Layouter

- Overview
- Functional Overview
  - Basic Layouter Properties
  - Canvas Text - Size Property
  - Types of Layouters
  - Culture Dependent Layout
  - Automatic Resizing Layouter - CGI Studio vs. WPF
- Structural View
- Dynamic View
- Layout Monitor

Overview

General demands for Layouters

In a typical user interface the users need to distribute screen space to different elements. If the size of the elements is dynamic, it's desired that the screen space distribution is done automatically based on the content's natural size. To specify the behavior of this automatic layout, different layouters are used, each of which has its own layout strategy.

Adaption to the Window or the Display

Fixed pixel arrangement of controls doesn't work when the application needs to run on different targets. When creating a UI, the GUI elements (CGI-Studio nodes) are arranged by location and size to form a layout. A key requirement of any layout is to adapt to changes in window size and display settings. CGI-Studio layouters provide the possibilities to adapt a layout in these circumstances. Relative positioning is a keystone of the CGI-Studio layout system. It increases the ability to adapt to changing window and display conditions. Additionally, the layout system manages the communication between different elements to determine the layout: In a first step, a component tells its parent what size at what position it needs to have and in the second step, the parent tells the component what space it may have. In short, laying out is a recursive system that leads to an element being sized, positioned, and drawn.

Choosing a Layouter

The laying out of controls is very important and critical for an application's usability. It is used to arrange GUI elements in an application. The following important things have to be considered when selecting a layouter:

- Positions of the child elements
- Size of the child elements
- Layering of overlapping child elements on top of each other

Layouter Types

The CGI-Studio layout system provides several layouters for various use cases:

The **Default Layouter** is used to position elements explicitly using absolute coordinates. When no layouter is selected, the default layouter is used.

With the **Grid Layouter** child elements can be positioned in a tabular structure of rows and columns.

The **Overlay Layouter** can be used to arrange elements one over another to end up having e.g. a foreground element and a background element.

The **Stack Layouter** simply stacks the elements together under consideration of their preferred size and its configured horizontal or vertical alignment.

The **BaseLine Layouter** works similar to the Horizontal Stack Layouter, but the child elements are aligned to a line instead of aligning them to the client area.

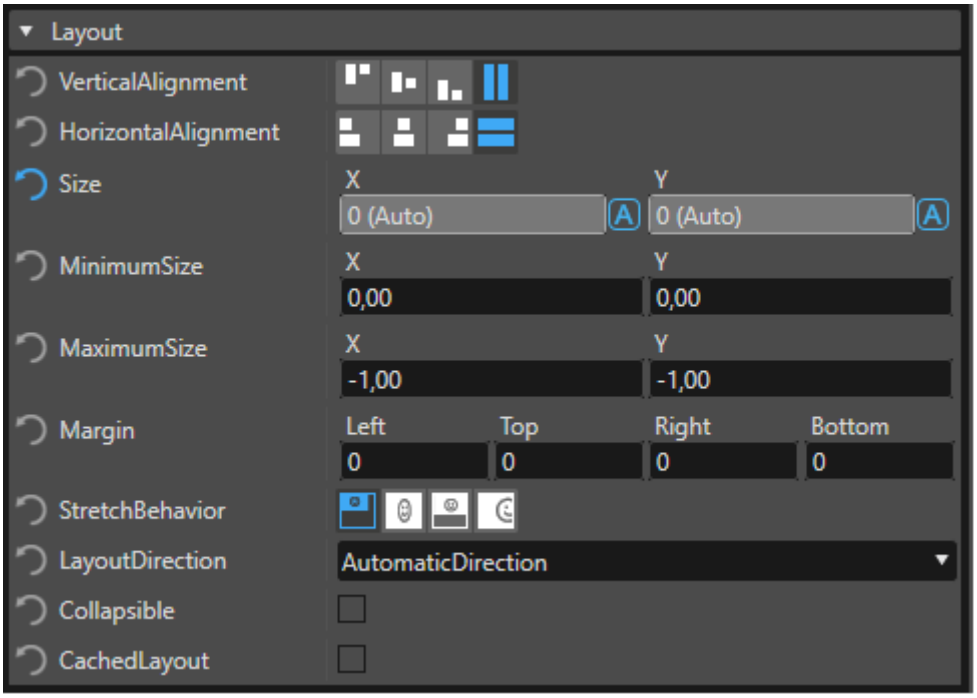
The **DockPanel Layouter** allows you to snap, or dock, elements to the edges (top, bottom, left or right) of the rectangular client area. The docking edge can be defined for each child element. One child element is docked after the other, each leaving the remaining space for the rest of the elements, considering the order in which the child elements are added.

Functional Overview

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.

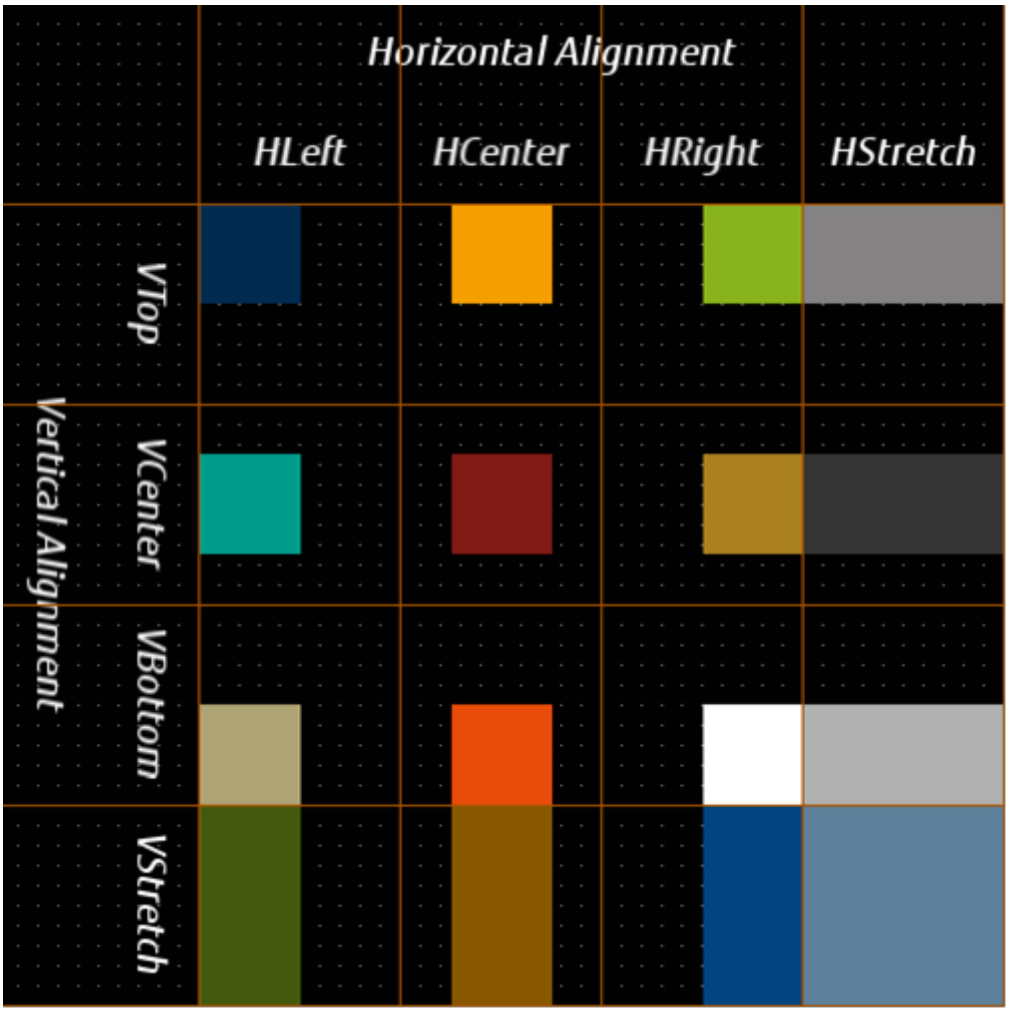


Vertical Alignment and Horizontal Alignment

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).

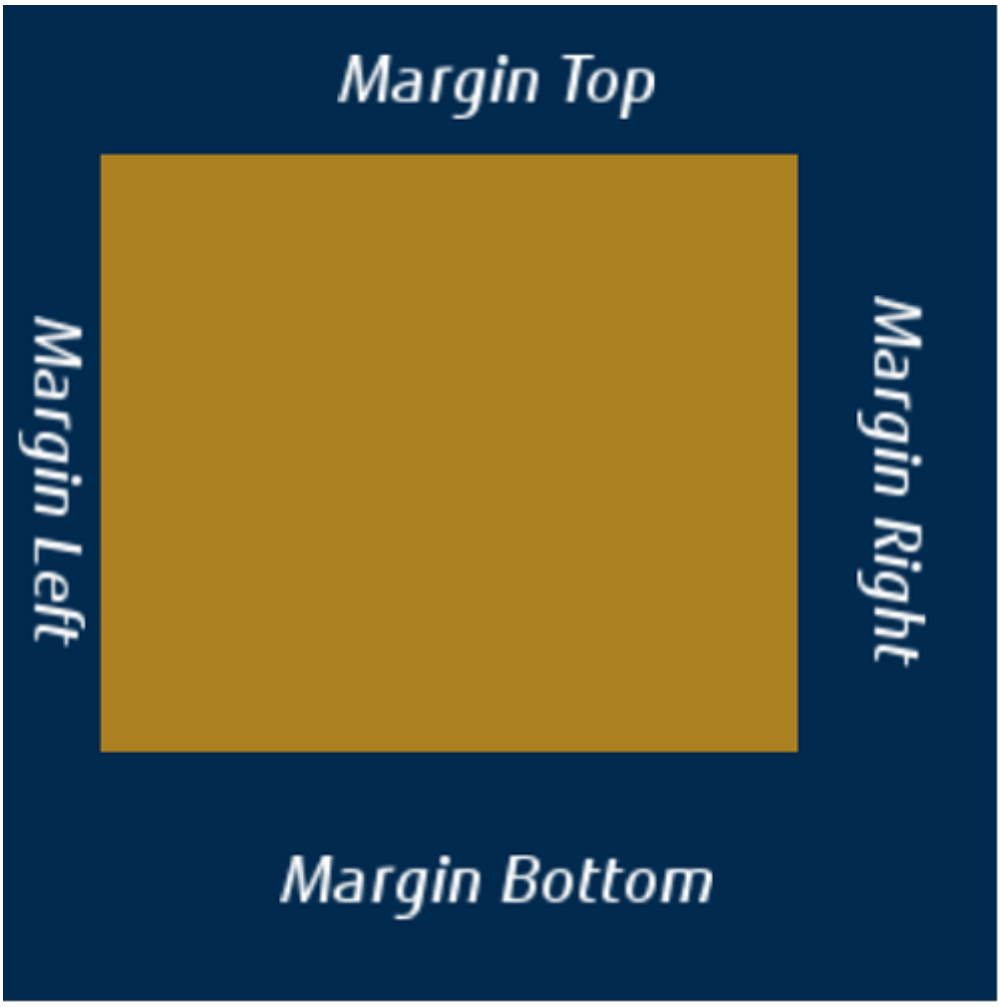
Similar to the Size property -1 can be used to indicate that the value is not set.

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).



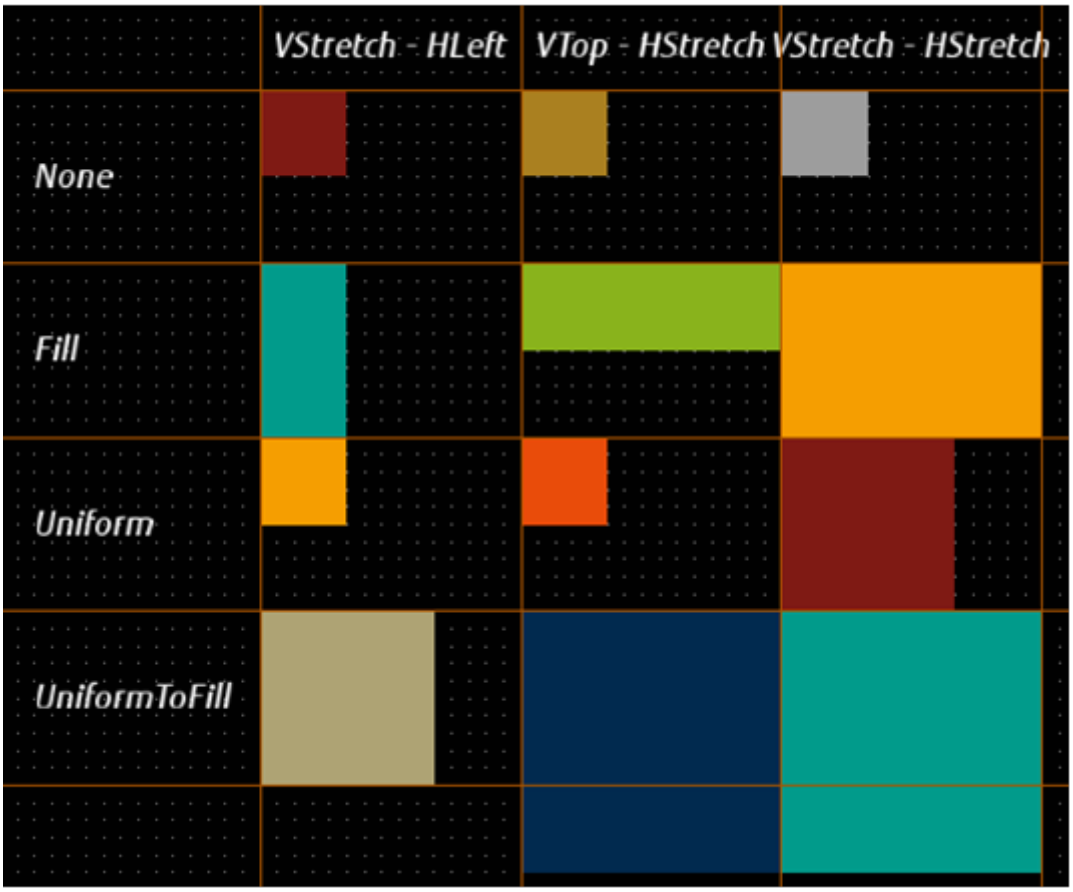
Stretch Behavior

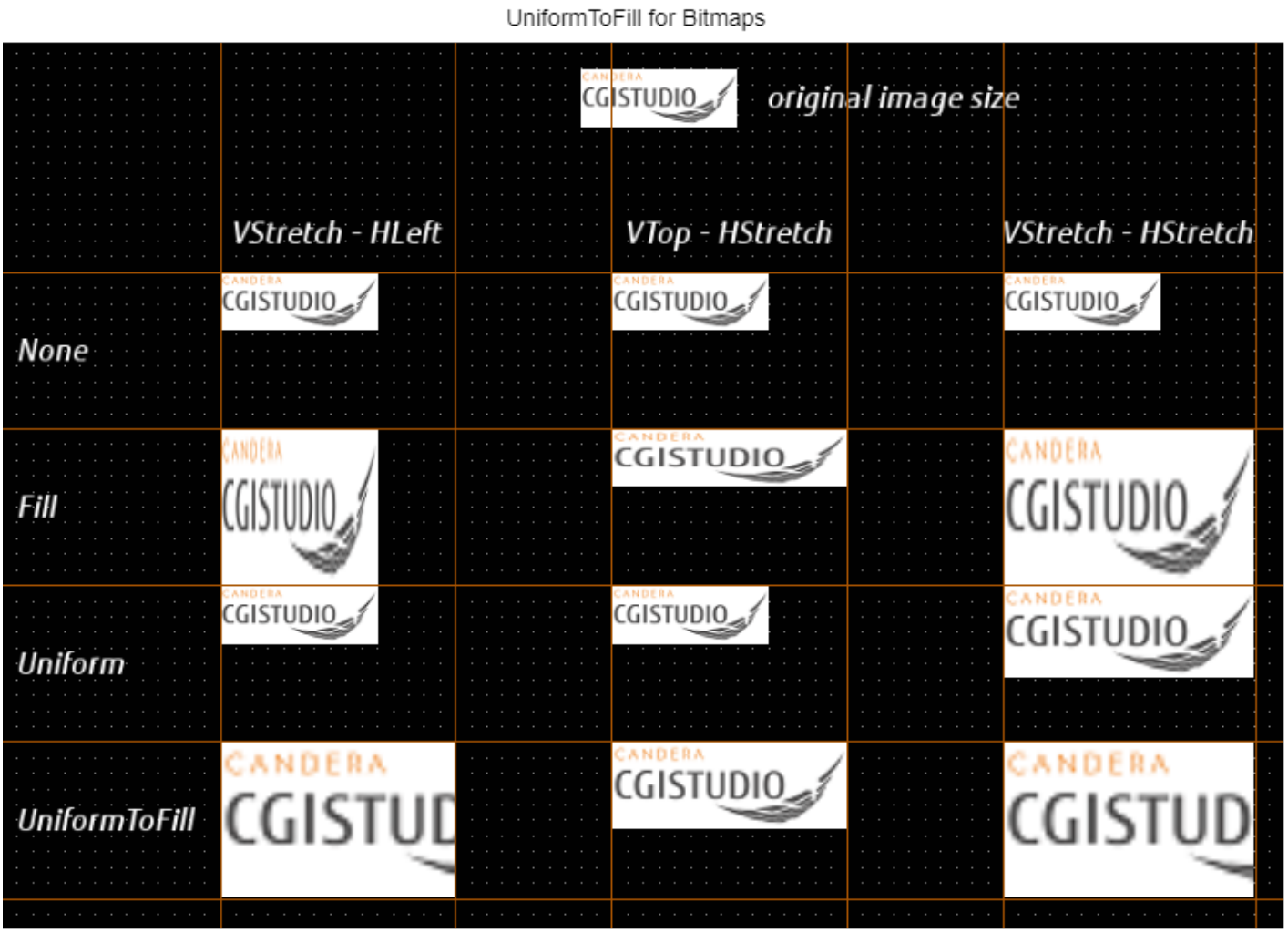
The Stretch Behavior property is only useful on RenderNodes because it defines how they have to be adjusted to the nodes' Scale property to fit into the layout area. Use None if the node has to keep its normal size. Use Fill if you want to stretch it in the axis which is configured for stretching. Use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

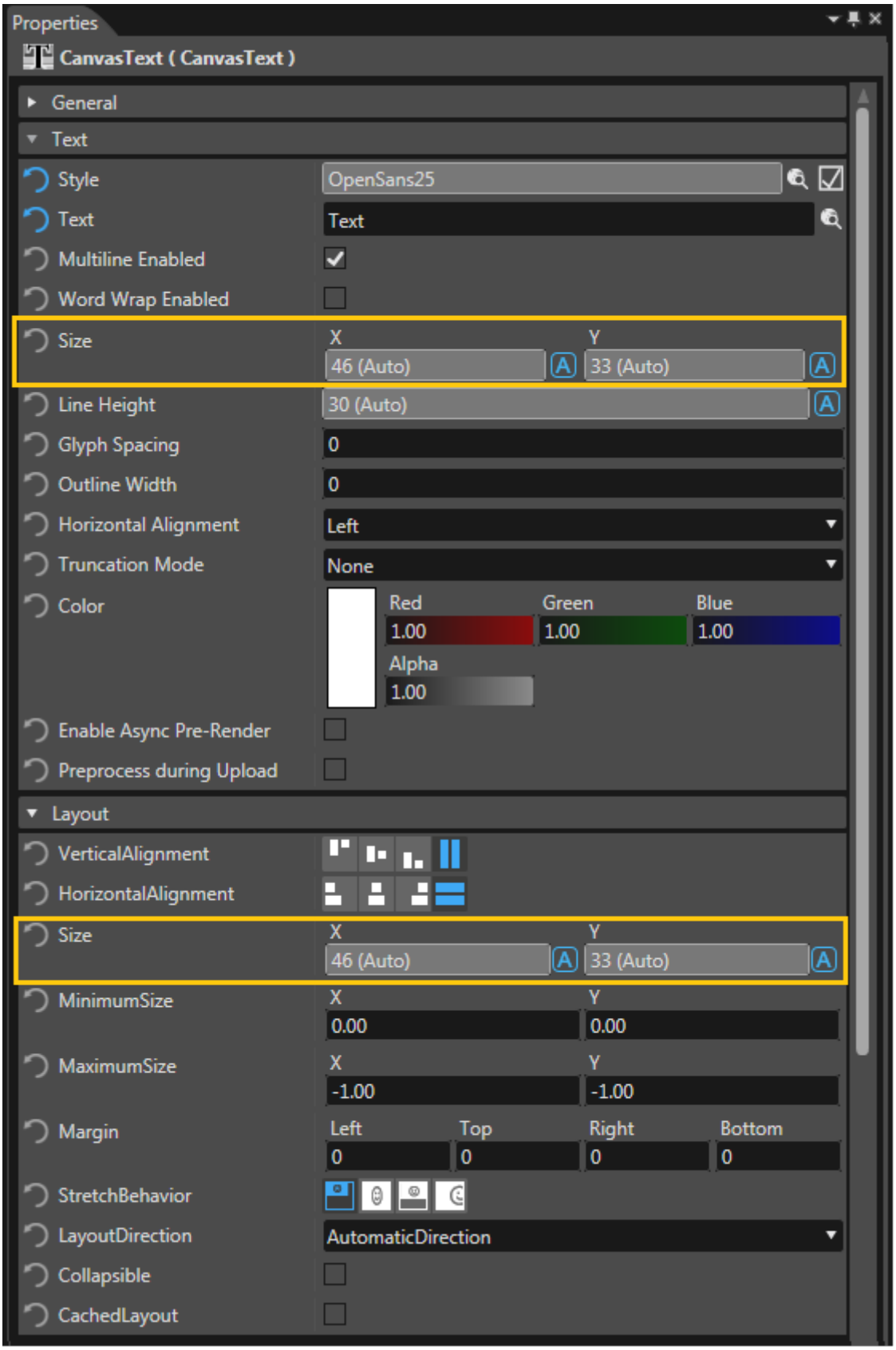
Collapsible

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:

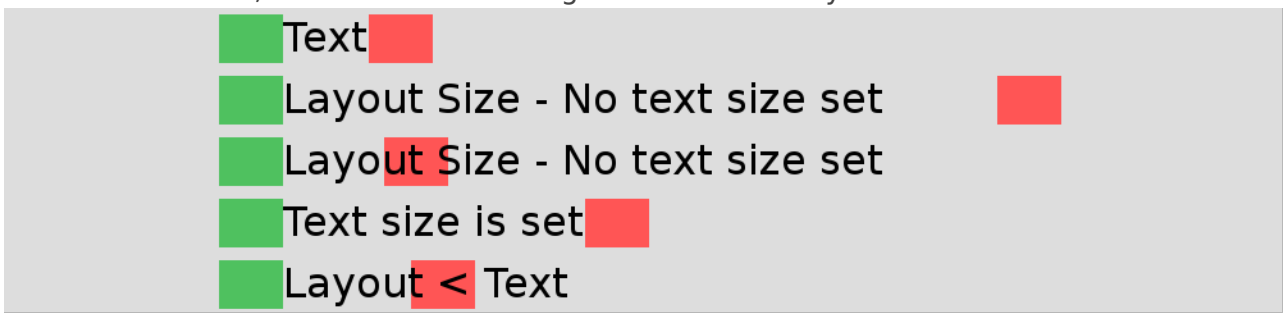
Alignment: The text has a property for horizontal alignment. This property aligns the text within Text::Size. If no Text::Size is set, the Layout::Size is used. The layout has a horizontal and vertical layout alignment property which is used to align the 'block' of the measured text within the Layout::Size. 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).

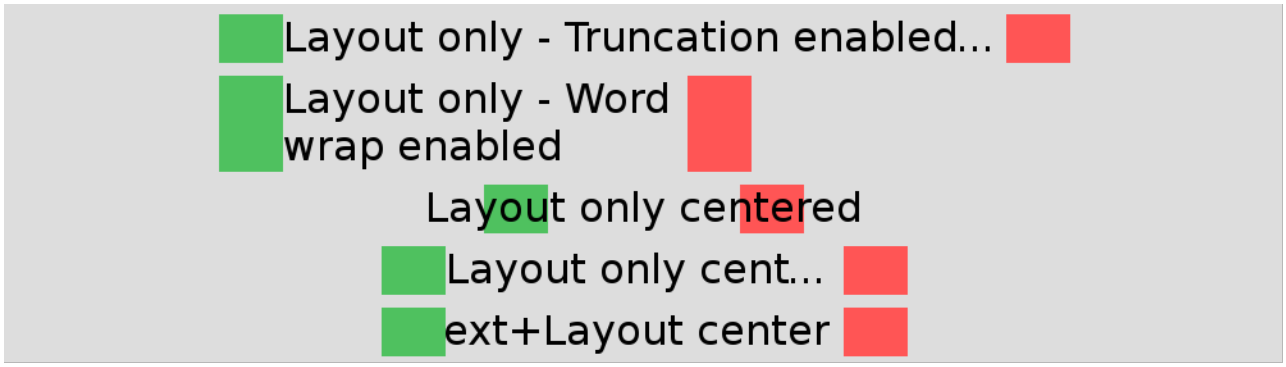
Examples

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.



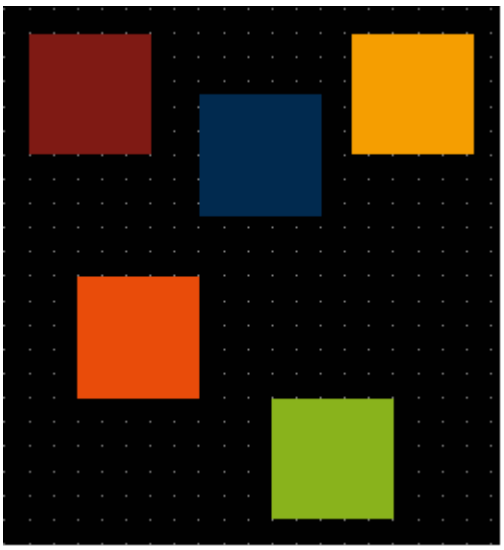
6. The layout size is set and Text::Size is not. The text would normally write over other elements. However, truncation is enabled.
7. Same as case as 6. but word wrap is enabled.
8. The Layout::Size is set and the layout alignment is center. It overlaps other elements but the text keeps being centered.
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.



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>
```

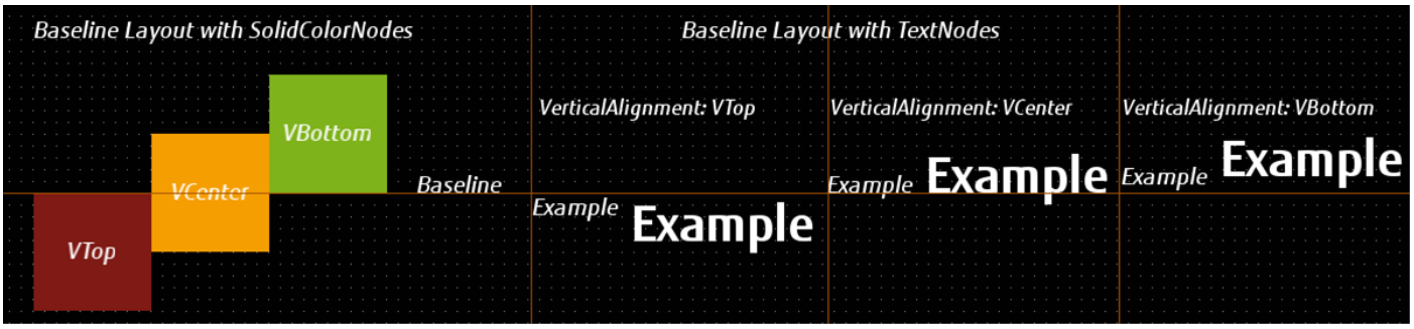
```
</Canvas>
```



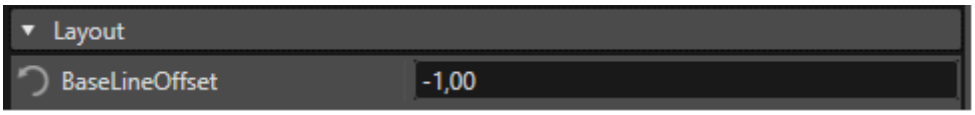
Baseline Layouter

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.



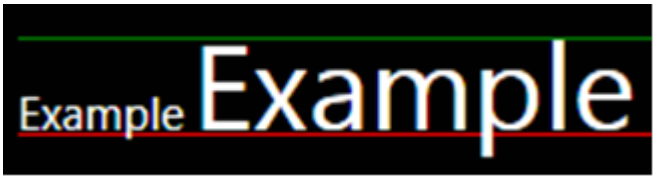
After selecting the Baseline Layouter, an additional layout property Base Line Offset becomes available. Using this property, the baseline can be set to a fixed position within the client area. Setting the baseline offset to a negative value (e.g. -1) configures the baseline for automatic, which means, 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

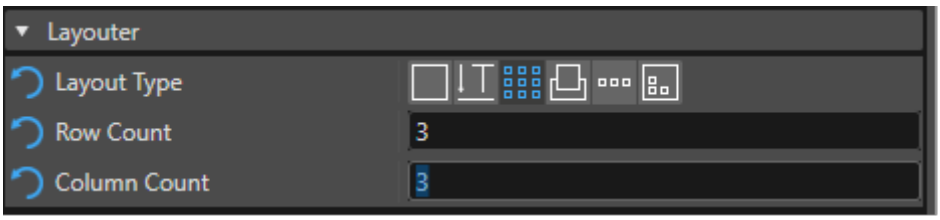
There is no implementation similar to a Baseline Layouter in WPF. However, texts with different sizes can be aligned to a common baseline using nested TextBlocks.

```
<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>
```

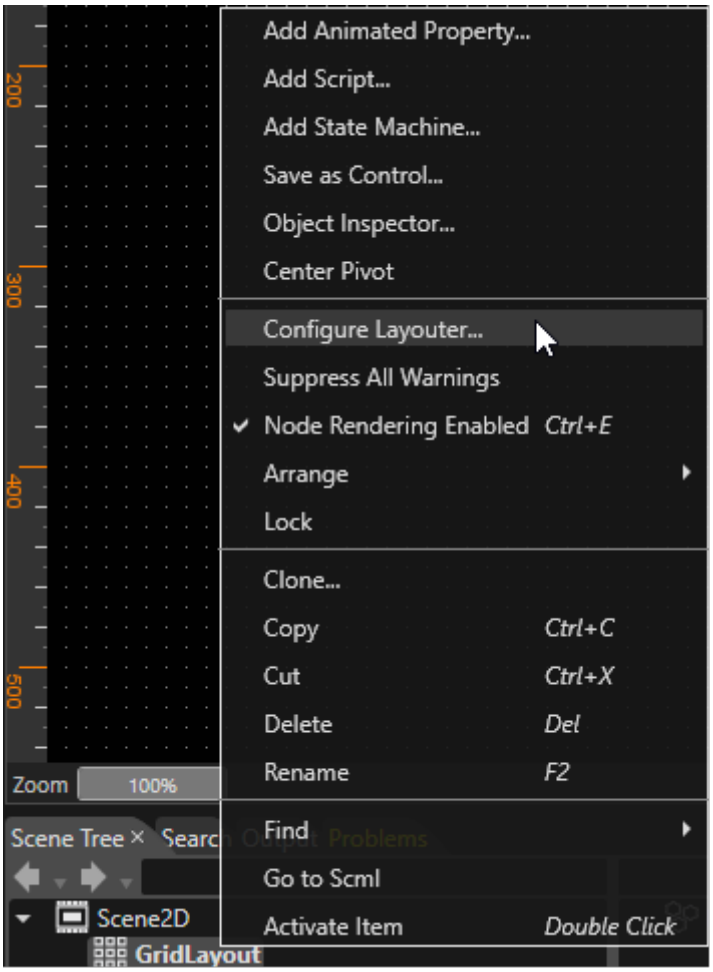


Grid Layouter

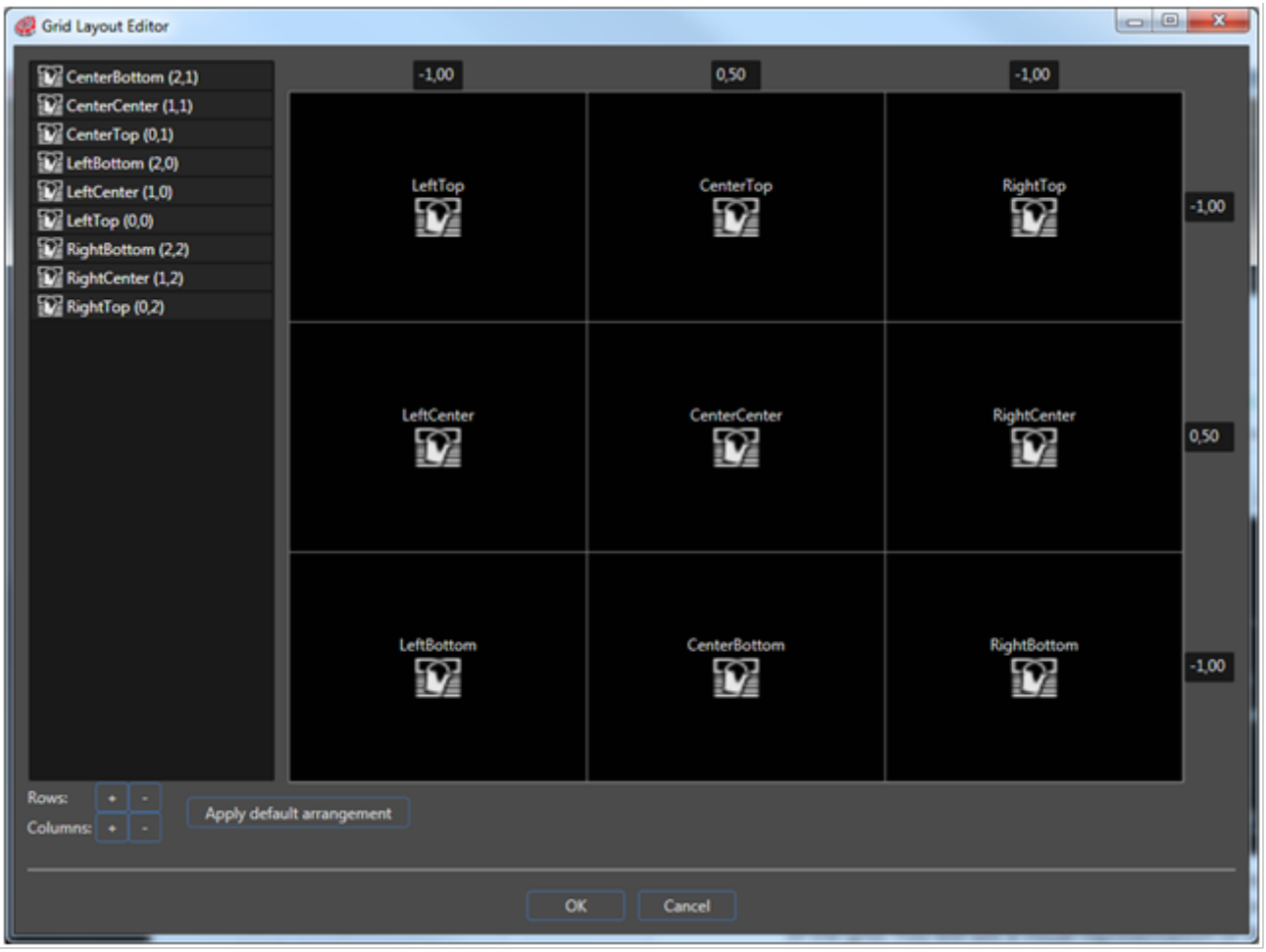
This layouter allows you to define a grid, assign nodes to the grid cells and layout these nodes to fit into the cells. If you select the grid layouter for a node, the properties for Row Count and Column Count can be configured in the properties panel.



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" />

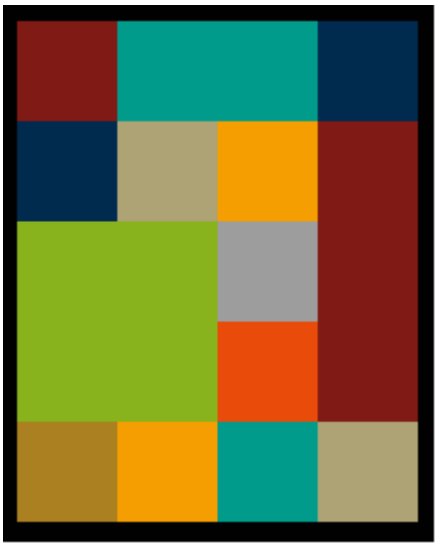
    <Rectangle Height="50" Width="50"  Fill="Aqua"   Grid.Row="1" Grid.Column="0" />
    <Rectangle Height="50" Width="50"  Fill="Yellow" Grid.Row="1" Grid.Column="1" />
    <Rectangle Height="50" Width="50"  Fill="Coral"  Grid.Row="1" Grid.Column="2" />
    <Rectangle Height="150" Width="50" Fill
="Purple"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

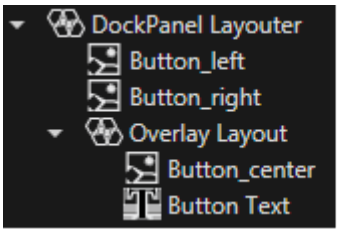
There is no separate layout panel available in WPF like the CGI Studio Overlay Layouter. However, the same layout can be achieved using a grid without column and row information.

```
<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:

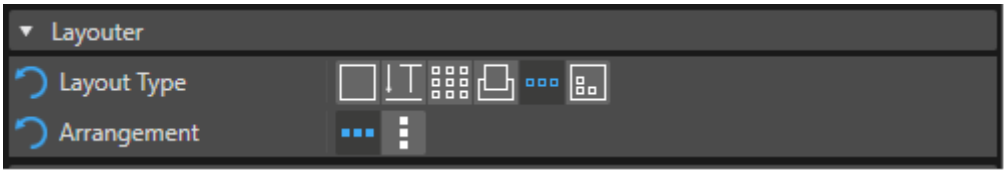




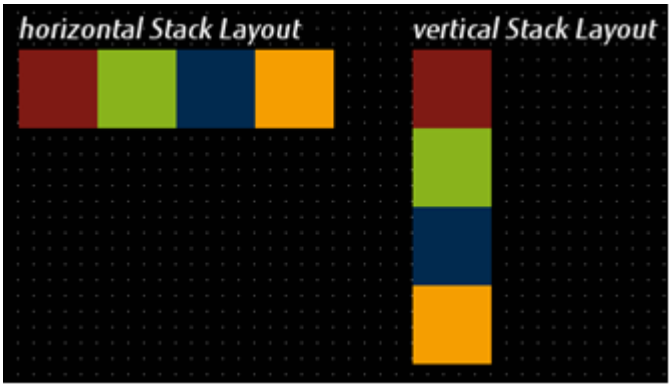
The DockPanel Layouter contains a node for the left of the button (docked left), a node for the right of the button (docked right) and an Overlay Layouter for the background of the button and the button's label. The Button_center node's stretch behavior is set to Fill, so that the image is resized automatically 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.

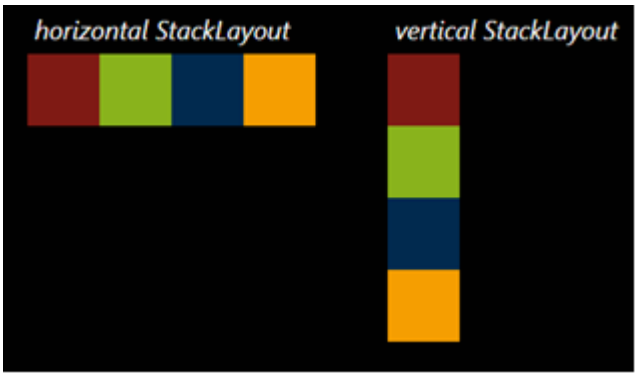


Similar to WPF StackPanel

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.

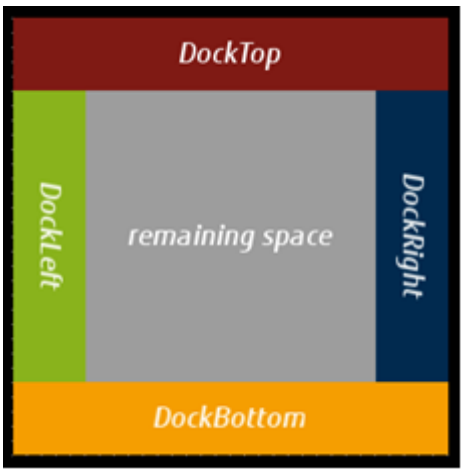
If a CGI-Studio Dock Panel layouter is set on a group node, an additional property Dock Side is available to all its child nodes. This property can be used to configure the side they should be docking to.



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

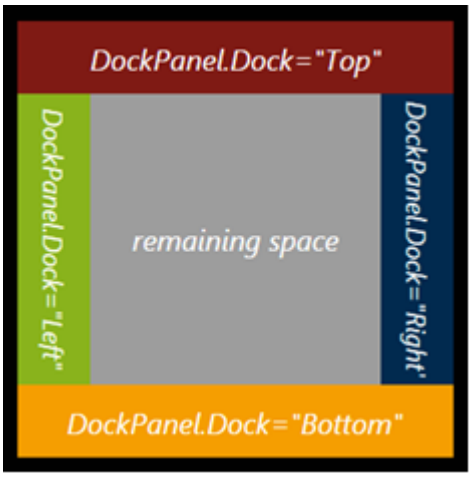


There is no center dock side because it is simply not needed and potentially confusing. The last child node takes over the complete remaining area, since there is no more node to be arranged.



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.

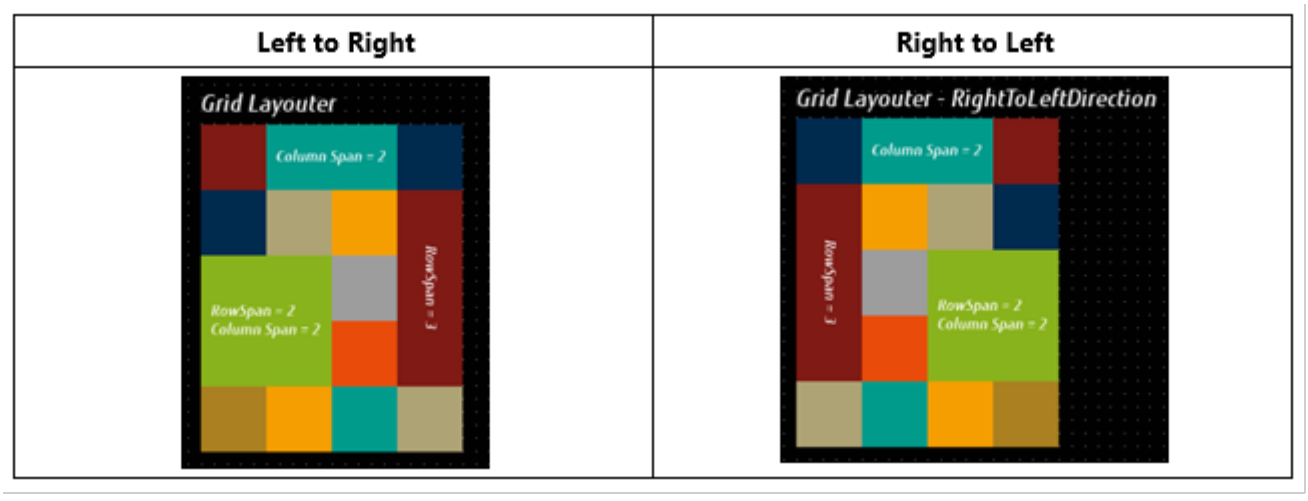
Culture Dependent Layout

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

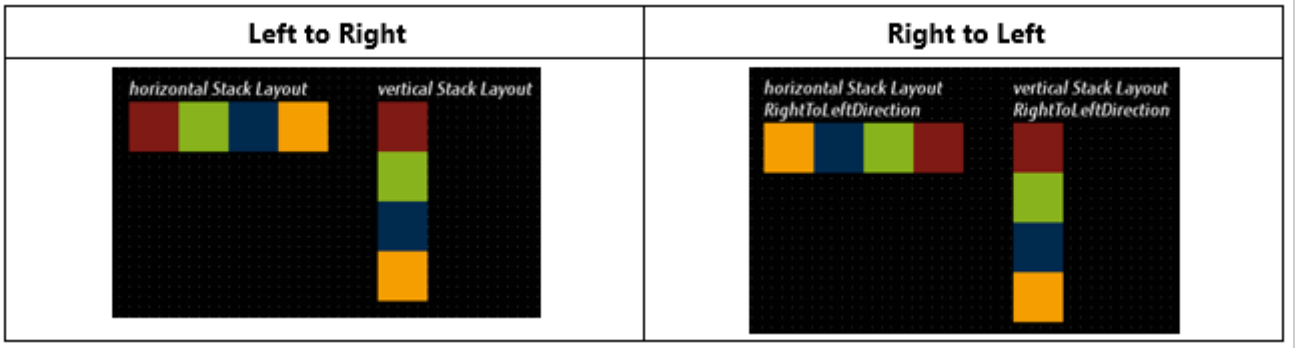
sizes of its children.

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.



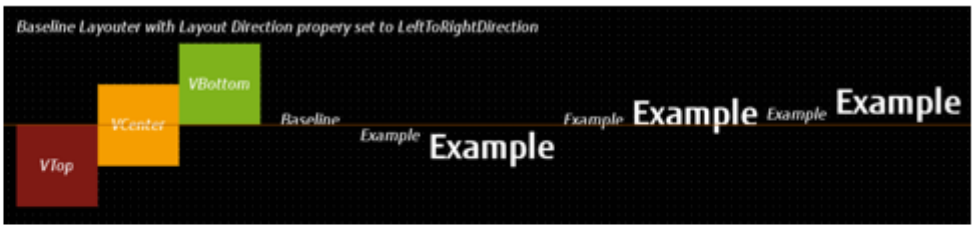
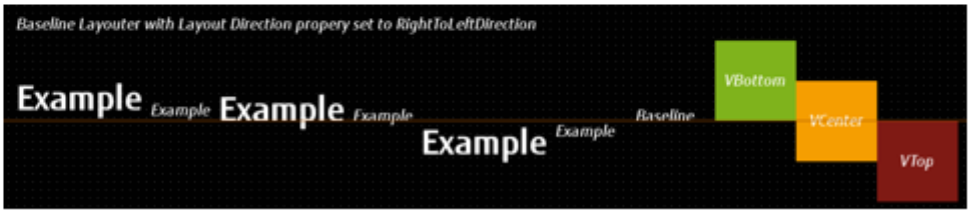
Stack Layouter

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.



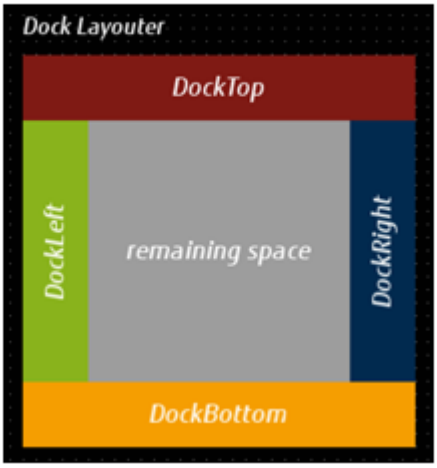
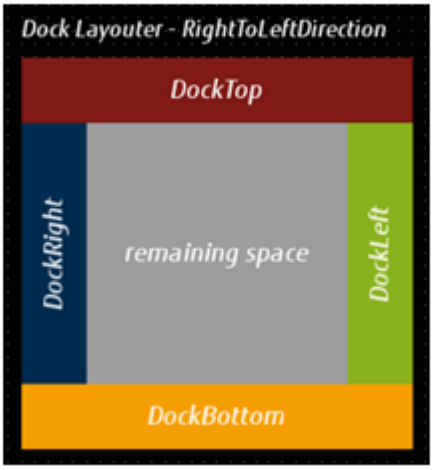
Baseline Layouter

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	

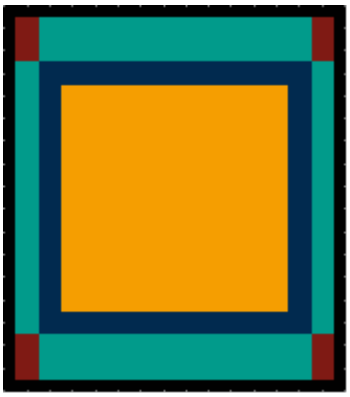
DockPanel Layouter

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

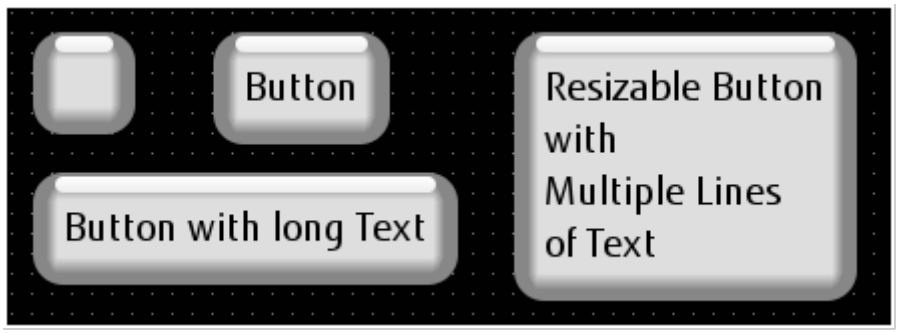
Left to Right	Right to Left
	

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.



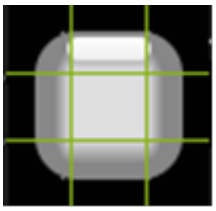
Using the layout above, a resizable button can be composed.



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

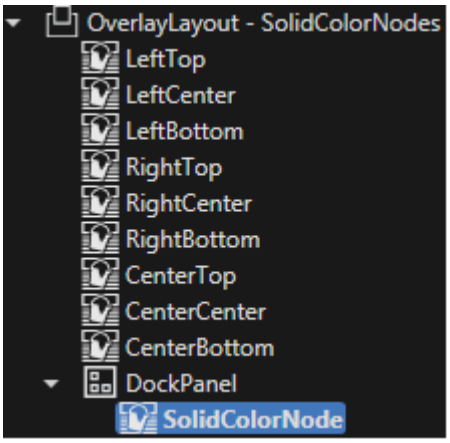
The colored rectangles are positioned in a grid, each at the same position, alignment and margins make them all visible next to each other.

```
<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

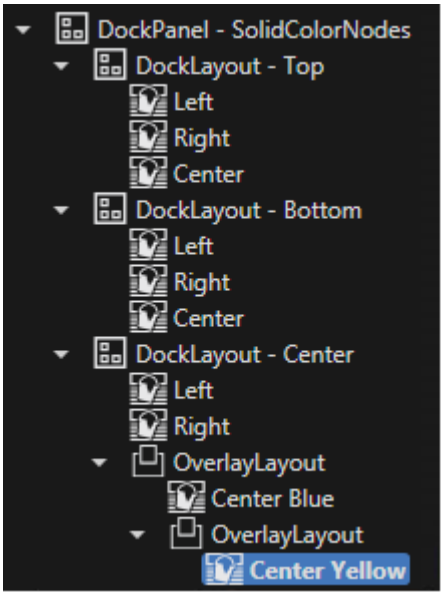
The same can be achieved using a Dock Panel/DockLayouter: A Dock Panel/DockLayouter for the top row is docked to the top and one for the bottom row is docked to the bottom. Another Dock Panel/DockLayouter is used for the middle row. In the center there is a grid with two overlapping rectangles, a blue 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

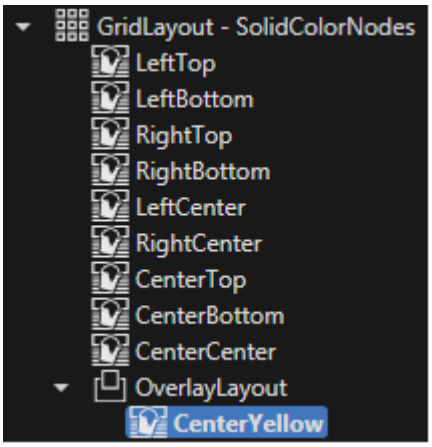
With a Grid the WPF configuration would look like this:

```
<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" />
```

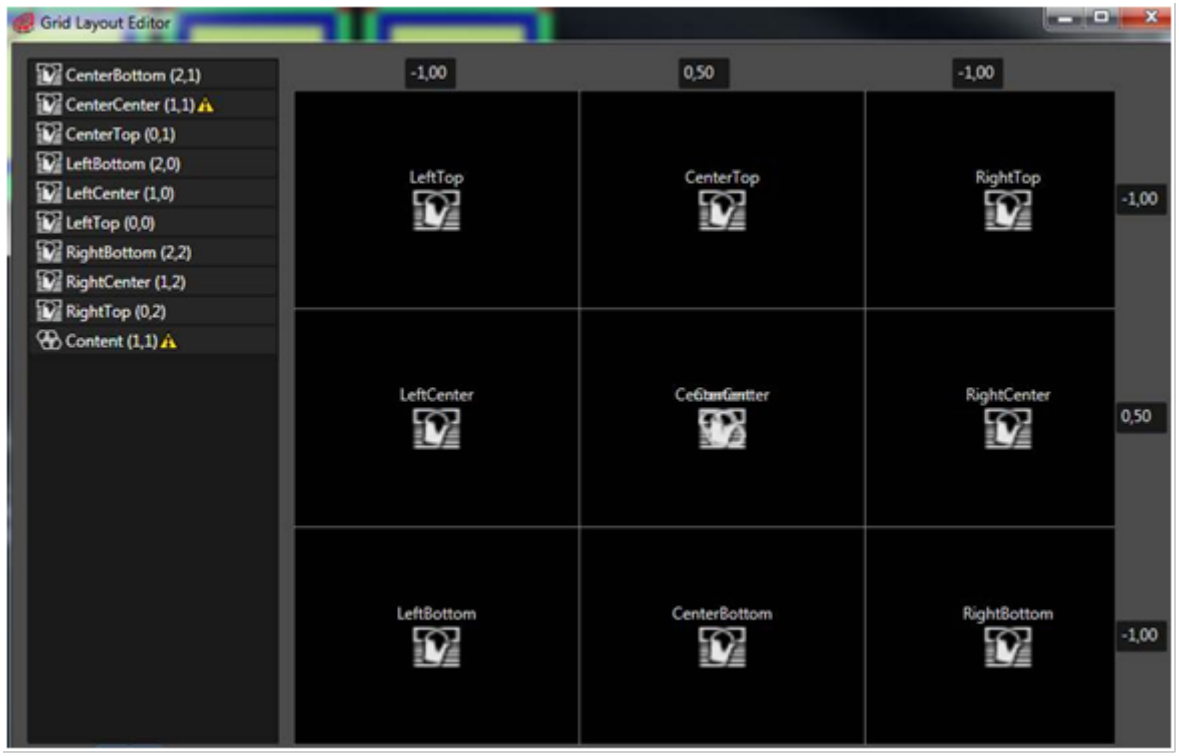
```
        <Rectangle MinWidth="10" MinHeight="20" Grid.Row="1" Grid.Column="1" Fill="Blue" />
        <Grid Grid.Row="1" Grid.Column="1" >
            <Rectangle Margin="10,10,10,10" HorizontalAlignment="Stretch" Width="100" MinHeight=
        </Grid>
    </Grid>
</Canvas>
```

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.

Structural View

Design Constraints

Layouters for 2D scenes

CGI Studio Layouters are currently available for the 2D scene environment only. However the application of layouters with the 3D Canvas is being prepared.

Disable 2D layout

To save computing power and code size for low end devices, the CMake feature flag CANDERA_LAYOUT_ENABLED has been introduced. This flag allows to remove all code needed for dynamic layout calculation. To achieve this, the complete code for layout calculation are pooled in following folder:

cgi_studio_candera/src/Candera/Engine2D/Layout

By default, layout calculation is enabled, the flag is set to ON.

Design Considerations

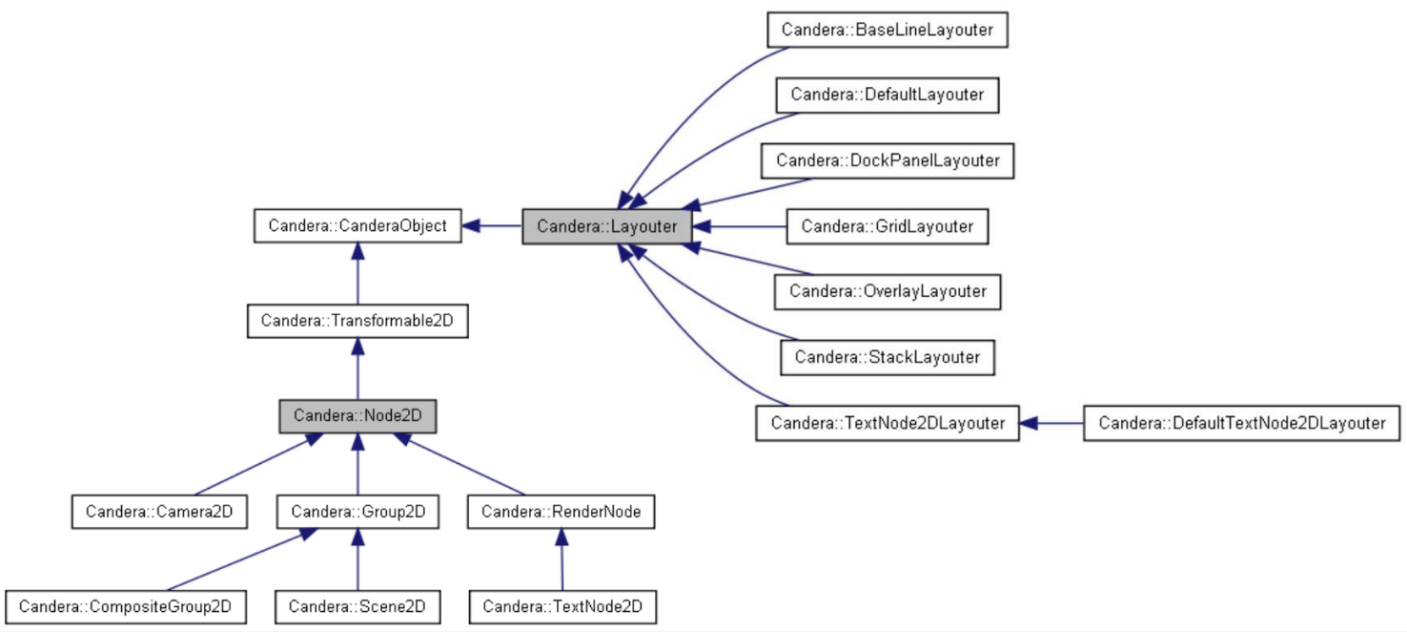
Default Layouter

For all nodes there is a Default Layouter available that behaves conform to the WPF Canvas control: Elements can be put into the scene and positioned using explicit coordinates.

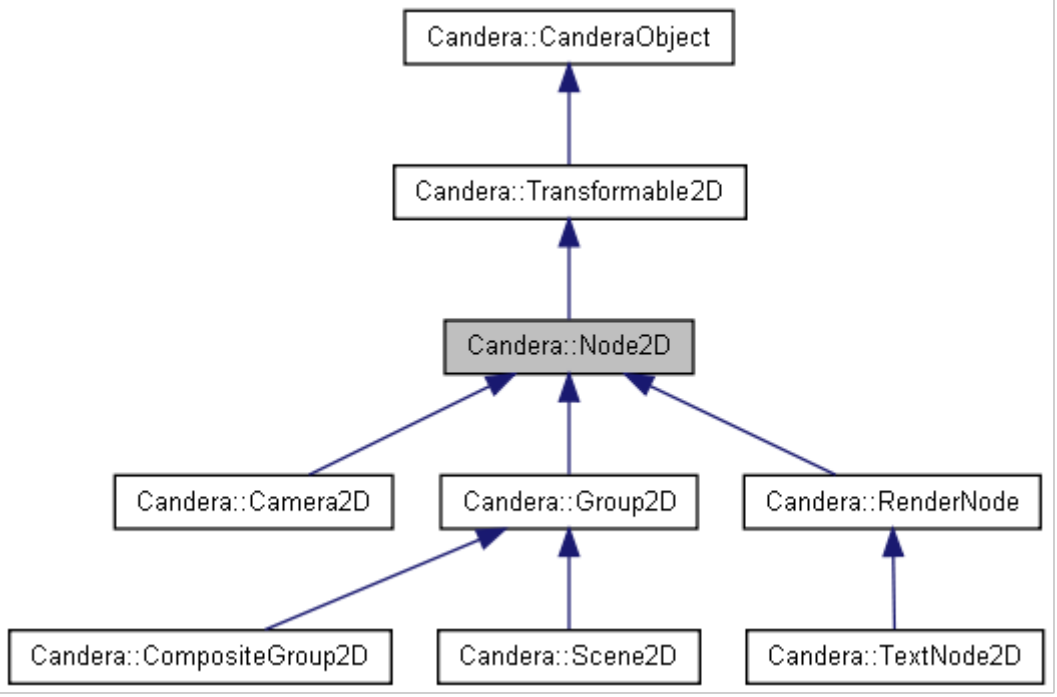
Dynamic Properties

Layout properties have been realized as dynamical properties of nodes. This has been done due to the fact that the use of layouters is optional and this way, memory can be saved.

Software Elements



Node2D and its Subclasses



Node2D
The class Node2D is an abstract base class for all 2D scene graph nodes. Each node defines a local coordinate system relative to the coordinate system of the parent node. The functionality to define the local coordinate system is derived from Transformable2D and consists of following components: translation, rotation, scale and a generic matrix called transform matrix. If the node is transformed from the local coordinate system to the world's coordinate system, then the node's local transformation is multiplied with all its parents’ transformations. A Node2D can also store a set of nodes as its children but a node can at most have one parent at a time. Cycles are prohibited.

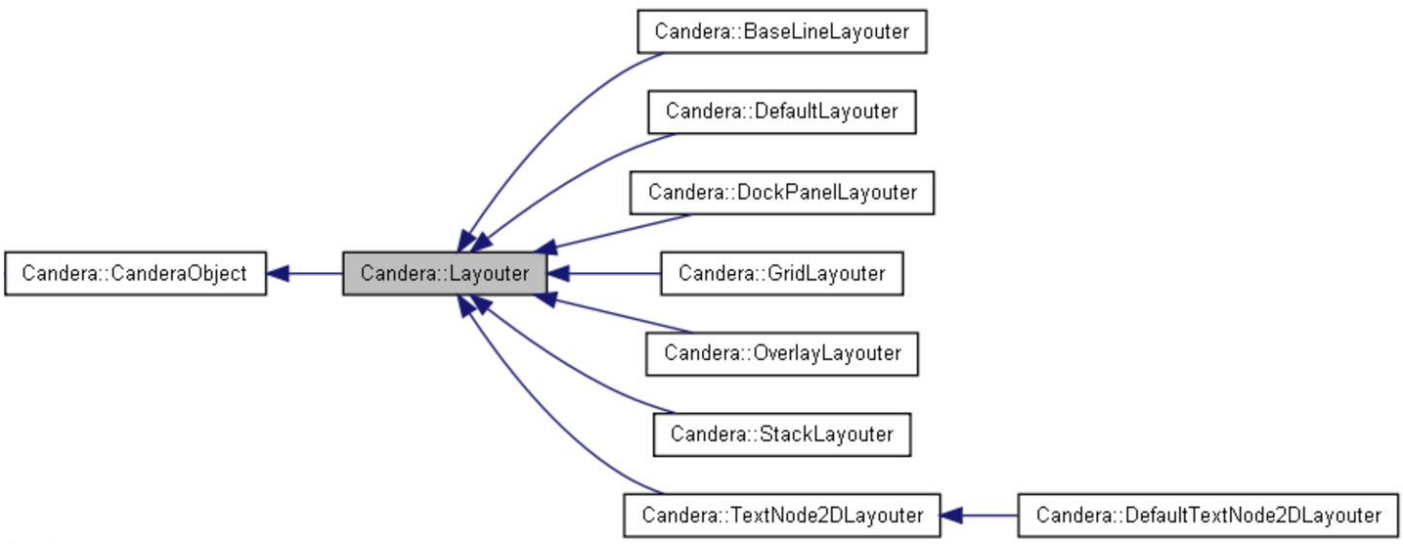
RenderNode
RenderNode is a 2D node that links to an effect chain. Therefore this is the only type of 2D node which has content to be rendered. All nodes of type RenderNode are added to a list which is sorted regarding each node's render order rank. The render order rank can be set using the SetRenderOrderRank method. The render order rank defines the relative value within render order. Nodes with lower render order rank are rendered before nodes with higher value, and therefore appear in the background. The effective render order rank is the sum of this node's and all its ancestor nodes' render order rank values.

TextNode2D
TextNode2D is a RenderNode that renders text using the attached BitmapBrush effect. To render text a FeatStd::String and a TextRendering::Style needs to be provided, as well as a TextNodeRenderer object which implements the strategy for providing Image2D objects, which will be rendered by the associated BitmapBrush. For text layout a TextNode2DLayouter should be attached to the TextNode2D.

Group2D
The class Group2D is a scene graph node that stores a set of nodes as its children. The parent-child relationship between groups and nodes is bidirectional. Whereas a group can hold a group of nodes, a node can at most have one parent at a time. Cycles are prohibited. Changing the groups' transformation or alpha value affects the child nodes of the group.

Scene2D
The class Scene2D represents a top level scene graph node. Multiple scenes are allowed. A scene cannot be part of any other scene.

Layouter and its Subclasses



Layouter
To layout a group of elements an instance of a Layouter has to be attached to a node. Child nodes act automatically as elements for the parent layouter. The order of the child nodes defines the order of the layout items.

DefaultLayouter
DefaultLayouter is one static instance used by all 2D nodes by default. This allows all nodes to get laid out without the need of adding a specialized layouter.

DockPanelLayouter
The Dock PanelLayouter provides an easy docking of elements to the left, right, top, bottom or center of the panel. The dock side of an element is defined by the attached dynamic DockSide property. To dock an element to the center of the panel, it must be the last child of the panel.

BaselineLayouter
A BaselineLayouter behaves similar to a Horizontal StackLayouter but instead of aligning the objects to the client area of the Layouter, it aligns them to a line. Horizontal stretching is disabled to avoid greedy items. Vertical stretching is disabled as result of DefaultLayouter::GetScale. The Vertical Alignment is interpreted as follows:

- VTop: the top of the child area is aligned to the base line.
- VBottom: the bottom of the child area is aligned to the base line.
- VCenter: the point defined by the child as its base point is aligned to the base line. This is typically in the center of an image, or on the base line of a text.
- VStretch: same as VCenter, as stretching is disabled. The baseline can be either fixed or automatic. The automatic baseline is configured such that all the children fit from the top of the client area.

TextNode2DLayouter
Base Layouter for TextNode2D nodes. The TextNode2DLayouter arranges and truncates the text associated with a TextNode2D.

DefaultTextNode2DLayouter
Default layouter for TextNode. The DefaultTextNode2DLayouter arranges and truncates the text associated with a TextNode2D.

StackLayouter
A StackLayouter stacks its elements either in horizontal or vertical order.

- Vertical stack: The preferred width of the stack is defined by the widest element.
- Horizontal stack: The preferred height of the stack is defined by the highest element.

OverlayLayouter
The OverlayLayouter sets sizes of all child elements to fulfill the given area.

GridLayouter
The GridLayouter arranges its elements in a grid. The amount of rows and columns for the grid must be specified. Each row and column can be configured with one float value:

- Automatic size (value < 0): The row height (or column width) will be defined by the highest value of its child elements. So the row or column is given exactly the amount of

space it needs, and no more.
- Proportional size (0.0 < value < 1.0): The space is divided between a group of rows or columns.
- Absolute size (value >= 1.0): Defines an exact (fixed) size in pixels.

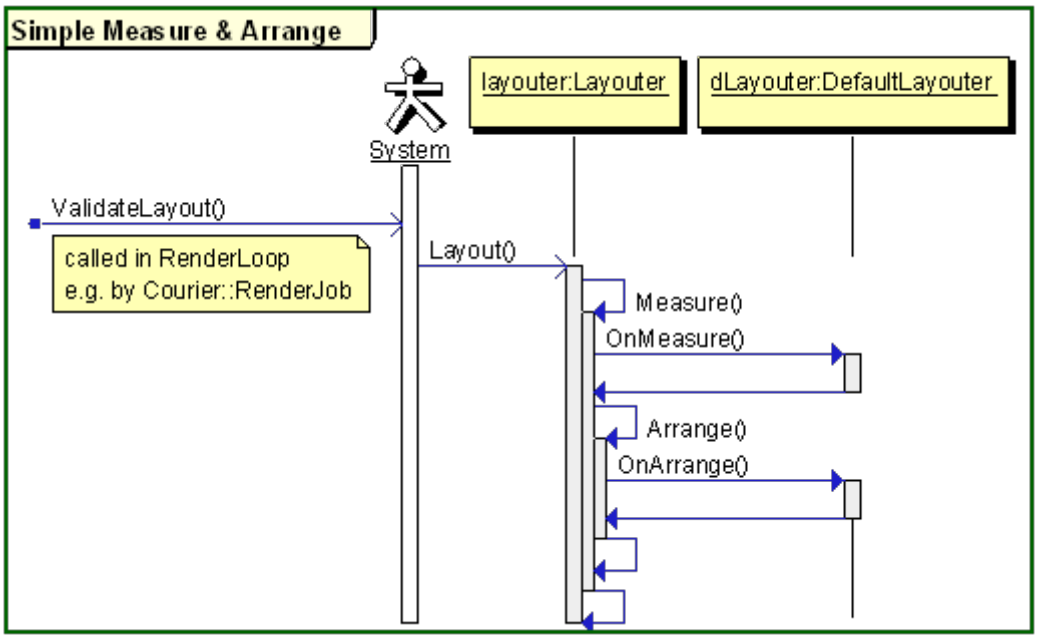
Dynamic View

Measure and Arrange

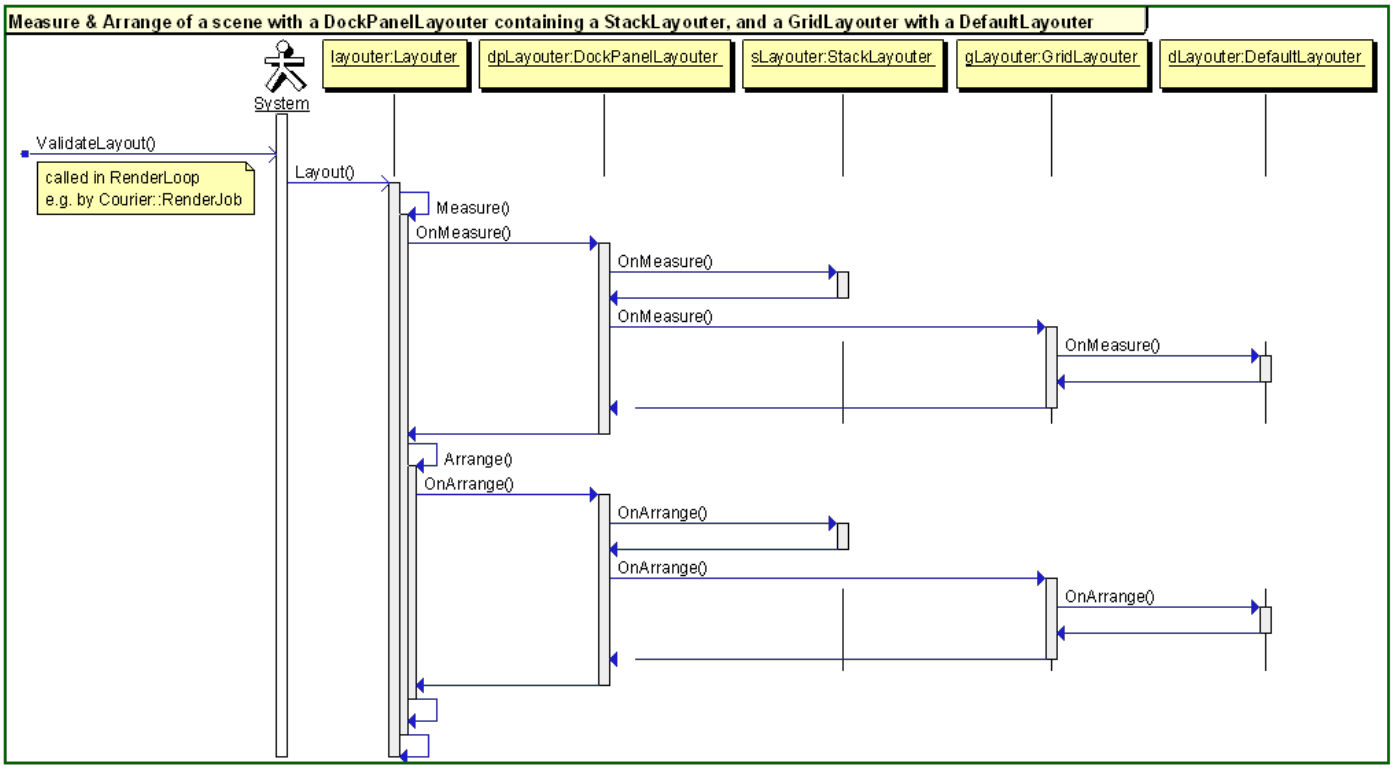
Computing the layout in CGI Studio consists of two calls: a measure call and an arrange call. In the measure call the system determines the elements' size requirements. The parent element tells its child elements how much space is available and the child elements tell their parent elements how much space they require. In the arrange call the layout system finalizes the size and position of the elements.

Sequence diagram for layout computation

For a simple 2D scene with a DefaultLayouter, the sequence of computing the layout is as follows: The RenderLoop (e.g. Courier::RenderJob) calls Scene2D::ValidateLayout(), which triggers a call to the Layouter::Layout(). Then the measure call is started with a call to Layouter::Measure(), that calls DefaultLayouter::OnMeasure(). After that, the arrange call is started with a call to Layouter::Arrange(), that calls DefaultLayouter::OnArrange().



For a scene with a DockPanelLayouter containing a StackLayouter and a GridLayouter with a DefaultLayouter, the sequence diagram looks as follows:


Measure & Arrange of a scene with a DockPanelLayouter containing a StackLayouter, and a GridLayouter with a DefaultLayouter
System
layouter:Layouter
dpLayouter:DockPanelLayouter
sLayouter:StackLayouter
gLayouter:GridLayouter
dLayouter:DefaultLayouter
ValidateLayout()
called in RenderLoop
e.g. by Courier::RenderJob
Layout()
Measure()
OnMeasure()
OnMeasure()
OnMeasure()
OnMeasure()
Arrange()
OnArrange()
OnArrange()
OnArrange()
OnArrange()

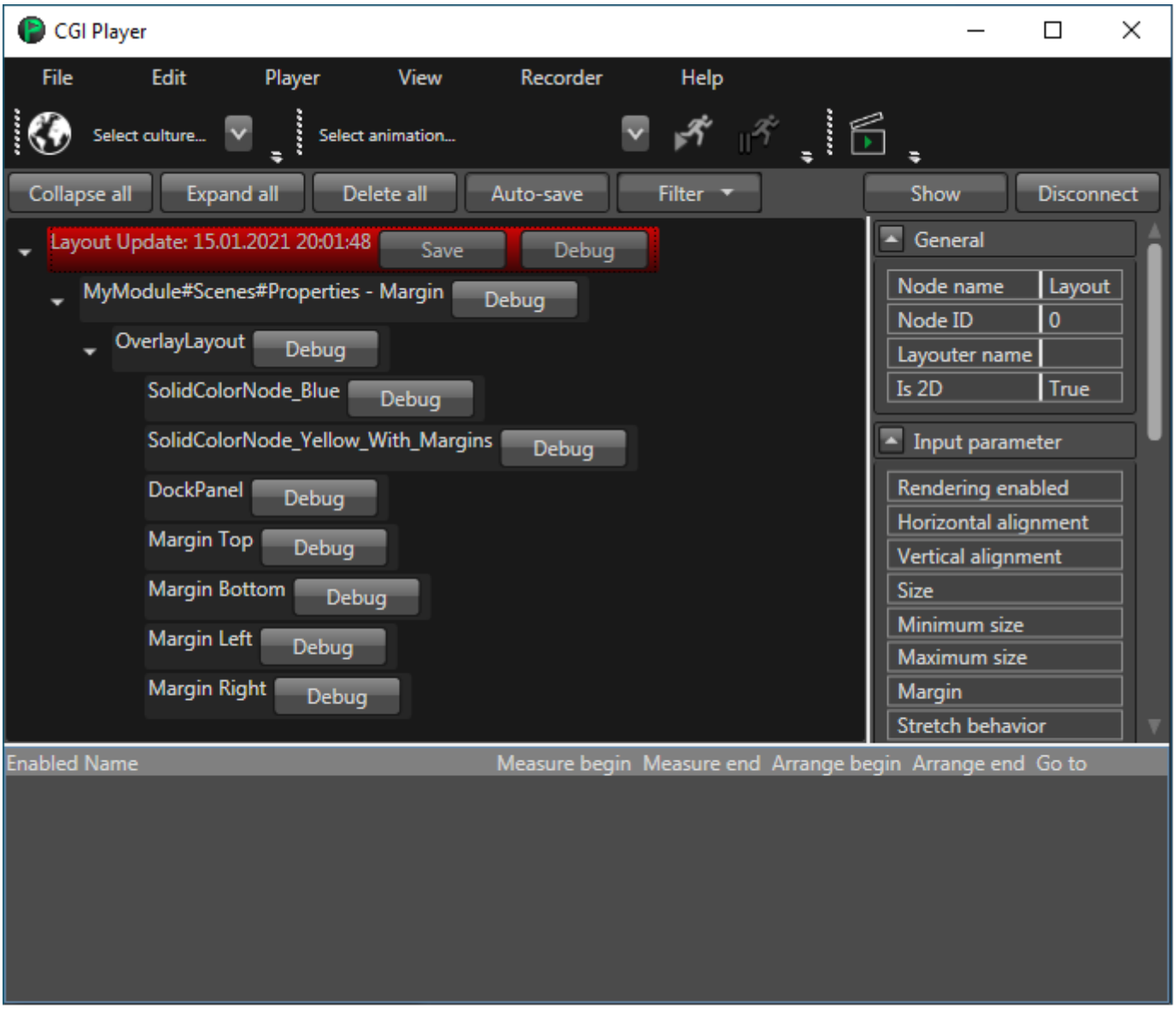
Layout Monitor

Introduction

The Layout Monitor is a feature that allows the user to see the relevant areas computed by the layouter as an overlay on top of the rendering of the application. The user can select nodes in the scene tree, the layouter rectangles are drawn automatically. The retrieval of various layout properties is available. Additionally the visualization is useful to understand how the dynamic layout is applied. The feature is available in the Player.

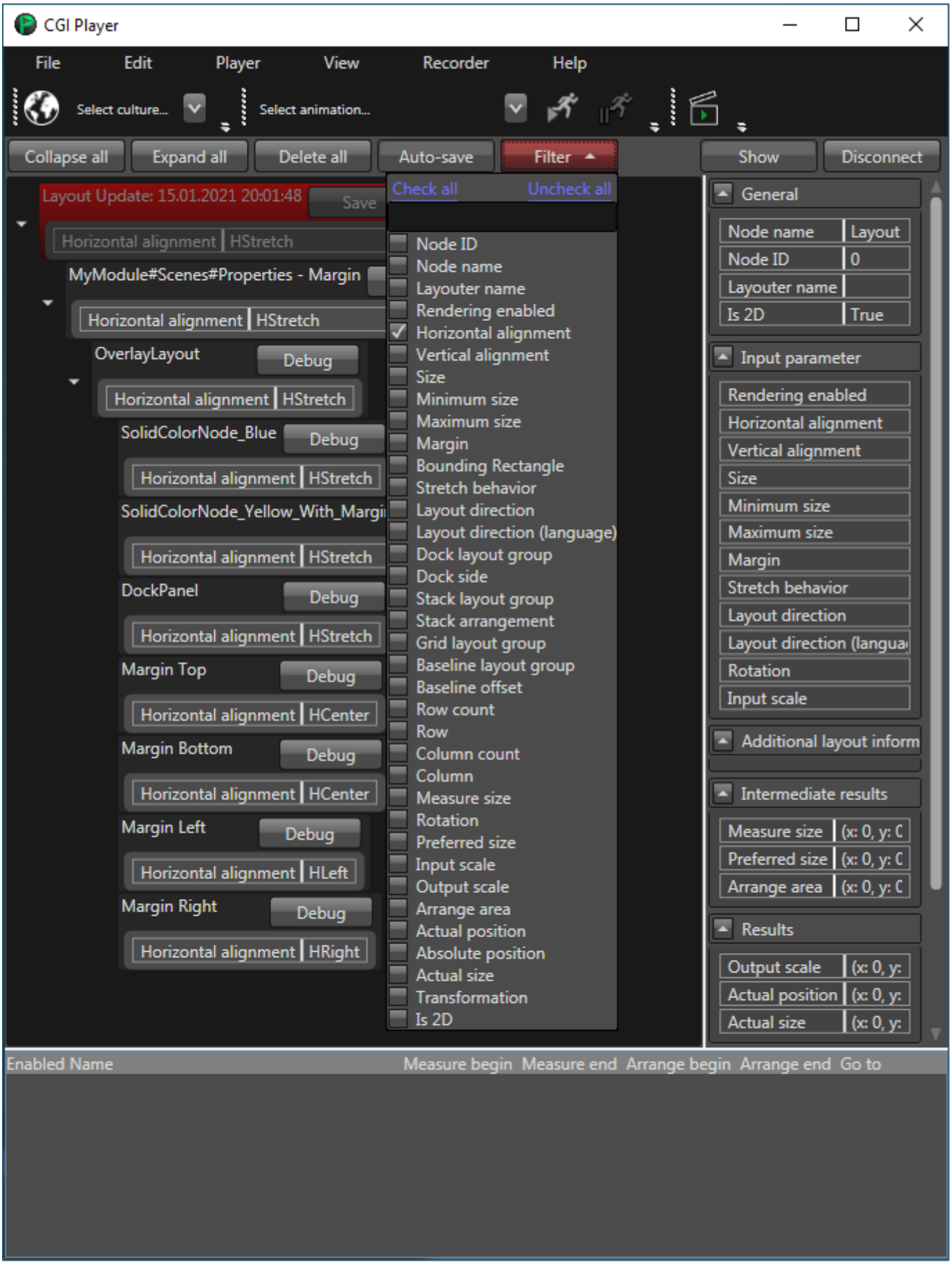
Layout Monitor user interface

The figure below shows the Layout Monitor user interface.



In the middle of the window, all the layout trees are displayed. To customize the view, these can be individually collapsed or expanded by clicking the small arrows to the left of the layout tree. It is also possible to collapse, expand and delete all layout trees by clicking the buttons above them. By clicking the Auto-save button, all incoming layout trees will automatically be saved to a location relative to the application workspace. The individual layout trees can be expended, collapsed and deleted by right clicking on them.

The layout trees can be filtered by clicking the Filter button as well as typing text in the textbox on top of the list. A dropdown list opens that enables choosing from various filters. These can be individually enabled or disabled. By clicking the according options in the list, all of the filters can be enabled or disabled at once as well. If a filter is enabled, it displays the according information inline with the layout tree. The figure below shows the filters available in Layout Monitor.

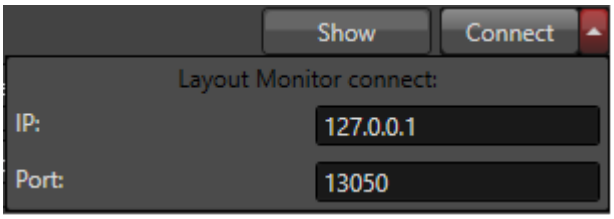


Clicking the Show button to the right shows the according overlay. For more information refer to Information shown in Layout Monitor. Clicking the Connect button connects to the target device.

The connection is required always (also on host) and must be enabled before the Layout Monitor is used. The IP-Address and the Port used in connecting can be changed in the Settings window which can be opened by clicking File -> Settings... as well as by opening the drop down menu to the right of the Connect button. For more information refer to Connection Setup.

Connection Setup

A connection to the target or to host can be established by clicking the Connect button to the right of the window. It is possible to adjust the settings of the connection by clicking the dropdown button to the right of it. In the dropdown window the IP address and the Port can be set. After the connection is established, it can receive layout trees.



> To produce less network traffic, disconnect and connect again to the target to receive events. This reduces network traffic considerably.

Layout Monitor specific settings

In the settings window, which can be opened by clicking File -> Settings... settings specific to the Layout Monitor can be changed. These settings are:

- The Layout entry directory is the default directory for saving logs which is used by the autosave- and recording-functions.
- The Max direct children is a threshold for maximal scene tree children, it needs to be changed only when more children are required in the scene tree.
- The IP-Adress and the Port indicate where the monitor connects to.
- The coordinate grid enables and disables the coordinate grid in the overlay.

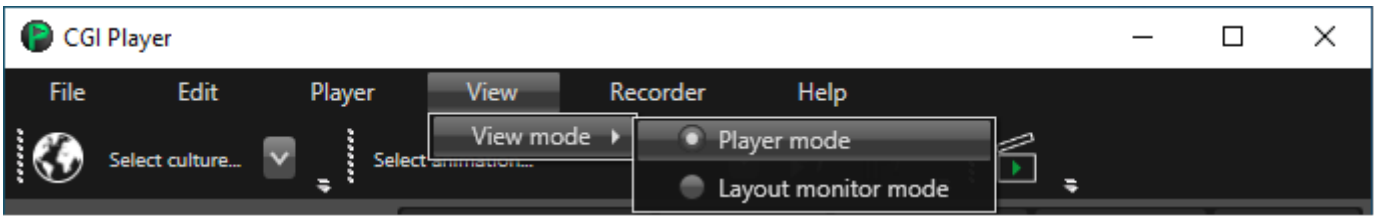
View Modes

The Layout Monitor mode can be used for applications that do not derive from the Player. These applications do not support all the other interfaces that would typically be required by the Player.

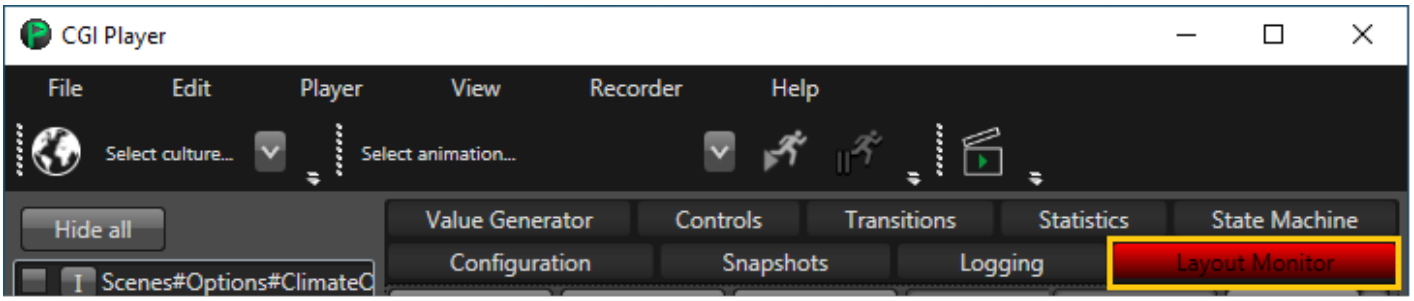
The Layout Monitor can be used in both of the CGIPanel modes:

- Player mode
- Layout Monitor mode (standalone mode)

The mode can be selected using the View -> View mode option.

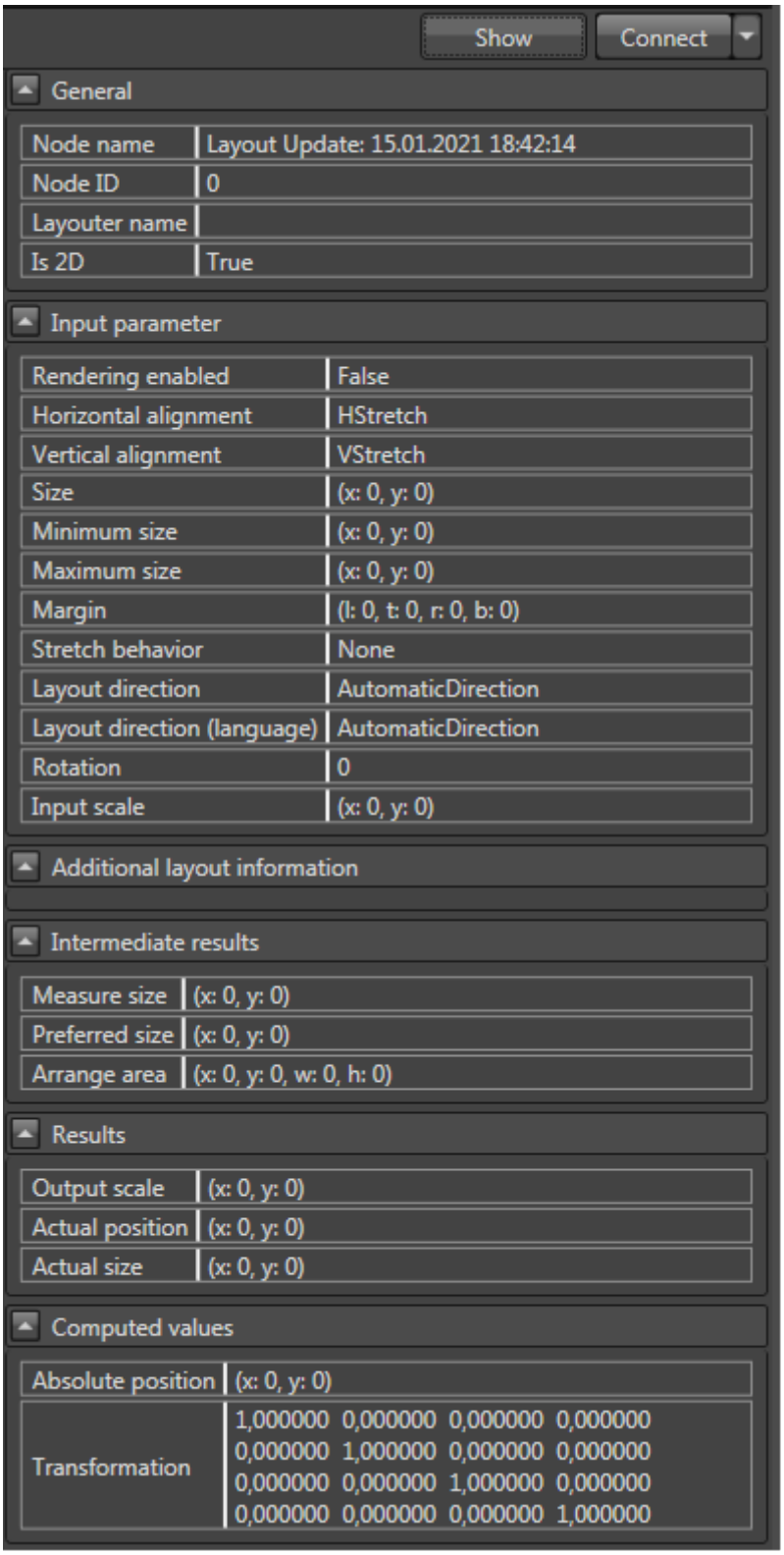


In Player mode, Layout Monitor can be accessed by pressing the "Layout Monitor" button, shown in the figure below.



Information shown in Layout Monitor

By clicking the Show button, Layout Monitor shows various information of the selected layout tree and also the overlays.



- The General dropdown field shows general information from the received layout tree.
- The Input parameter dropdown field shows information set within SceneComposer.
- The Additional layout information shows miscellaneous information that do not fit in the other categories.
- Intermediate values are values used as an intermediate result for the end results.
  - For example the Measure size defines the size a node would like to have under specific layout preconditions.
- The Results section shows the final values that define the node element.

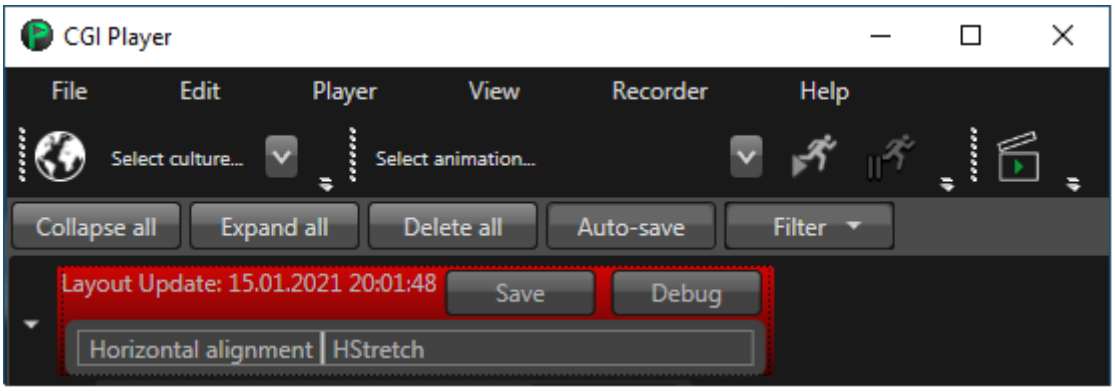
- The values shown in the Computed values dropdown field are computed directly on the host, not on the target.

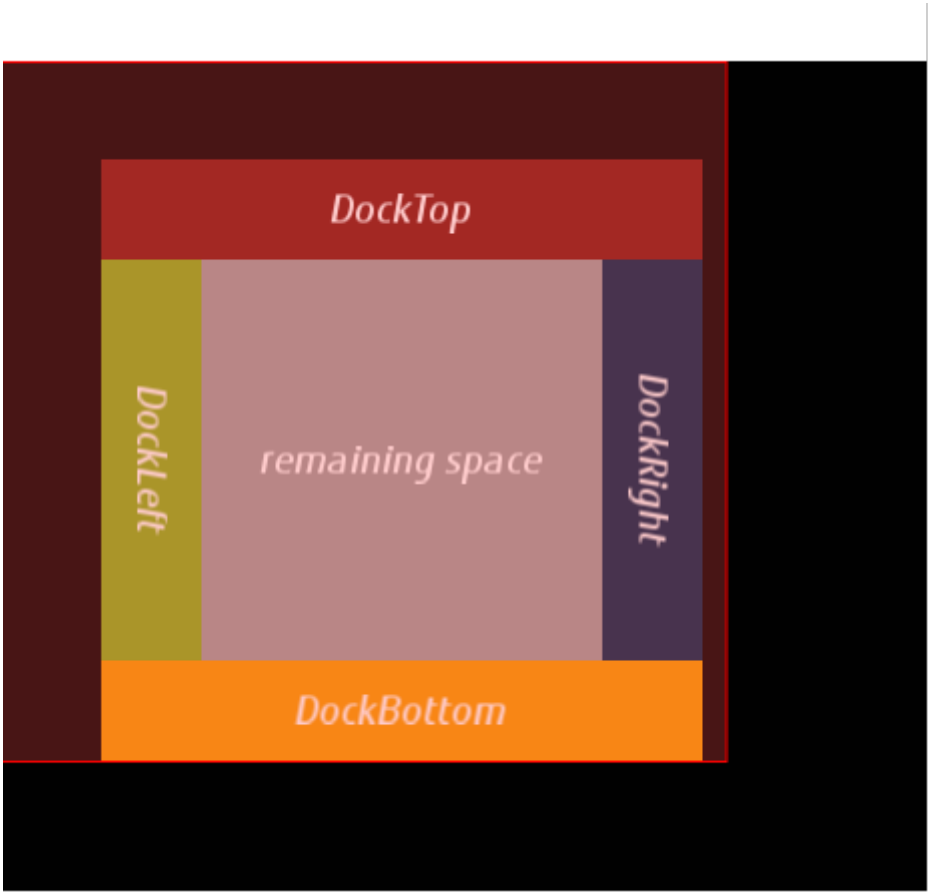
For every individual value there is also a tooltip available that explains the values in more detail.

By selecting nodes of the layout tree, differently colored rectangles are shown as an overlay on the screen.

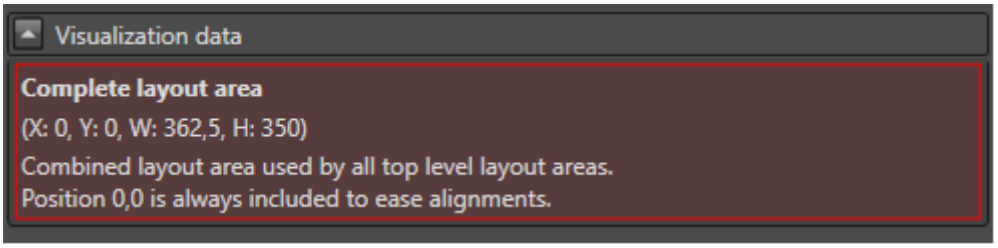
> These rectangles are a host-only overlay. If they should be applied to an application displayed on the target, screenshots have to be used. For more information on how to make screenshots, please refer to Player. To use a screenshot with the overlay, show the screenshot and the overlay. Then move the overlay over the screenshot.

If the top node is selected, a red rectangle appears. This rectangle can be moved freely and is to be placed atop the displayed layout. The top left corner of the overlay should be placed on the top left corner of the Player screen. This is also shown in the figure below. If the mouse is hovered above the rectangle, a tooltip appears that shows its size and position.

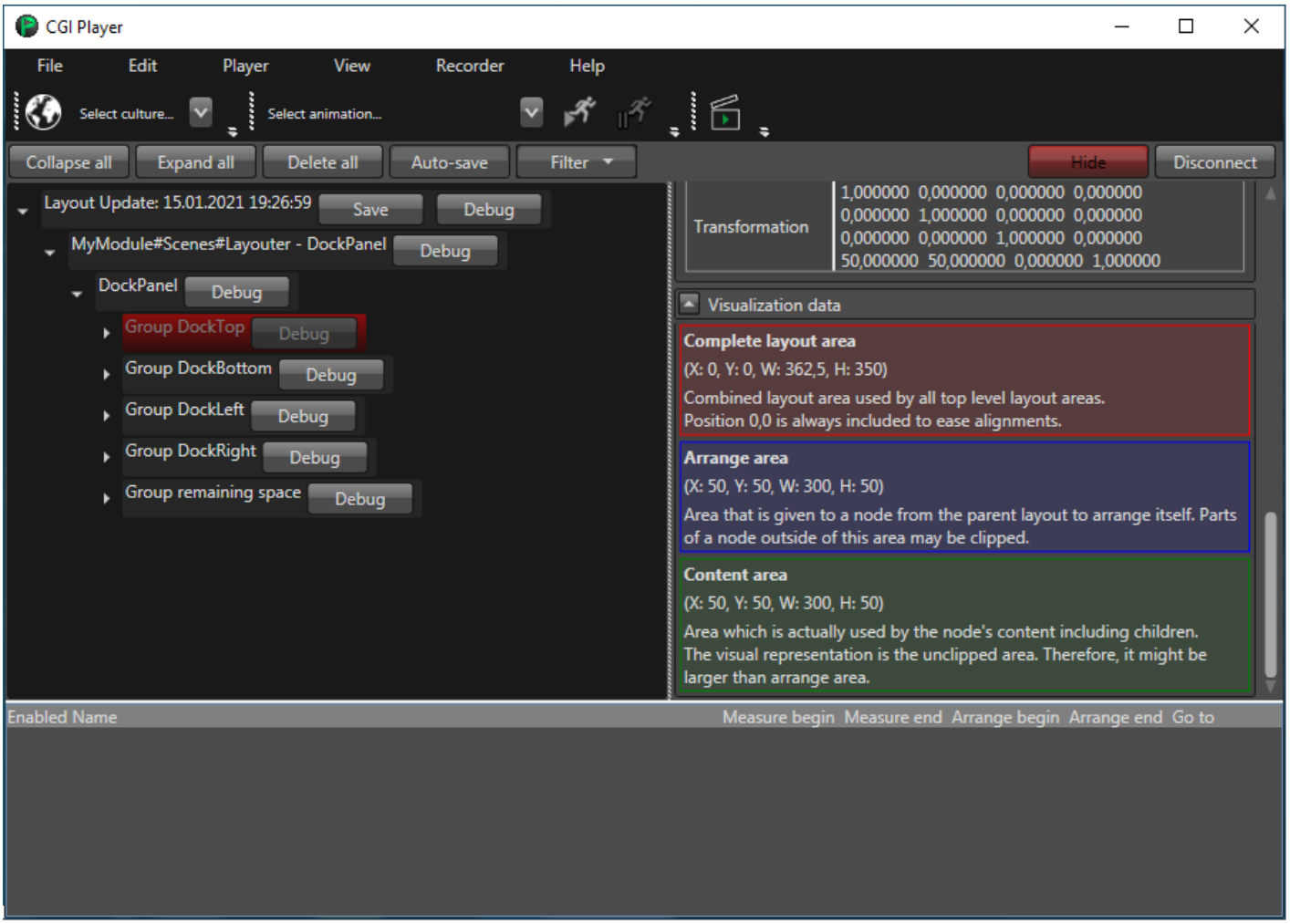




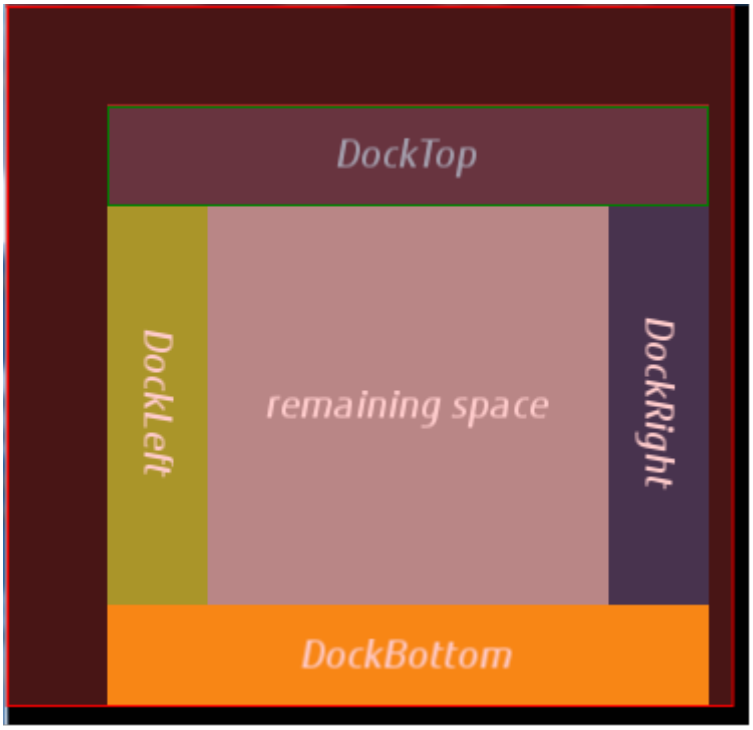
Additionally to the layout, the field Visualization data appears to the right of the user interface. This field shows the size, position and a description of the overlay. To help distinguishing between the different overlays, the different entries in the visualization data section of the informations panel to the right have the same color as the according overlay. In case of the top layer overlay, the field looks as displayed in the figure below.



If a sub-node is selected in Layout Monitor, rectangles appear that display all included parent nodes. As an example, SolidColorNode is selected. Now the Visualization data dropdown field contains three entries of the three shown overlays, their data and their description.



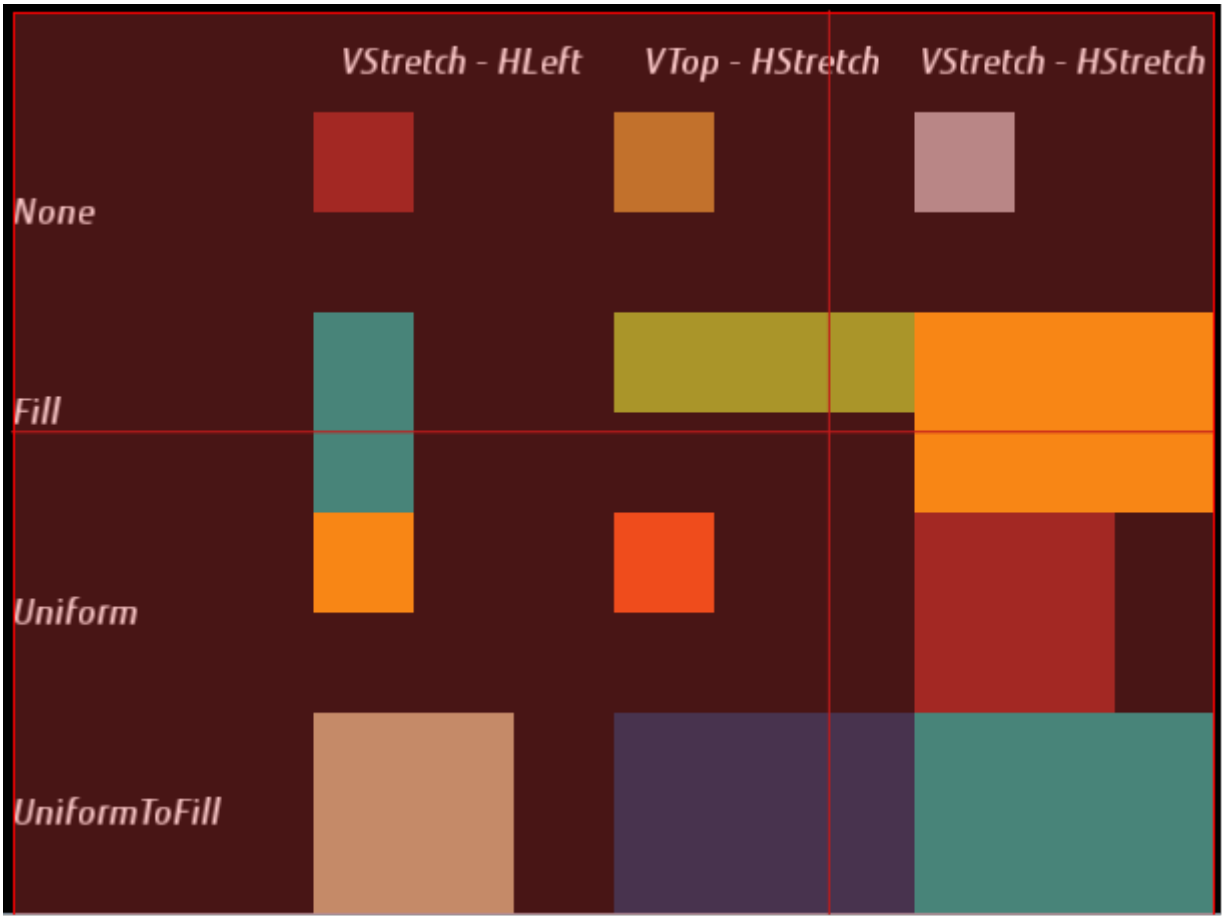
Now there are three overlays that show how the layouts are set.



It is also possible to show the axis (which is important for 3D usage). The axis serves as an aid to make the alignment of the overlays easier. The axis are always positioned to the 0,0,0 position. Using the mouse to hover over the show/hide button shows a menu which allows the user to

- enable/disable the axis
- set the axis color
- enable/disable the blinking

As is shown below, white lines appear which are the X and Y axis that indicate the 0/0 position and help aligning the overlay.



Debugging using Layout Monitor

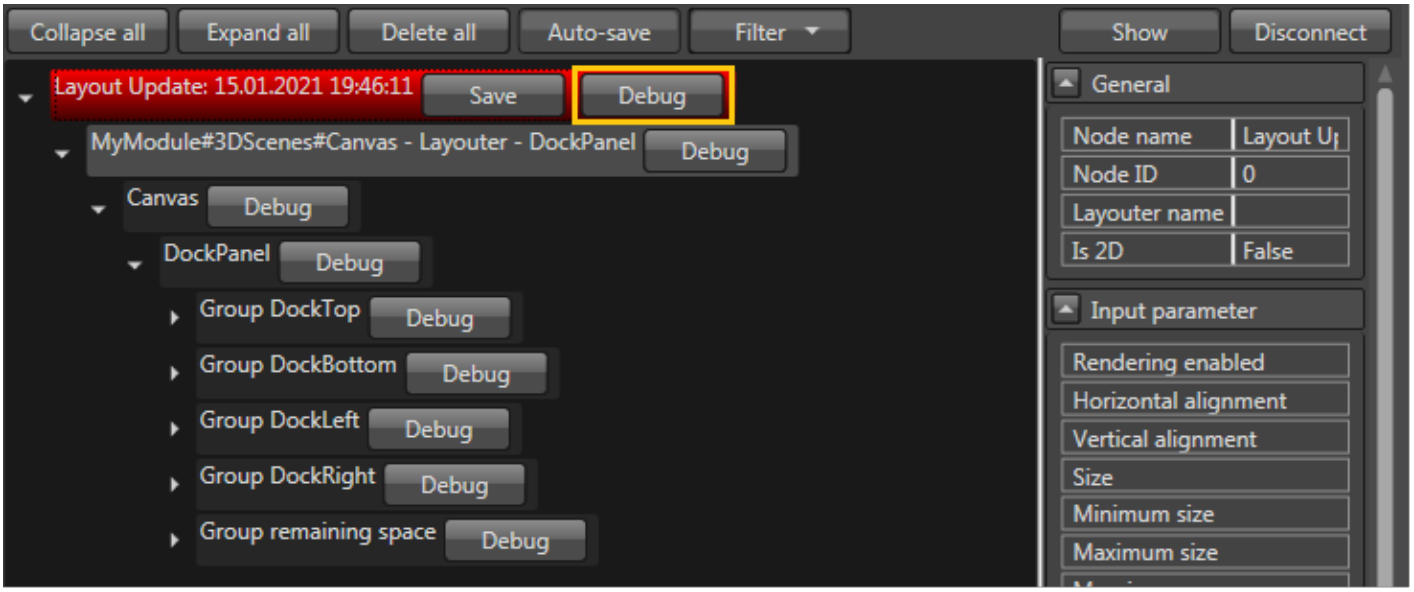
It is possible to debug either individual nodes or a parent node and its child nodes. To do this, simply press the debug button to the right of the node. If the node has child nodes, these will be debugged too.

The preconditions necessary for debugging are:

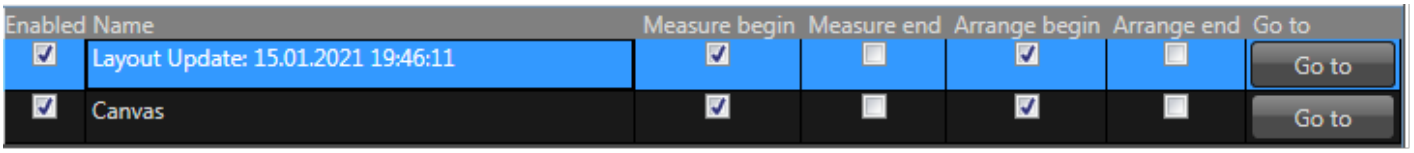
- Debug build of application
- JIT debugging must be enabled in Visual Studio
- Visual Studio must be present

Activate just-in-time-debugging in Visual Studio following these steps:

- In the Tools or Debug menu, select Options -> Debugging -> Just-in-time-debugger
- In the Enable Just-in-time debugging settings, ensure that "Native" is enabled



Clicking the debug button will cause breakpoints to appear on the bottom of the window. In this example, two breakpoints are set.



There are several options available for setting up breakpoints, which can be enabled or disabled using the checkoxes, as indicated in the Figure above. The options are:

- Enabled: breaks when node rendering is enabled or disabled
- Measure begin: breaks at the beginning of the measurement phase
- Measure end: breaks at the end of the measurement phase
- Arrange begin: breaks at the beginning of the arrange phase
- Arrange end: breaks at the end of the arrange phase

The breakpoints are actual code breakpoints so that Visual Studio is opened.

The Go to button selects the node in the layout tree.

To completely remove a breakpoint, select it so that it is marked blue and press the Delete button on the keyboard.

Saving the layout tree

It is possible to save the layout tree by clicking the Save button to the right of the main layout tree. It is then persistently available. Drag and drop can be used to import the previously saved tree into the Layout Monitor.

How to integrate Layout Monitor in existing custom applications

As a tutorial on how to integrate Layout Monitor into a custom application, please see following code sample:

LayoutMonitor.h must be included:

```
// Main include
#include <Candera/System/Diagnostics/LayoutMonitor.h>

// includes for the threading
#include <FeatStd/Platform/TcpServer.h>
#include <FeatStd/Platform/CriticalSectionLocker.h>
#include <FeatStd/Platform/Thread.h>
```

```
class LayoutMonitorListener : public FeatStd::Internal::Thread {
    public:
        static LayoutMonitorListener& GetInstance()
        {
            FEATSTD_UNSYNCED_STATIC_OBJECT(LayoutMonitorListener, s_inst);
            return s_inst;
        }

        void Stop()
        {
            {
                FeatStd::Internal::CriticalSectionLocker l(&m_memberSection);
                if (m_server != 0) {
                    m_server->Close();
                }
            }
            Candera::Diagnostics::LayoutMonitor::CloseStream();
        }

        bool Start(FeatStd::UInt32 port)
        {
            if (CreateServer(port)) {
                return Run();
            }
            return false;
        }

        LayoutMonitorListener() :m_server(0)
        {
```

```
            Candera::Diagnostics::LayoutMonitor::Register();
        }

        ~LayoutMonitorListener()
        {
            Candera::Diagnostics::LayoutMonitor::Unregister();
        }
    protected:
        bool CreateServer(FeatStd::UInt32 port)
        {
            FeatStd::Internal::CriticalSectionLocker l(&m_memberSection);
            if (m_server != 0) {
                return false;
            }
            m_server = FeatStd::Internal::TcpServer::Create(port);
            return m_server != 0;
        }

        virtual FeatStd::Int ThreadFn() override
        {
            do {
                FeatStd::Internal::IO::Stream * localStream = m_server->Accept();
                if (localStream != 0) {
                    Candera::Diagnostics::LayoutMonitor::Run(
FeatStd::Internal::IO::SharedStream::Create(localStream));
                }
            } while (m_server->IsOpen());
            if (m_server != 0) {
                FeatStd::Internal::CriticalSectionLocker l(&m_memberSection);
                FEATSTD_SAFE_DELETE(m_server);
            }
            Candera::Diagnostics::LayoutMonitor::CloseStream();
            return 0;
        }
    private:
        mutable FeatStd::Internal::CriticalSection m_memberSection;
        FeatStd::Internal::TcpServer * m_server;
    };
```

To start the monitor:

```
return LayoutMonitorListener::GetInstance().Start(CGIAPP_LAYOUTMONITOR_PORT); // default port: 1
```

To stop the monitor:

```
LayoutMonitorListener::GetInstance().Stop();
```