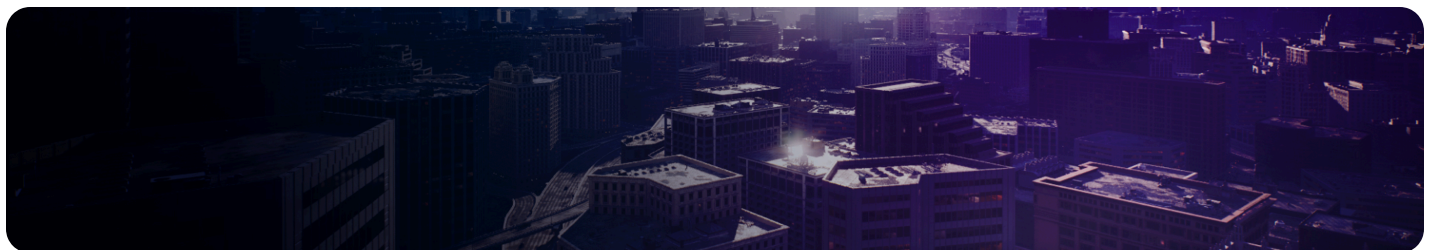


Slate Widget Examples

Layout and widget complexities not demonstrated in the Slate Viewer Widget Gallery.



Common Slate Arguments

The following arguments are common to every single widget that is created.

- **IsEnabled** - This will specify whether or not the widget is able to be interacted with. If it is disabled, it will be greyed out.
- **ToolTip** - This will specify what kind of custom `SToolTip` widget will be used for this widget's tool tip. If not specified, it will not appear.
- **ToolTipText** - This will specify what kind of text will show up as a simple tooltip for this widget's tool tip. If not specified, or if the `ToolTip` attribute was used, it will not appear.
- **Cursor** - This will specify what cursor will appear while the mouse is hovering over this widget.
- **Visibility** - See the Visibility section below.

The following arguments are not on every single widget, but they are on most widgets.

- **Text** - This will specify the text that this widget will have, if applicable.
- **Content** - This will specify what widget should be placed in the content section of the widget, if applicable.
- **ReadOnly** - This will prevent this widget from being editable if *true*.

- **Style** - This will specify the style of images or text font used by the widget. How this is applicable varies by widget.
- **[...]ColorAndOpacity** - The color and opacity of the widget or of the specified system.
- **Padding** - The padding of a widget amount of spacing in slate units around the left, top, right, and bottom parts of the widget within its parent. These can be specified as a single value for all four parts, or as a horizontal and vertical value, or as four separate values.
- **HAlign** - The horizontal alignment of content within the widget.
- **VAlign** - The vertical alignment of content within the widget.

Visibility

The visibility of a widget determines how the widget will appear, as well as its interactivity.

- **Visible (Default)** - The widget will appear normally.
- **Collapsed** - The widget will not be visible and will take up no space in the layout. It will not be interactive.
- **Hidden** - The widget will not be visible, but will take up space in the layout. It will not be interactive.
- **HitTestInvisible** - Visible to the user, but only as art. It will not be interactive.
- **SelfHitTestInvisible** - Same as HitTestInvisible, but does not apply to child widgets.

Alignment

The alignment of a widget determines the location of the widget within its parent. If a widget's parent is the same size (excluding padding) as the widget, then alignment is meaningless. This will happen by default with box slots, when you specify "Fill" for the widget, or a box slot specifies a Fill Size.

The possible alignments are listed below:

- **HAlign_Fill/VAlign_Fill**
- **HAlign_Left**
- **VAlign_Top**
- **HAlign_Center/VAlign_Center**
- **HAlign_Right**
- **VAlign_Bottom**

Box Panels

SHorizontalBox and **SVerticalBox** are the most common widgets for arranging your layout. These box panels are declared as widgets, but then they have slots inserted into them. SHorizontalBoxes have these slots arranged with the first widget on the left and the last to the right, while SVerticalBoxes have these slots arranged from top to bottom.

SScrollBox functions similar to SVerticalBox, with the exception that it allows vertical scrolling of the child slots.

Slot Attributes

- Width or Height Settings (The following options are mutually exclusive):
 - **Auto Size** (Default) - This will specify that the slot will fill up as much space as it needs, but no more. Alignment within slots does not matter here.
 - **Fill Size** - Specifying a fill coefficient will cause it to fill up the space based on the fill coefficients of other slots. Alignment of the slot in the same orientation does not matter here.
- **Max Size** - This will specify the maximum size that the slot can be in slate units.
- **Padding** - This will specify the padding of the slot within the panel.
- **Alignment** - This will determine how the child widget will be aligned within the slot. This option is meaningless with Fill Size specified in the same orientation.

Below is an example of both autosized horizontal boxes and fill sized horizontal boxes nested within a scrollbar. Also shown is how the alignment can be used with these slots.

```
1 SNew(SScrollBox)
2 +SScrollBox::Slot() .Padding(10,5)
3 [
4 SNew(SHorizontalBox)
5 +SHorizontalBox::Slot() .HAlign(HAlign_Left)
6 [
7 ...
8 ]
9 +SHorizontalBox::Slot() .HAlign(HAlign_Center)
10 [
11 ...
```

```

12 ]
13 +SHorizontalBox::Slot() .HAlign(HAlign_Right)
14 [
15 ...
16 ]
17 ]
18 +SScrollBox::Slot() .Padding(10,5)
19 [
20 SNew(SHorizontalBox)
21 +SHorizontalBox::Slot() .FillWidth(2)
22 [
23 ...
24 ]
25 +SHorizontalBox::Slot() .FillWidth(1)
26 [
27 ...
28 ]
29 +SHorizontalBox::Slot() .FillWidth(3)
30 [
31 ...
32 ]
33 ]
34

```

 Copy full snippet

Uniform Grid Panels

SUniformGridPanel is a panel that distributes its child widgets evenly both vertically and horizontally. Its child slots are specified using a pair of integers which specify the index of the child. Below is such a uniform grid panel.

```

1 SNew(SUniformGridPanel)
2 .SlotPadding( FMargin( 5.0f ) )
3 +SUniformGridPanel::Slot(0,0)
4 .HAlign(HAlign_Right)
5 [
6 ...
7 ]
8 +SUniformGridPanel::Slot(0,1)

```

```
9  .HAlign(HAlign_Right)
10 [
11 ...
12 ]
13 +SUniformGridPanel::Slot(0,2)
14 .HAlign(HAlign_Center)
15 [
16 ...
17 ]
18 +SUniformGridPanel::Slot(1,0)
19 [
20 ...
21 ]
22 +SUniformGridPanel::Slot(1,1)
23 [
24 ...
25 ]
26
```

 Copy full snippet

Wrap Boxes

SWrapBox is a box that lays out widgets horizontally until these widgets exceed a width, at which point they will be placed on the next line, and so on. Below is an example of such.

```
SNew(SWrapBox)
.PreferredWidth( 300.f )
+SWrapBox::Slot()
.Padding( 5 )
.VAlign(VAlign_Top)
[
    ...
]
+SWrapBox::Slot()
.Padding( 5 )
.VAlign(VAlign_Bottom)
[
    ...
]
```

```

+WrapBox::Slot()
.Padding( FMargin(20, 5, 0, 5) )
.VAlign(VAlign_Center)
[
    ...
]
+WrapBox::Slot()
.Padding( 0 )
.VAlign(VAlign_Fill)
[
    ...
]

```

 Copy full snippet

Radio Buttons

Radio Buttons are check boxes in slate, because check boxes require a delegate for how to determine whether they are checked. To make a series of radio buttons, the easiest way to do this is to create an enum which determines which check box is checked. Save an instance of the enum which determines the current choice. Then, for the checking delegate, pass in a function which compares a passed in payload of the correct enum with the current choice. On changing the selection, the current choice needs to be updated.

```

ERadioChoice CurrentChoice;

...

ECheckBoxState::Type IsRadioChecked( ERadioChoice ButtonId ) const
{
    return (CurrentChoice == ButtonId)
        ? ECheckBoxState::Checked
        : ECheckBoxState::Unchecked;
}

...


void OnRadioChanged( ERadioChoice RadioThatChanged, ECheckBoxState::Type NewRadio

```

```

{
    if (NewRadioState == ECheckBoxState::Checked)
    {
        CurrentChoice = RadioThatChanged;
    }
}

```

 Copy full snippet

Menus and Toolbars

Menus

To create menus or toolbars, use multiboxes. Commands should use Slate's UI_COMMAND system, though this cannot be done with dynamically generated buttons/controls.

To create a menu, create an **FMenuBarBuilder** with an **FUICommandList** passed in. Finally, you can call **MakeWidget()** on the menu bar builder to get a widget to place.

```

FMenuBarBuilder MenuBarBuilder( CommandList );
{
    MenuBarBuilder.AddPullDownMenu( TEXT("Menu 1"), TEXT("Opens Menu 1"), FNewMenu1

    MenuBarBuilder.AddPullDownMenu( TEXT("Menu 2"), TEXT("Opens Menu 2"), FNewMenu1
}

return MenuBarBuilder.MakeWidget();

```

 Copy full snippet



In the menus that are created, you can add menu headers, entries, separators, submenus, editable text, or custom widgets to the menu that gets created.

```

static void FillMenu1Entries( FMenuBarBuilder& MenuBuilder )
{
    MenuBuilder.AddEditableText( TEXT( "Editable Item" ), TEXT( "You can edit this

```

```

MenuBuilder.AddMenuEntry( FMultiBoxTestCommandList::Get().EighthCommandInfo );

MenuBuilder.AddMenuSeparator();

MenuBuilder.AddSubMenu( TEXT("Sub Menu"), TEXT("Opens a submenu"), FNewMenuDele

MenuBuilder.AddWidget(SNew(SVolumeControl), TEXT("Volume"));
}

```

 Copy full snippet

Context Menus

To summon one of these menus as a context menu, get the widget generated from an **FMenuBuilder**, and pass it into the **PushMenu()** function, as shown here.

```

FSlateApplication::Get().PushMenu(
    MenuBuilder.MakeWidget(),
    MouseCursorLocation,
    FPopupTransitionEffect( FPopupTransitionEffect::ContextMenu )
);

```

 Copy full snippet

Toolbars

To create a toolbar, use an **FToolBarBuilder** instead. For children, you can add tool bar buttons, combo buttons, plain buttons, and pull down menus.

```

FToolBarBuilder GameToolBarBuilder( InCommandList );
{
    GameToolBarBuilder.AddToolBarButton( FLevelEditorCommands::Get().Simulate );

    GameToolBarBuilder.AddToolBarButton(

```



```

FLevelEditorCommands::Get().RepeatLastPlay,
LOCTEXT("RepeatLastPlay", "Play"),
TAttribute< FString >::Create( TAttribute< FString >::FGetter::CreateRaw( &FLevelEditorCommands::RepeatLastPlay,
TAttribute< const FSlateBrush* >::Create( TAttribute< const FSlateBrush* >::FGetter::CreateRaw( &FLevelEditorCommands::RepeatLastPlay,
);

GameToolBarBuilder.AddComboButton(
    SpecialPIEOptionsMenuAction,
    FOnGetContent::CreateRaw( &FLevelEditorToolBar::GeneratePIEMenuContent, InContent ),
    FText(),
    LOCTEXT("PIEComboToolTip", "Play-In-Editor options") );
}

return GameToolBarBuilder.MakeWidget();

```

 Copy full snippet

Item Views

These views take a template argument of a shared pointer to some sort of data. They are populated by a **TArray** or **TSharedPtr**'s to the data type. Their internal widgets are populated dynamically with the **OnGenerateRow** or **OnGenerateTile** delegate passed in, and different widgets are generated for each column.

List Views

List views are widgets which store a list of child widgets. They are made as

```
SListView<...>.
```

```

SNew( SListView< TSharedPtr<FTestData> > )
    .ItemHeight(24)
    .ListItemsSource( &Items )
    .OnGenerateRow( this, &TableViewTesting::OnGenerateWidgetForList )
    .OnContextMenuOpening( this, &TableViewTesting::GetListContextMenu )
    .SelectionMode( this, &TableViewTesting::GetSelectionMode )
    .HeaderRow

```

```
(
  SNew(SHeaderRow)
+ SHeaderRow::Column("Name")
[
  SNew(SBorder)
  .Padding(5)
  .Content()
  [
    SNew(STextBlock)
    .Text(TEXT("Name"))
  ]
]
+ SHeaderRow::Column("Number") .DefaultLabel(TEXT("Number"))
+ SHeaderRow::Column("TextField") .DefaultLabel(TEXT("Text Field"))
+ SHeaderRow::Column("TextBlock") .DefaultLabel(TEXT("Text Block"))
+ SHeaderRow::Column("AddChild") .DefaultLabel(TEXT("Add Child"))
)
```

 Copy full snippet

Tree Views

Tree views, made as `STreeView<...>`, are similar to list views, except that they support parenting widgets to other widgets in the list. To determine which widgets are parented to which, the **OnGetChildren** delegate is used to pass back the children of the passed in item.

```
SNew( STreeView< TSharedPtr<FTestData> > )
.ItemHeight(24)
.TreeItemsSource( &Items )
.OnGenerateRow( this, &STableViewTesting::OnGenerateWidgetForTree )
.OnGetChildren( this, &STableViewTesting::OnGetChildrenForTree )
.OnContextMenuOpening( this, &STableViewTesting::GetTreeContextMenu )
.SelectionMode( this, &STableViewTesting::GetSelectionMode )
.HeaderRow
(
  SNew(SHeaderRow)
+ SHeaderRow::Column("Name") .DefaultLabel(TEXT("Name"))
+ SHeaderRow::Column("Number") .DefaultLabel(TEXT("Number"))
+ SHeaderRow::Column("TextField") .DefaultLabel(TEXT("Text Field"))
)
```


```
+ SHeaderRow::Column("TextBlock") .DefaultLabel(TEXT("Text Block"))  
+ SHeaderRow::Column("AddChild") .DefaultLabel(TEXT("Add Child"))  
)
```

 Copy full snippet

Tile Views

Tile views, made as `STileView<...>`, are similar to list views, except the child widgets are laid out in a grid rather than a list. Because of this, they do not support columns or column headers.

```
SNew( STileView< TSharedPtr<FTestData> > )  
.ItemWidth(128)  
.ItemHeight(64)  
.ListItemsSource( &Items )  
.OnGenerateTile( this, &STableViewTesting::OnGenerateWidgetForTileView )  
.OnContextMenuOpening( this, &STableViewTesting::GetTileViewContextMenu )
```

 Copy full snippet

```
.SelectionMode( this, &STableViewTesting::GetSelectionMode )
```