical layout alignment property which is used to align the 'block' of the measured text within the Layout::Size. The defined layout alignment can be propagated to the text alignment when the text alignment is set to 'Auto'.
Otherwise, it is possible to right align the text within Text::Size but left align this block of right aligned text within Layout::Size. The properties will be inverted if the layout direction is set to 'right to left'.

Word wrap and truncation: Both properties prepare a text for use cases in which they do not fit into a size. Therefore, enabling one of them forces the text to fit into the provided smallest size (Layout::Size or Text::Size).

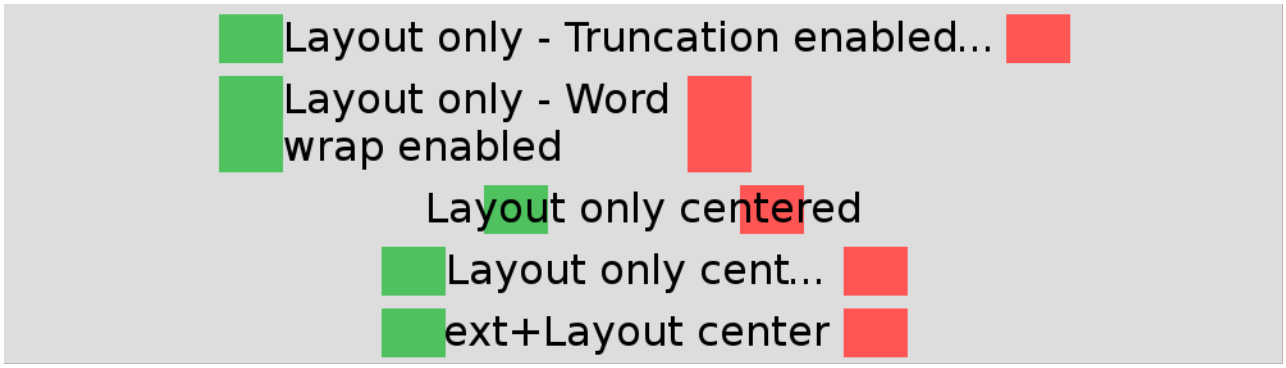
Examples

The following image shows some use cases to provide a better understanding of the layout behavior with different sizes. Every use case is based on a StackPanel with a CanvasSprite, a CanvasText and another CanvasSprite.

1. Default Text: No sizes set. The text is measured and the result is used for layout.
2. The layout size is set to a huger value than the text actually requires. This result is expected when the text is smaller than the layout size or the text size is limited to the shown size. The layout size defines the size for the layout in the end.
3. Same case as 2. But the layout size is smaller. There is no explicit restriction for the text itself.
   Therefore, the layout area is used for the layout process but the text will be drawn overlapping the other nodes of the layout as it requires more space.
4. Layout size is not set but text size is set to the actual size of the text. The layout uses this size to position the text.
5. Both sizes are set, but the Text::Size is greater than the layout size.



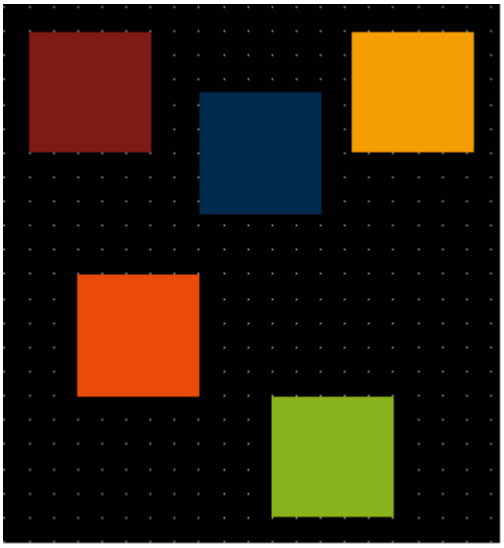
6. The layout size is set and Text::Size is not. The text would normally write over other elements. However, truncation is enabled.
7. Same as case as 6. but word wrap is enabled.
8. The Layout::Size is set and the layout alignment is center. It overlaps other elements but the text keeps being centered.
9. Same as 8. but truncation is enabled. Now the text is not centered anymore, as it requires more space than available. The beginning of the text is shown until the end of space is reached.
10. Layout::Size as well as Text::Size is set. Normally, the beginning of the text would be shown. However, the text alignment is also centered. Therefore, the center part of the text is visible.


Layout only - Truncation enabled...
Layout only - Word wrap enabled
Layout only centered
Layout only cent...
ext+Layout center

Types of Layouters

Default Layouter

When no layouter is set (Layouter Type is set to None), the default layouter is used. Absolute positioning with the default layouter is not very flexible for dynamic structures. A better layout behavior that e.g. adapts to dynamic changes of elements in their size can be achieved using the defined CGI-Studio layouters. Using the Default Layouter all nodes are positioned with absolute coordinates in pixel. In this example below, each of the five colored squares is positioned with an absolute x and y coordinate.



Similiar to WPF Canvas

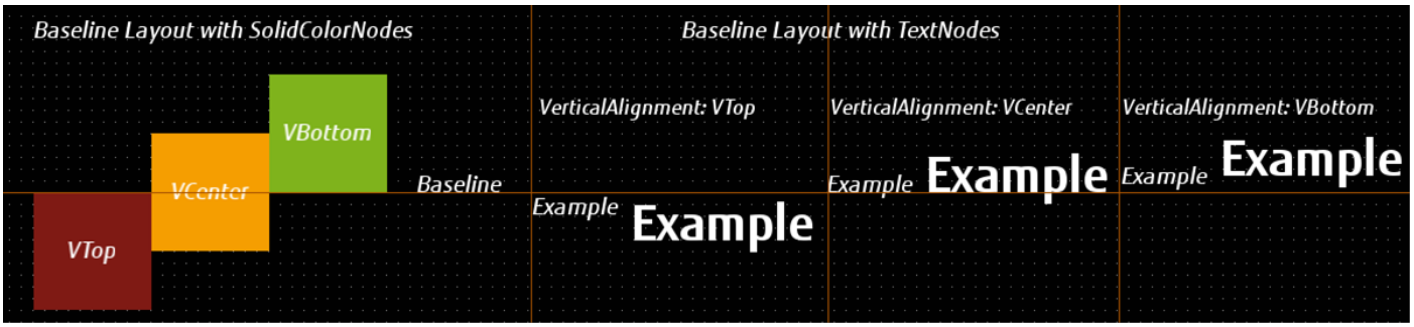
The CGI Studio Default Layouter is similar to the WPF Canvas, which also allows absolute positioning. The same layout as above can be realized with the following configuration using WPF canvas:

```
<Canvas>
    <Rectangle Height="50" Width="50" Fill="Red" Canvas.Left="0" Canvas.Top="0"></Rectangle>
    <Rectangle Height="50" Width="50" Fill
="Green" Canvas.Left="100" Canvas.Top="150"></Rectangle>
    <Rectangle Height="50" Width="50" Fill="Blue" Canvas.Left="70" Canvas.Top="25"></Rectangle>
    <Rectangle Height="50" Width="50" Fill
="DarkOrange" Canvas.Left="133" Canvas.Top="0"></Rectangle>
    <Rectangle Height="50" Width="50" Fill
="Gray" Canvas.Left="20" Canvas.Top="100"></Rectangle>
</Canvas>
```

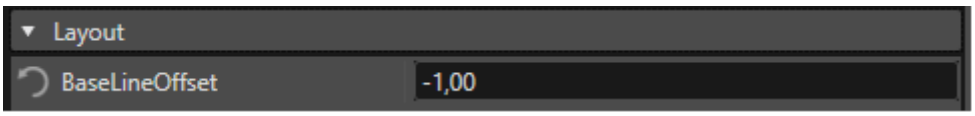
Baseline Layouter

The Baseline Layouter realizes a stacked layout in the horizontal direction. It behaves similar to the Horizontal Stack Layouter but it aligns the objects to a line instead of aligning them to the client area. This layouter offers a property Base Line Offset to configure the position of the Baseline. The main purpose of this layouter is to align texts with different sizes in various ways. Elements can be aligned along the top (VTop), the center (VCenter) or the bottom (VBottom) of their bounding boxes, which can be configured with the child elements' Vertical Alignment property.

The order of the elements is given by the order of the child nodes within the laid out group.



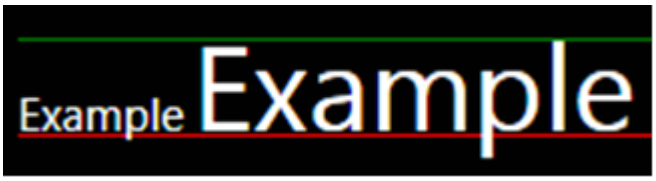
After selecting the Baseline Layouter, an additional layout property Base Line Offset becomes available. Using this property, the baseline can be set to a fixed position within the client area. Setting the baseline offset to a negative value (e.g. -1) configures the baseline for automatic, which means, the baseline is automatically set to a position where all children are positioned below the top of the client area. The Base Line Offset property is not only available to the Baseline Layouter, but also its child objects. The property can be found under the layout section in the properties panel.



WPF Implementation

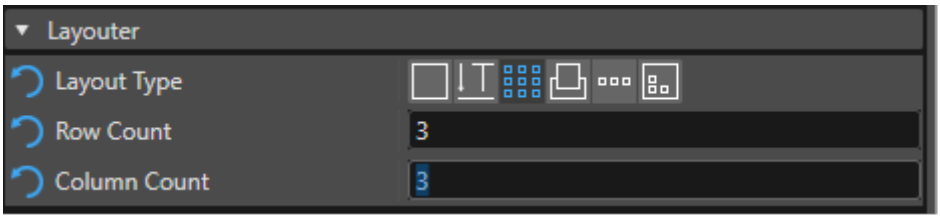
There is no implementation similar to a Baseline Layouter in WPF. However, texts with different sizes can be aligned to a common baseline using nested TextBlocks.

```
<Rectangle Height="1" Width="650" Fill="green" VerticalAlignment="Top" Margin="0,12,0,0"/>
<Rectangle Height="1" Width="650" Fill="red" VerticalAlignment="Top" Margin="0,44,0,0"/>
<StackPanel Orientation="Horizontal">
    <TextBlock Foreground="white">
        <TextBlock FontSize="15">Example </TextBlock>
        <TextBlock FontSize="40">Example</TextBlock>
    </TextBlock>
</StackPanel>
```

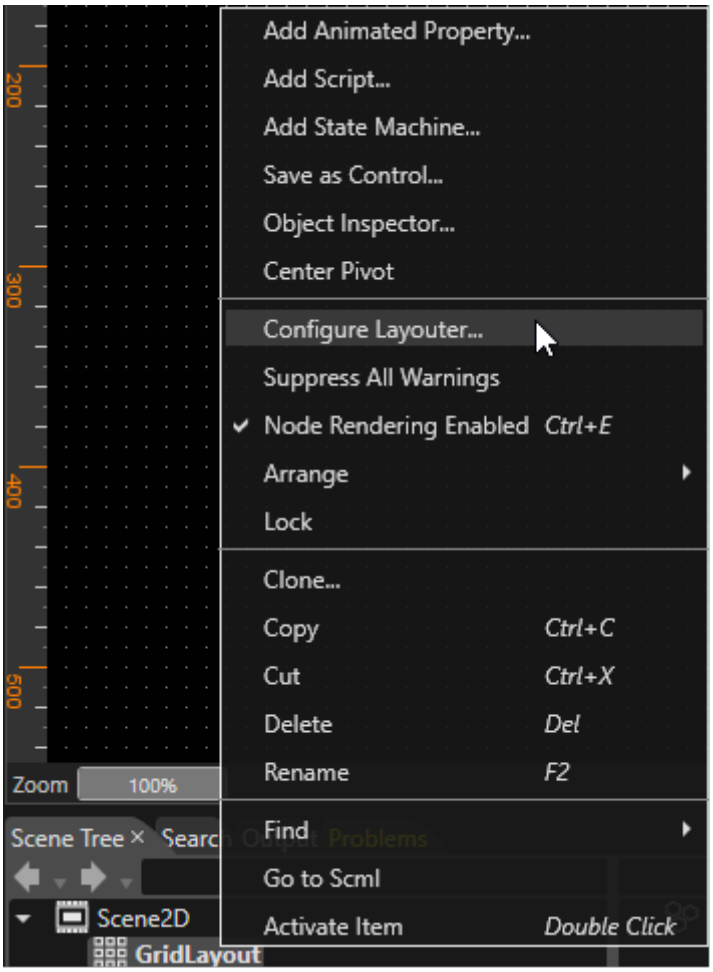


Grid Layouter

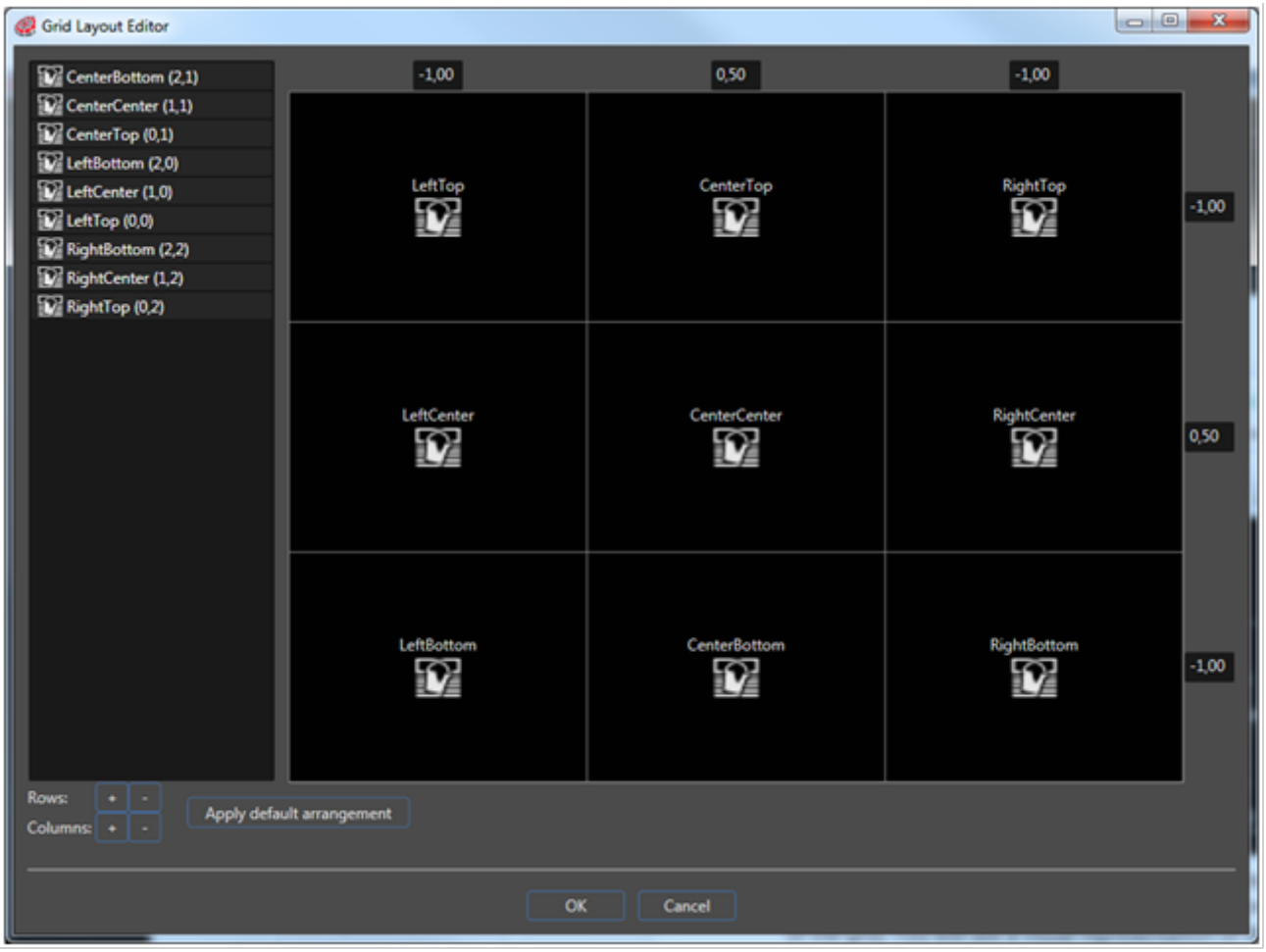
This layouter allows you to define a grid, assign nodes to the grid cells and layout these nodes to fit into the cells. If you select the grid layouter for a node, the properties for Row Count and Column Count can be configured in the properties panel.



Additionally, the context menu for the grid layout node (right-click on the specific grid layout) is extended by the entry Configure Layouter.



After choosing this context menu entry the Grid Layout Editor will open.



On the left side there is a list of nodes that can be assigned to the cells. Each cell shall contain only one node. At the bottom you can increase/decrease the number of rows/columns of the grid. You will see a visual representation of the grid and the assigned nodes in the center. The width of the columns and height of the rows can be configured at the top and on the right side. The value -1 means that it is flexible and the remaining area will be distribute by the grid layouter. Values larger than or equal to 1.0 are absolute values. The grid layouter will use exactly this size for the column/row and reduce the remaining area by this value. Values below 1.0 are relative values. In the sample above, the center row will receive 50% of the remaining height (after all absolute values have been considered) and the center column will receive 50% of the remaining height.

> Relative values will be normalized. This means, since a configuration of e.g. 0.1, 0.3 and 0.1 doesn’t sum up to 1.0, the values will be recalculated. The sum of all relative values is 0.5, so the relative value of 0.1 will be normalized with 0.1/0.5, which results in 0.2. So this part of the grid will get 20% of the available space.

A child node can be configured to span over multiple cells using its layout properties Row Span or Column Span. Here is an example on a Grid Layout configuration in CGI Studio with various cells using the column span and row span properties:

Similar to WPF Grid Layout

A similar GridLayout can be configured using a WPF GridLayout:

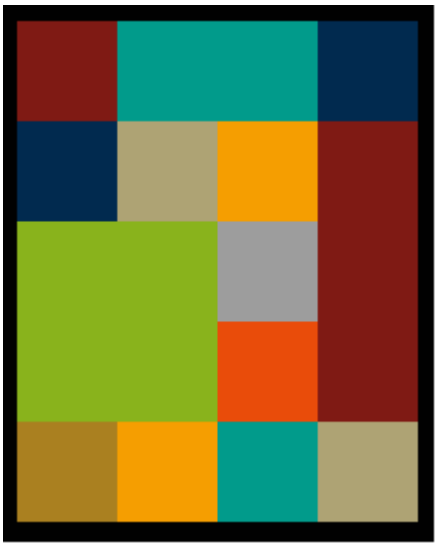
```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Rectangle Height="50" Width="50"  Fill="Red"   Grid.Row="0" Grid.Column="0"/>
    <Rectangle Height="50" Width="100" Fill
="Green" Grid.Row="0" Grid.Column="1" Grid.ColumnSpan="2" />
    <Rectangle Height="50" Width="50"  Fill="Blue"  Grid.Row="0" Grid.Column="3" />

    <Rectangle Height="50" Width="50"  Fill="Aqua"   Grid.Row="1" Grid.Column="0" />
    <Rectangle Height="50" Width="50"  Fill="Yellow" Grid.Row="1" Grid.Column="1" />
    <Rectangle Height="50" Width="50"  Fill="Coral"  Grid.Row="1" Grid.Column="2" />
    <Rectangle Height="150" Width="50" Fill
="Purple" Grid.Row="1" Grid.Column="3" Grid.RowSpan="3"/>

    <Rectangle Height="100" Width="100" Fill
="LimeGreen"  Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2" Grid.RowSpan="2"/>
    <Rectangle Height="50"  Width="50"  Fill="Chartreuse" Grid.Row="2" Grid.Column="2" />

    <Rectangle Height="50"  Width="50"  Fill="Crimson"  Grid.Row="3" Grid.Column="2" />
```

```
    <Rectangle Height="50" Width="50"  Fill="RosyBrown"   Grid.Row="4" Grid.Column="0"/>
    <Rectangle Height="50" Width="50"  Fill="GreenYellow" Grid.Row="4" Grid.Column="1" />
    <Rectangle Height="50" Width="50"  Fill="DarkCyan"    Grid.Row="4" Grid.Column="2" />
    <Rectangle Height="50" Width="50"  Fill="Salmon"      Grid.Row="4" Grid.Column="3" />
</Grid>
```



Overlay Layouter

This is the simplest layouter. It delegates the same layout area that it has received from its parent layouter to its child layouters and returns the maximum size back to its parent layouter. The Overlay Layouter can be used to layout the same area for a foreground and a background part.

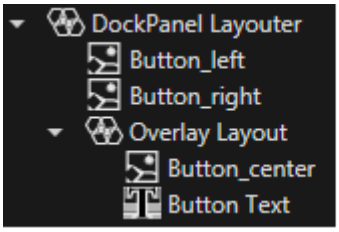
WPF Implementation

There is no separate layout panel available in WPF like the CGI Studio Overlay Layouter. However, the same layout can be achieved using a grid without column and row information.

```
<Grid>
    <Image Source="..\Images\button.png"/>
    <Label FontSize="18" FontStyle="Italic" VerticalAlignment="Center" HorizontalAlignment="Cent
       Simple Button
    </Label>
</Grid>
```

Use Cases

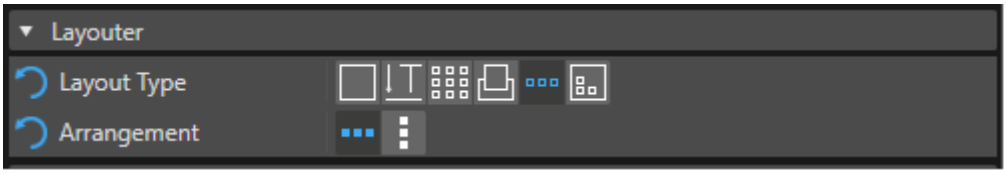
A button that can grow with the length of its text can be configured as follows with CGI Studio Layouters:



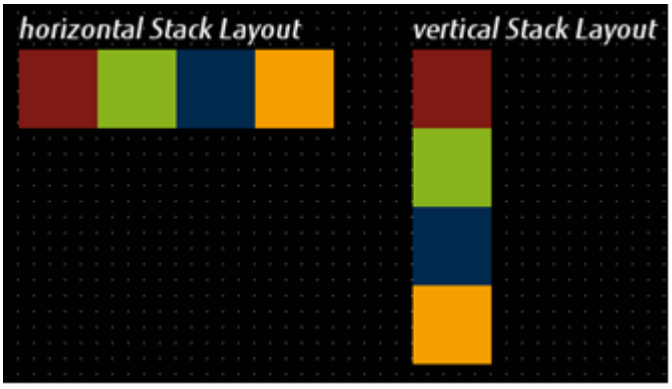
The DockPanel Layouter contains a node for the left of the button (docked left), a node for the right of the button (docked right) and an Overlay Layouter for the background of the button and the button's label. The Button_center node's stretch behavior is set to Fill, so that the image is resized automatically when the button's label is changed. The text node as foreground should be aligned to vertical and horizontal center.

Stack Layouter

The Stack Layouter stacks the nodes on top of each other under consideration of their preferred size. The additional Arrangement property will be shown in the properties panel as soon as you switch to a Stack Layouter. You can choose to configure horizontal or vertical stacking (horizontal is the default). There are no additional properties at child node level for this layouter.



Here is an example of a horizontal and a vertical Stack Layout using the CGI Studio Stack Layouter.

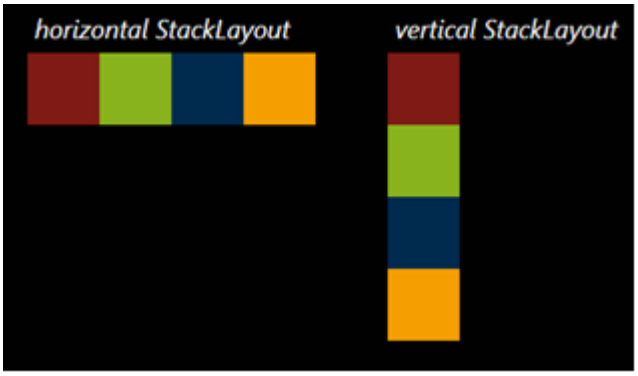


Similar to WPF StackPanel

The CGI Studio Stack Layouter is similar to the WPF StackPanel. WPF defined a vertical alignment as default. Here is how to configure a horizontal and a vertical Stack Layout like the one above using WPF:

```
<StackPanel Orientation="Horizontal">
    <Rectangle Fill="Red" Width="50" Height="50"/>
    <Rectangle Fill="Green" Width="50" Height="50"/>
    <Rectangle Fill="Blue" Width="50" Height="50"/>
    <Rectangle Fill="Gray" Width="50" Height="50"/>
</StackPanel>

<StackPanel>
    <Rectangle Fill="Red" Width="50" Height="50"/>
    <Rectangle Fill="Green" Width="50" Height="50"/>
    <Rectangle Fill="Blue" Width="50" Height="50"/>
    <Rectangle Fill="Gray" Width="50" Height="50"/>
</StackPanel>
```



Use Cases

A vertical stack layouter can be used to create any kind of lists. Horizontal stack layouters can be useful when placing ok and cancel buttons in dialog windows.

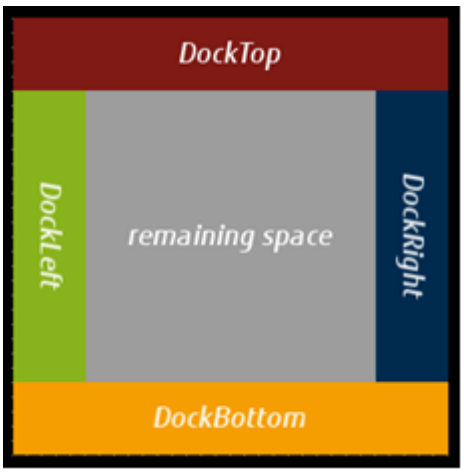
Dock Panel Layouter

The dock panel layouter is a very simple but powerful layouter. It provides an easy snapping or docking of elements to the top, bottom, left or right of the client area. One child is docked after the other, each leaving the remaining space for the rest of the elements, with regard to the order in which the child elements are added.

If a CGI-Studio Dock Panel layouter is set on a group node, an additional property Dock Side is available to all its child nodes. This property can be used to configure the side they should be docking to.

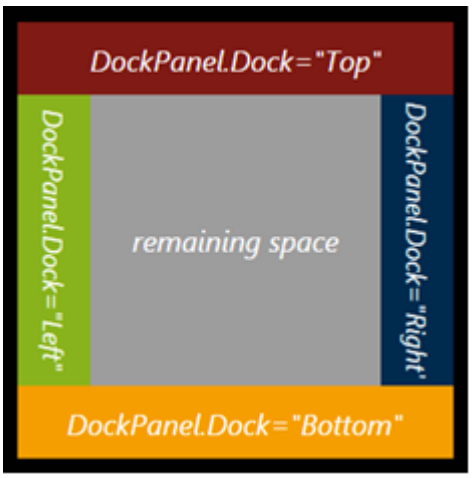
Elements can be aligned to the following four dock sides: DockLeft, DockTop, DockRight and DockBottom.

There is no center dock side because it is simply not needed and potentially confusing. The last child node takes over the complete remaining area, since there is no more node to be arranged.



WPF equivalent - DockPanel

Using WPF, a similar result can be achieved using the WPF DockPanel.




```
<DockPanel Height="300" Width="300">
    <Label DockPanel.Dock="Top" Height="50" Background="Red" Foreground="White" FontStyle="Itali
    <Label DockPanel.Dock="Bottom" Height="50" Background="DarkKhaki" Foreground="White" FontSty
    <TextBlock  DockPanel.Dock="Left" Width="50" Background="Green" Foreground="White" FontStyle
                    <LineBreak />Panel
                    <LineBreak />=
                    <LineBreak/>"Left"
```

```
    </TextBlock>
    <TextBlock DockPanel.Dock="Right" Width="50" Background="Blue" Foreground="White" FontStyle=
                    <LineBreak /> Panel
                    <LineBreak /> =
                    <LineBreak/> "Right"
    </TextBlock>
    <Label Background="Gray" Foreground="White" FontStyle="Italic" FontSize="20">remaining space
</DockPanel>
```

Use Cases

- A button that can grow with the length of its text can be configured using the Dock Laoyuter. A detailed description can be found in the chapter of the Overlay Layouter.
- If you want to create a center, make five groups, dock the first four in this order: top, bottom, left and right. The fifth group then builds the center. But this is only one of several possible center layouts (depending on the dock side order of the first four nodes).
- If you always use DockLeft you can achieve a horizontal stack like layout. If you always use DockTop you can achieve a vertical stack like layout. In difference to the stack layouter the last node receives the remaining area.

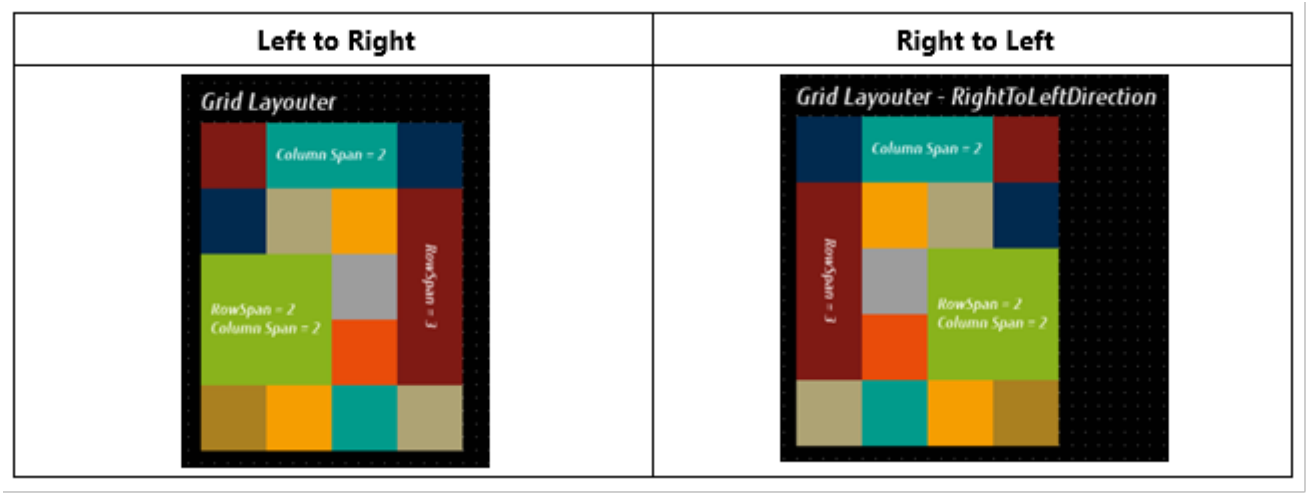
Culture Dependent Layout

When catering to end users globally, a user interface design benefits from accounting for both left-to-right (LTR) and right-to-left (RTL) writing systems. This chapter describes how the Layout Direction property works with different layouters. AutomaticDirection is currently the default value. You can change the interpretation of the AutomaticDirection by calling Layouter::SetAutomaticDirectionBehavior(). The default interpretation is to inherit the parent's value. If no value is set on any parent, the culture layout direction is used. The value CultureDirection directly takes on the culture Layout Direction. The value InheritDirection takes on the Layout Direction of its parent.

> Changing any child node's Layout Direction property from LeftToRightDirection to RightToLeftDirection additionally results in a reinterpretation of the Horizontal Alignment property - values HLeft and HRight take on each other's effects. The same is true for its children's margins.

Grid Layouter

If a grid layouter's Layout Direction property is set to RightToLeftDirection, the horizontal order in which its children are arranged will be reversed. If a child has its Column Span greater than 1, it will span across the cells on its left side instead of the right side.



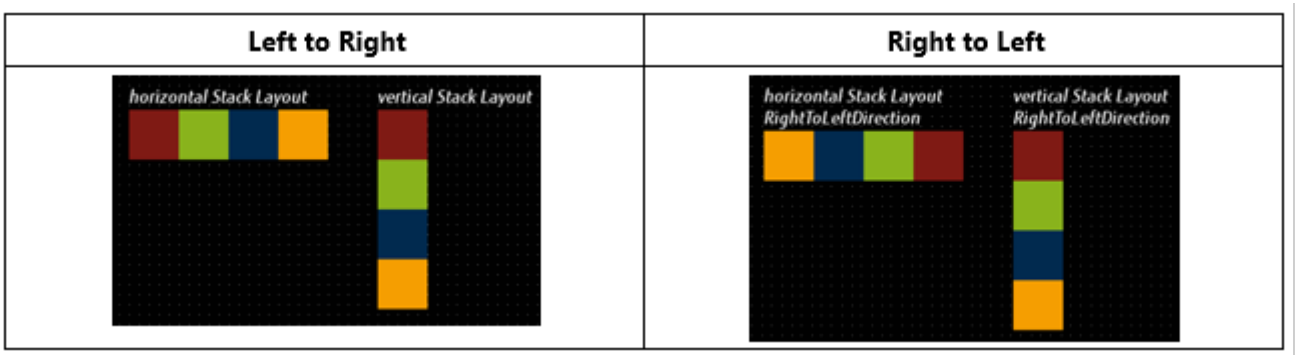
Overlay Layouter

Overlay layouter's Layout Direction property only has the general effects that all layouters share: its values HLeft and HRight take on each other's effects and it exchanges left and right margin sizes of its children.

In the example screenshot below an overlay layouter’s children are a text node; two arrow buttons and a bar; and a solid color node with Stretch Behavior set to fill. The left arrow button has its left margin set to 750 (pixels), the bar to 805 and the right arrow to 1010. These left margins becomes the right margins when overlay layouter’s Layout Direction is set to RightToLeftDirection. The third example puts the two arrow buttons and the bar as children of a group node with Layout Direction set to LeftToRightDirection to prevent the three nodes from changing the order. Only the group node’s margin changes.

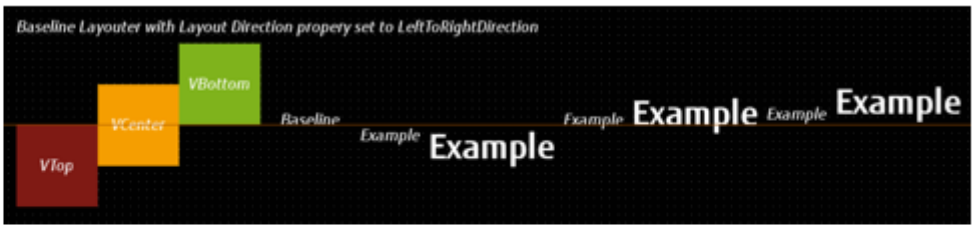
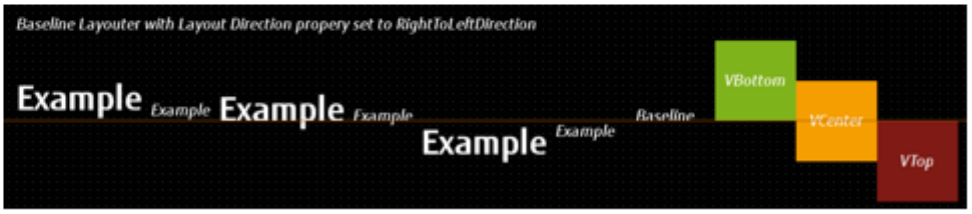
Stack Layouter

With the stack layouter’s Arrangement property set to Horizontal, the layout direction determines the order of its children. When the stack layouter’s Arrangement property is set to Vertical, the layout direction does not affect its children’s order.



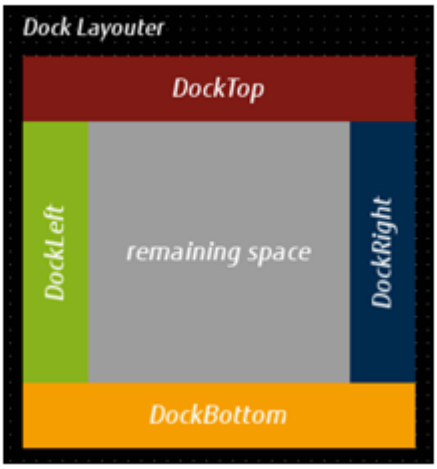
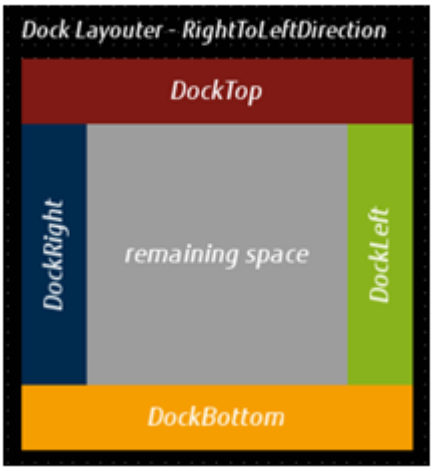
Baseline Layouter

Changing baseline layouter’s Layout Direction property from LeftToRightDirection to RightToLeftDirection reverses the direction in which baseline layouter’s children are ordered. The horizontal alignment is similar to the StackLayouter.

Left to Right	
Right to Left	

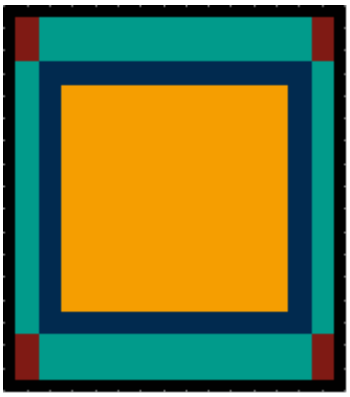
DockPanel Layouter

When changing the layout direction of the DockPanel layouter to RightToLeftDirection, the left and right docks will be exchanged.

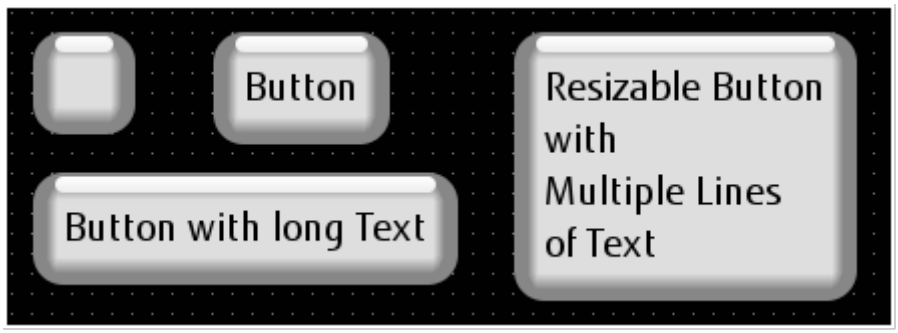
Left to Right	Right to Left
	

Automatic Resizing Layouter - CGI Studio vs. WPF

In order to adapt to changing window dimensions or the content’s size, the layouter is often required to resize automatically. The goal in this use case was to realize a sort of dialog or a button that looks as follows.



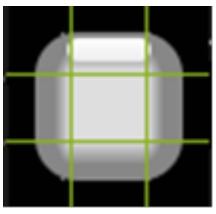
Using the layout above, a resizable button can be composed.



This button consists of 9 pictures, one for each of the following fields:

Top left	Top	Top right
Left	Center	Right
Bottom left	Bottom	Bottom right

The nodes of the fields marked in orange have to be configured to stretch and the Stretch Behavior has to be set to *Fill*.



Here are three possibilities on how to auto size content with different layouters and a direct comparison between a configuration using CGI Studio and a configuration using WPF.

Grid as Overlay

This solution uses a Grid as Overlay.

Solution using WPF

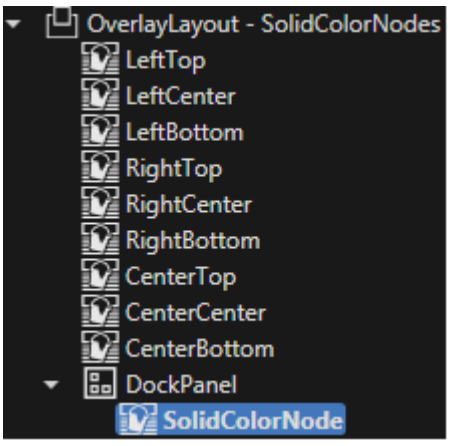
The colored rectangles are positioned in a grid, each at the same position, alignment and margins make them all visible next to each other.

```
<Canvas>
    <Grid>
        <Rectangle HorizontalAlignment="Left" VerticalAlignment="Top" MinWidth="10" MinHeight="2
        <Rectangle HorizontalAlignment="Left" VerticalAlignment="Stretch" Margin="0,20,0,20" Mir
        <Rectangle HorizontalAlignment="Left" VerticalAlignment="Bottom" MinWidth="10" MinHeight
        <Rectangle HorizontalAlignment="Right" VerticalAlignment="Top" MinWidth="10" MinHeight="
        <Rectangle HorizontalAlignment="Right" VerticalAlignment="Stretch" Margin="0,20,0,20" Mi
        <Rectangle HorizontalAlignment="Right" VerticalAlignment="Bottom" MinWidth="10" MinHeigh
        <Rectangle HorizontalAlignment="Stretch" VerticalAlignment="Top" Margin="10,0,10,0" MinW
        <Rectangle HorizontalAlignment="Stretch" VerticalAlignment="Stretch" Margin="10,20,10,20
        <Rectangle HorizontalAlignment="Stretch" VerticalAlignment="Bottom" Margin="10,0,10,0" M
        <Grid Margin="10,20,10,20" >
            <Rectangle Margin="10,10,10,10" Width="100" Height="100" Fill="Yellow" />
        </Grid>
    </Grid>
</Canvas>
```

The WPF Canvas is used to simulate CGI Studio’s Default Layouter environment with infinite size.

CGI Studio Solution

The configuration in CGI Studio is similar, but instead of the grid for the content, the more resource saving Overlay Layouter is used. The colored rectangles are positioned in the OverlayLayouter (OverlayAutoResize). Different horizontal and vertical alignments as well as margins make them visible next to each other.



Dock Panel / Dock Layouter

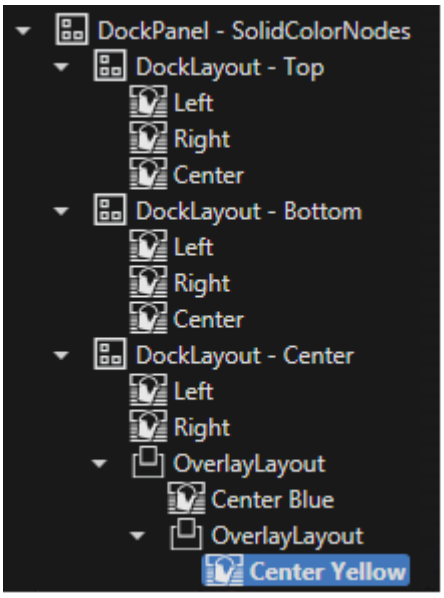
The same can be achieved using a Dock Panel/DockLayouter: A Dock Panel/DockLayouter for the top row is docked to the top and one for the bottom row is docked to the bottom. Another Dock Panel/DockLayouter is used for the middle row. In the center there is a grid with two overlapping rectangles, a blue one as background and a yellow rectangle with a margin of 10 to each direction, to create the blue border.

Solution with WPF

```
<Canvas>
    <DockPanel>
        <DockPanel DockPanel.Dock="Top">
            <Rectangle MinWidth="10" MinHeight="20" DockPanel.Dock="Left" Fill="Red" />
            <Rectangle MinWidth="10" MinHeight="20" DockPanel.Dock="Right" Fill="Red" />
            <Rectangle MinWidth="10" MinHeight="20" Fill="Green"/>
        </DockPanel>
        <DockPanel DockPanel.Dock="Bottom">
            <Rectangle MinWidth="10" MinHeight="20" DockPanel.Dock="Left" Fill="Red" />
            <Rectangle MinWidth="10" MinHeight="20" DockPanel.Dock="Right" Fill="Red" />
            <Rectangle MinWidth="10" MinHeight="20" Fill="Green"/>
        </DockPanel>
        <DockPanel>
            <Rectangle MinWidth="10" MinHeight="20" DockPanel.Dock="Left" Fill="Green" />
            <Rectangle MinWidth="10" MinHeight="20" DockPanel.Dock="Right" Fill="Green" />
            <Grid>
                <Rectangle MinWidth="10" MinHeight="20" Fill="Blue"/>
                <Grid>
                    <Rectangle Margin="10,10,10,10" Width="100" Height="100" Fill="Yellow" />
                </Grid>
            </Grid>
        </DockPanel>
    </DockPanel>
</Canvas>
```

CGI Studio Solution

In CGI Studio the solution is similar, but instead of configuring the content as grid, an overlay layouter is used in order to save resources.



Grid Layout

A Grid is defined for the colored rectangles. The left and right column as well as the top and bottom row are configured to be as wide and as high as needed by their child elements. The center row and the center column get the rest of the available space. The content of the centered cell is configured to stretch vertically and horizontally.

Solution using WPF

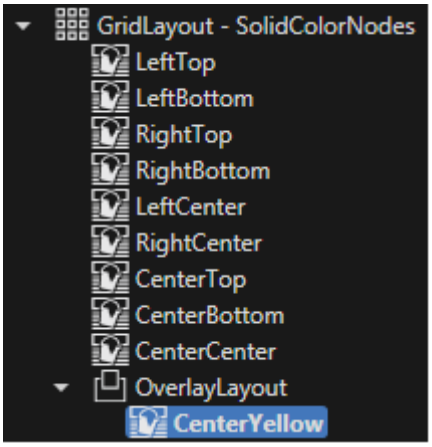
With a Grid the WPF configuration would look like this:

```
<Canvas>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="auto" />
        </Grid.ColumnDefinitions>
        <Rectangle MinWidth="10" MinHeight="20" Grid.Row="0" Grid.Column="0" Fill="Red" />
        <Rectangle MinWidth="10" MinHeight="20" Grid.Row="0" Grid.Column="2" Fill="Red" />
        <Rectangle MinWidth="10" MinHeight="20" Grid.Row="2" Grid.Column="0" Fill="Red" />
        <Rectangle MinWidth="10" MinHeight="20" Grid.Row="2" Grid.Column="2" Fill="Red" />
        <Rectangle MinWidth="10" MinHeight="20" Grid.Row="0" Grid.Column="1" Fill="Green" />
        <Rectangle MinWidth="10" MinHeight="20" Grid.Row="2" Grid.Column="1" Fill="Green" />
        <Rectangle MinWidth="10" MinHeight="20" Grid.Row="1" Grid.Column="0" Fill="Green" />
        <Rectangle MinWidth="10" MinHeight="20" Grid.Row="1" Grid.Column="2" Fill="Green" />
        <Rectangle MinWidth="10" MinHeight="20" Grid.Row="1" Grid.Column="1" Fill="Blue" />
        <Grid Grid.Row="1" Grid.Column="1" >
            <Rectangle Margin="10,10,10,10" HorizontalAlignment="Stretch" Width="100" MinHeight=
        </Grid>
```

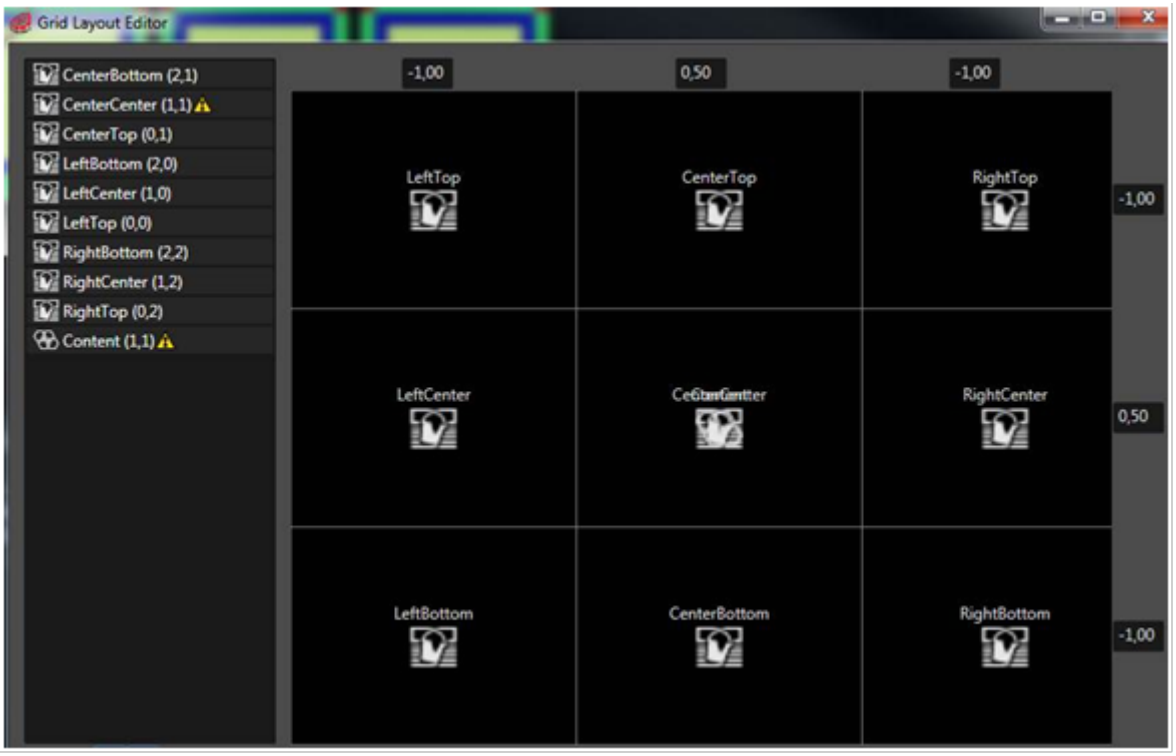
```
    </Grid>
</Canvas>
```

CGI Studio Solution

With CGI Studio, the same has been done, but instead of a Grid Layout for the Content, an Overlay Layouter has been used, that consumes less resources.



The configuration of the Grid Layouter is mapped like this:



Comparison

WPF row/column auto configuration is the same as the values -1 for the row height/column width configuration in CGI Studio. The WPF row/column fraction configuration (“*”) maps to any value greater than 0 and less than 1 for the row/column configuration in CGI Studio. In the sample 0.5 is configured for the center row height/column width. Note that this is not a percentage value, it is a fraction.

> Not used in this example, however noted for the sake of completeness: The WPF row/column fixed size configuration (e.g. “10”) maps to the same value for the row/column configuration in CGI Studio, where all values greater than 1 are absolute values in pixel.
> WPF has some bugs in the WPF grid when it comes to column/row span that contain fixed size rows/columns or fraction rows/columns (for the fraction it is even possible that the item does not fit into the span).

Conclusion

The solution using an overlay seems to be the approach that needs the minimum number of nodes. However, it requires the most content related configuration, the reason being that the size of the border primitives (WPF rectangle and CGI Studio RenderNode with SolidColorBrush) have to be reflected by the margins of some items. This applies for both WPF and CGI Studio. From the number of nodes point of view the grid approach comes next in the row. However, the grid layout uses the most resources of the considerable layouters: it is not a singleton and uses some internal data structures to perform the layout. The solution with the dock panel layouter uses the highest number of nodes. But, it adapts very flexible to the content, is a singleton and – like the overlay layout - requires only one pass to layout.