Falagard skinning system for CEGUI

A tutorial and reference

Version 1.0

written by
Paul D. Turner

Table of Contents

Part One

Tutorial Style Introduction

Introduction and overview

The Falagard skinning system for CEGUI consists partly of a set of enhancements to the CEGUI base library, and partly of a new 'look' module called "CEGUIFalagardBase". Combined, these elements are intended to make it easier to create custom skins or 'looks' for CEGUI window and widget elements.

The Falagard system is designed to allow widget imagery specification, sub-widget layout, and default property initialisers to be specified via XML files rather than in C++ or scripted code (which, before now, was the only way to do these things).

The system is named "Falagard" after the forum name of the person who initially suggested the feature (as is the trend in all things CEGUI), although it was designed and implemented by the core CEGUI team.

The Falagard extensions are not limited to the 'looknfeel' XML files only; there are supporting elements within the core library, as well as extensions to the GUI scheme system to allow you to create what are essentially new widget types. This is achieved by mapping a named widget 'look' to a base widget type taken from the CEGUIFalagardBase module (I know I'm probably just about losing you now, don't worry about all these details too much for the time being!).

Once your new type has been defined in a scheme and loaded, you can specify the name of that new type name when creating windows or widgets via the WindowManager singleton as you would for any other widget type. There are no additional issues to be considered when using a 'skinned' widget than when using one of the old 'programmed' widget types.

The Unified Co-ordinate System

As part of the Falagard system, CEGUI has effectively replaced the old either/or approach to relative and absolute co-ordinates with a new 'unified' co-ordinate system. Using this new system, each co-ordinate can specify both a parent-relative and absolute-pixel component. Since most people baulk at the idea of this, I'll use examples to introduce these concepts.

Introducing UDim

The basic building block of the unified system is the UDim, which is defined as:

```
UDim(scale, offset)
```

where:

- 'scale' represents what would be the relative component, as is usually a value between 0 and 1.0.
- 'offset' represents the absolute component and basically represents a number of screen

'pixels'.

Still confused? On to the examples!

Simple UDim examples

Example 1

```
UDim(0, 10)
```

Here we see a UDim with a scale of 0, and an offset of 10. This simply represents an absolute value of 10, if you used such a UDim to set a window width, then under the old system it's the equivalent of:

```
myWindow->setWidth(Absolute, 10);
```

Example 2

```
UDim(0.25f, 0)
```

Here we see a UDim with a scale of 0.25 and an offset of 0. This represents a simple relative co-ordinate. If you were to set the y position of a window using this UDim, then the window would be a quarter of the way down it's parent, and it's the same as the following under the old system:

```
myWindow->setYPosition(Relative, 0.25f);
```

Example 3

```
UDim(0.33f, -15)
```

Here we see the power of UDim. We have a scale of 0.33 and an offset of -15. If we used this as the height of a window, you would get a height that is approximately one third of the height of the window's parent, minus 15 pixels. There is no simple equivalent for this under the old system.

Property format

The format of a UDim to be used in the window property strings is as follows:

```
{s,o}
```

where:

- 's' is the scale value
- 'o' is the pixel offset.

The UVector2 type

There is a UVector2 type which consists of two UDim elements; one for the x axis, and one for the y axis. Note that the UVector2 is used to specify both positional points and also sizes (i.e. there is no separate USize type).

The UVector2 is defined as:

```
UVector2(x_udim, y_udim)
```

where:

- 'x_udim' is a UDim value that specifies the x co-ordinate or width.
- 'y_udim' is a UDim value that specifies the y co-ordinate or height.

Simple UVector2 examples

Example 1

```
UVector2(UDim(0, 25), UDim(0.2f, 12))
```

The above example specifies a point that is 25 pixels along the x-axis and one fifth of the way down the parent window plus twelve pixels.

Example 2

```
UVector(UDim(1.0f, -25), UDim(1.0f, -25))
```

This example, intended as a size for a window, would give the window the same width as its parent, minus 25 pixels, and the same height as its parent, minus 25 pixels.

Property format

The format of a UVector2 to be used in the window property strings is as follows:

```
{{sx,ox},{sy,oy}}
```

where:

- 'sx' is the scale value for the x-axis, and 'ox' is the pixel offset for the x-axis.
- 'sy' is the scale value for the y-axis, and 'oy' is the pixel offset for the y-axis.

Finally the URect

The last of the Unified co-ordinate types is URect. The URect defines four sides of a rectangle using UDim elements. You generally access the URect as you would the normal 'Rect' type (though each 'side' of the rectangle is made up of a UDim rather than a float):

```
URect(left_udim, top_udim, right_udim, bottom_udim)
```

where:

- 'left_udim' is a UDim defining the left edge.
- 'top_udim' is a UDim defining the top edge.
- 'right_udim' is a UDim defining the right edge.
- 'bottom_udim' is a UDim defining the bottom edge.

You can also define a URect with two UVector2 objects; one for the top-left corner, and the other for the bottom-right corner:

```
URect(tl_uvec2, br_uvec2)
```

where:

- 'tl_uvec2' is a UVector2 that describes the top-left point of the rect area.
- 'br_uvec2' is a UVector2 that describes the bottom-right point of the rect area (not the size of the area).

Simple URect Example

```
URect(UDim(0, 25),
      UDim(0, 25),
      UDim(1.0f, -25),
      UDim(1.0f, -25))
```

If we used the URect defined here to specify the area for a window, we would get a window that was 25 pixels smaller than its parent on each edge.

Property format

The format of a URect to be used in the window property strings is as follows:

```
{{sl,ol},{st,ot},{sr,or},{sb,ob}}
```

where:

- 'sl' is the scale value for the left edge, and 'ol' is the pixel offset for the left edge.
- 'st' is the scale value for the top edge, and 'ot' is the pixel offset for the top edge.
- 'sr' is the scale value for the right edge, and 'or' is the pixel offset for the right edge.
- 'sb' is the scale value for the bottom edge, and 'ob' is the pixel offset for the bottom edge.

New Window Alignments

The Falagard enhancements also include new settings to specify alignments for windows. This gives the possibility to position child windows from the right edge, bottom edge and centre positions of their parents, as well as the previous left edge and top edge possibilities.

It is possible to set the alignment options in C++ code by using methods in the Window class, and also via the properties system which is used by XML layouts system.

Vertical Alignments

To set the vertical alignment use the Window class member function:

```
void setVerticalAlignment(const VerticalAlignment alignment);
```

This function takes one of the VerticalAlignment enumerated values as its input. The VerticalAlignment enumeration is defined as:

```
enum VerticalAlignment
{
    VA_TOP,
    VA_CENTRE,
    VA_BOTTOM
};
```

Where:

- VA_TOP specifies that y-axis positions specify an offset for a window's top edge from the top edge of it's parent window.

- VA_CENTRE specifies that y-axis positions specify an offset for a window's centre point from the centre point of it's parent window.
- VA_BOTTOM specifies that y-axis positions specify an offset for a window's bottom edge from the bottom edge of it's parent window.

The window property to access the vertical alignment setting is:

```
“VerticalAlignment”
```

This property takes a simple string as its value, which should be one of the following options:

```
"Top"
"Centre"
"Bottom"
```

Where these setting values correspond to the similar values in the VerticalAlignment enumeration.

Horizontal Alignments

To set the horizontal alignment use the Window class member function:

```
void setHorizontalAlignment(const HorizontalAlignment alignment);
```

This function takes one of the HorizontalAlignment enumerated values as its input. The HorizontalAlignment enumeration is defined as:

```
enum HorizontalAlignment
{
    HA_LEFT,
    HA_CENTRE,
    HA_RIGHT
};
```

Where:

- HA_LEFT specifies that x-axis positions specify an offset for a window's left edge from the left edge of it's parent window.
- HA_CENTRE specifies that x-axis positions specify an offset for a window's centre point from the centre point of it's parent window.
- HA_RIGHT specifies that x-axis positions specify an offset for a window's right edge from the right edge of it's parent window.

The window property to access the horizontal alignment setting is:

```
“HorizontalAlignment”
```

This property takes a simple string as its value, which should be one of the following options:

```
"Left"
"Centre"
"Right"
```

Where these setting values correspond to the similar values in the HorizontalAlignment enumeration.

Falagard in Schemes

The CEGUI scheme system has had some extensions added to enable you to specify how the system is to load your XML skin definition files (known as 'looknfeel' files), and how these skins are to be mapped to the Falagard widget base classes to create new widget types.

The CEGUIFalagardBase module

One of the main parts of the Falagard system is the new 'look' module known as CEGUIFalagardBase (which will be named `libCEGUIFalagardBase.so` on linux style systems and `CEGUIFalagardBase.dll` on Win32 systems). This module is where actions are taken to transform skinning data loaded from XML files into rendering operations and layout adjustments required by the CEGUI base widget components.

The first important thing you need to know about this widget module is, that like the old TaharezLook and WindowsLook modules, you must specify it in one of your scheme files so that it's available to the system. This can be done with a single line of XML in a scheme file, such as:

```
<WindowSet Filename="CEGUIFalagardBase" />
```

Notice that there is no longer a need to specify the individual widget types to be loaded from the module; if you use XML as shown above, the module will register all widget types it has available.

The other important thing about the module is that for each widget base type, it defines various required "states" and other such elements. These required elements need to be defined within the widget look definitions of your looknfeel XML files; they enable the system to make use of your skin imagery and related data in a logical fashion. All of the required elements for each widget can be found in the “Falagard Base Widgets Reference”

section.

LookNFeel Elements

The new “LookNFeel” XML element for schemes is the means by which you will usually get CEGUI to load the XML 'looknfeel' files containing your widget skin definitions. The LookNFeel element should appear after any Font or Imageset elements, but before any WindowSet elements.

The following is an example of how to use the LookNFeel element:

```
<LookNFeel Filename="FunkyWidgets.looknfeel" />
```

Here we can see a single 'Filename' attribute which specifies the name the file to be loaded.

It is acceptable to specify as many LookNFeel elements as is required. This allows you to configure your XML files in the way that best suits your application. This might mean that all definitions for all widget elements will go into a single file, it might mean that you have multiple files with a single widget definition in each, or it could be any place in between the two extremes – it's up to you.

FalagardMapping Elements

The CEGUI scheme system now supports a "FalagardMapping" element. This new element creates a new window or widget type within the system. It does this by mapping a named widget 'LookNFeel' to a target base widget. The named 'LookNFeel' is what you specify in your XML looknfeel files (via the WidgetLook elements), and the available base widget types are those which get loaded from the CEGUIFalagardBase module.

A small example mapping:

```
<FalagardMapping
      WindowType="FunkyLook/Button"
      TargetType="Falagard/Button"
      LookNFeel="MyButtonSkin"
/>
```

In this example, a new widget type named "FunkyLook/Button" is being created. The new widget is based upon the "Falagard/Button" base type, and takes its skin from the loaded WidgetLook named "MyButtonSkin". Once the scheme with this mapping has been loaded, you can then use the new type within the system:

```
// Get access to the main window manager
CEGUI::WindowManager wMgr& = CEGUI::WindowManager::getSingleton();
```

```
// Create a new widget
wMgr.createWindow("FunkyLook/Button", "myFunkyButton");
```

Here we create an instance of the new widget, and name it "myFunkyButton". The widget can now be attached to other windows and generally used as you would any 'normal' widget.

Conclusion

This concludes the overview of the new parts of the CEGUI system.

You have seen how the new “Unified” co-ordinate system works, and how to make use of the new window alignment options.

You have also learned the basics of how to set up your scheme files to initialise the Falagard base module, and how to map XML defined skins to the base Falagard widgets to create new widget types.

The next section of this document will introduce the XML format and elements used in the 'looknfeel' files.

Introduction to Falagard 'looknfeel' XML

Before we get to the good stuff, I'd just like to point out that this section (or, indeed, the entire document) is not intended to teach you anything about XML in general. It is assumed the reader has some familiarity with XML and how to use it properly.

Before we begin: An empty skin

Before we can start adding widget skins, or WidgetLooks as they are known in the system, to our XML file, we need the basic file outline initialised. This is extremely trivial, and looks like this:

```
<?xml version="1.0" ?>
<Falagard>
</Falagard>
```

We will be placing our WidgetLook definitions between the <Falagard></Falagard> pair. It is possible to specify as many sub-elements as we require within these tags, so all of our definitions can go into a single file (in most cases this ends up being a very large file!)

Starting Simple: A Button

Without a doubt, the humble push button is the most common widget we're ever likely to see; without this workhorse, any UI would be virtually useless. So, this is where we will start.

To define any widget skin, you use the WidgetLook element and specify a name for the widget type that you're defining by using the *name* attribute. So we'll start off by adding the following empty WidgetLook to our initial skin file:

```
<WidgetLook name="TaharezLook/Button">
</WidgetLook>
```

As you can see from the reference for the Falagard/Button widget, we are required to specify imagery for numerous states, namely these are:

- Normal
- Hover
- Pushed
- Disabled

Since we now know what states are required for the widget, it's a good idea to add the framework for these first; this effectively makes the WidgetLook complete and usable – although obviously nothing would be drawn for it at this stage, since we have not defined any imagery. So, we add empty StateImagery elements for the required states, and we end up with this:

```
<WidgetLook name="TaharezLook/Button">
  <StateImagery name="Normal">
  </StateImagery>
  <StateImagery name="Hover">
  </StateImagery>
  <StateImagery name="Pushed">
  </StateImagery>
  <StateImagery name="Disabled">
  </StateImagery>
</WidgetLook>
```

To specify rendering to be used for a widget, we use the ImagerySection element. Each imagery section is given a name; this name is used later to 'include' the imagery section within layers defined for each of the state imagery definitions.

For our button, we will have an imagery section for each of the button states. We can add the outline of these to our existing, work-in-progress, widget-look:

```
<WidgetLook name="TaharezLook/Button">
  <ImagerySection name="normal_imagery">
  </ImagerySection>
  <ImagerySection name="hover_imagery">
  </ImagerySection>
  <ImagerySection name="pushed_imagery">
  </ImagerySection>
  <StateImagery name="Normal">
  </StateImagery>
  <StateImagery name="Hover">
  </StateImagery>
  <StateImagery name="Pushed">
  </StateImagery>
  <StateImagery name="Disabled">
  </StateImagery>
</WidgetLook>
```

Now we can start to define the ImageyComponents for each section; this will tell the system how we want our button to appear on screen.

The imagery for TaharezLook gives us three sections for each button state (except Disabled; for this we'll just re-use the 'normal_imagery' and use some different colours!). The available imagery sections give us a left end, a right end, and a middle section.

There are various ways that we can approach applying these image sections to the widget; although the intended use is to have the end pieces drawn at their 'natural' horizontal size and the middle section stretched to fill the space in between the two ends. This all sounds simple enough, although there is one issue; the actual pixel sizes of the imagery is not fixed. The TaharezLook imageset uses the auto-scaling feature, which means that the source images will have variable sizes dependant upon the display resolution. All this needs to be taken into account when specifying the imagery; this way we ensure the results will be what we expect – at all resolutions.

To specify an image to be drawn, we use the ImageryComponent element. This should be added as a sub-element of ImagerySection. So we'll start by adding an empty imagery component to the definition for 'normal_imagery':

```
...
<ImagerySection name="normal_imagery">
  <ImageryComponent>
  </ImageryComponent>
</ImagerySection>
...
```

The first thing we need to add to the ImageryComponent is an area definition telling the system where this image should be drawn:

```
<ImageryComponent>
  <Area>

  </Area>
</ImageryComponent>
```

We'll start by placing the image for the left end of the button. This is the simplest component to place, since its position is known as being (0, 0). To specify these absolute values, we use the <AbsoluteDim> element.

We start defining the required dimensions for our image area by using the <Dim> element, and using <AbsoluteDim> sub-element to indicate values to be used:

```
<ImageryComponent>
  <Area>
    <Dim type="LeftEdge">
      <AbsoluteDim value="0" />
    </Dim>
    <Dim type="TopEdge">
      <AbsoluteDim value="0" />
    </Dim>

  </Area>
</ImageryComponent>
```

We have defined the left and top edges which gives our image its position. Next we will specify dimensions to establish the area size.

We want the width of the area to come from the source image itself, to do this we use the <ImageDim> element and tell it to access the image that we will be using for this component:

```
<Dim type="Width">
  <ImageDim
    imageset="TaharezLook"
    image="ButtonLeftNormal"
    dimension="Width"
  />
</Dim>
```

This tells the system that for the width of the area being defined, use the width of the image named 'ButtonLeftNormal' from the 'TaharezLook' imageset.

The last part of our area is the height. This is another simple thing to specify, since we want the height to be the same as the full height of the widget being defined. We could use either the <UnifiedDim> element or the <WidgetDim> element for this purpose; we'll use the <UnifiedDim> here as it does not need to look up the widget by name and so is likely more economical:

```
<Dim type="Height">
  <UnifiedDim scale="1.0" type="Height" />
</Dim>
```

Here we use a scale value of 1.0 to indicate we want the full height of the widget.

Now we have completed our area definition for this first image, and it looks like this:

```
<ImageryComponent>
  <Area>
    <Dim type="LeftEdge">
      <AbsoluteDim value="0" />
    </Dim>
    <Dim type="TopEdge">
      <AbsoluteDim value="0" />
    </Dim>
    <Dim type="Width">
      <ImageDim
        imageset="TaharezLook"
        image="ButtonLeftNormal"
        dimension="Width"
      />
    </Dim>
    <Dim type="Height">
      <UnifiedDim scale="1.0" type="Height" />
    </Dim>
  </Area>
</ImageryComponent>
```

The next thing we need to do here is tell the system which image it should draw, this is done by using the <Image> element, and this should be placed immediately after the area definition:

```
...
<Image imageset="TaharezLook" image="ButtonLeftNormal" />
...
```

The final element that we need to add to this ImageryComponent definition is the <VertFormat> element. Using this we will tell the system to stretch the image vertically so that it covers the full height of our defined area:

```
...
<VertFormat type="Stretched" />
...
```

This completes the definition for the left end of the button, and the final xml for this component looks like this:

```
<ImageryComponent>
  <Area>
    <Dim type="LeftEdge">
      <AbsoluteDim value="0" />
    </Dim>
    <Dim type="TopEdge">
      <AbsoluteDim value="0" />
    </Dim>
    <Dim type="Width">
      <ImageDim
        imageset="TaharezLook"
        image="ButtonLeftNormal"
        dimension="Width"
      />
    </Dim>
    <Dim type="Height">
      <UnifiedDim scale="1.0" type="Height" />
    </Dim>
  </Area>
  <Image imageset="TaharezLook" image="ButtonLeftNormal" />
  <VertFormat type="Stretched" />
</ImageryComponent>
```

The next image we will set up is the right end. To show another approach for image placement, rather than precisely defining the area where the image will appear, here we will define the target area as covering the entire widget and use the image alignment formatting to draw the image on the right hand side of the widget.

The area definition that specifies the entire widget is something that you'll likely use a lot, and looks like this:

```
<Area>
  <Dim type="LeftEdge"><AbsoluteDim value="0" /></Dim>
  <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
  <Dim type="Width"><UnifiedDim scale="1" type="Width" /></Dim>
  <Dim type="Height"><UnifiedDim scale="1" type="Height" /></Dim>
</Area>
```

Next comes comes the image specification:

```
<Image imageset="TaharezLook" image="ButtonRightNormal" />
```

Then the vertical formatting option:

```
<VertFormat type="Stretched" />
```

Finally, we add the horizontal formatting option which tells the system to align this image on the right edge of the defined area:

```
<HorzFormat type="RightAligned" />
```

The completed definition for the right end image now looks like this:

```
<ImageryComponent>
  <Area>
    <Dim type="LeftEdge"><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
    <Dim type="Width"><UnifiedDim scale="1" type="Width" /></Dim>
    <Dim type="Height"><UnifiedDim scale="1" type="Height" /></Dim>
  </Area>
  <Image imageset="TaharezLook" image="ButtonRightNormal" />
  <VertFormat type="Stretched" />
  <HorzFormat type="RightAligned" />
</ImageryComponent>
```

The last image we need to place for the "normal_imagery" section is the middle section. Remember that we want this image to occupy the space between to two end pieces. The main part of achieving this is to correctly define the destination area for the image.

The vertical aspects of the image definition for the middle section will be the same as for the two ends, and as such these will not be discussed any further.

The first thing we need is to tell the system where the left edge of the middle section should appear. We know that the left edge of the image for the middle section needs to join to the right edge of the image for the left section. To achieve this we can make use of the ImageDim element to get the width of the left end image, and use this as the co-ordinate for the left edge of the middle section area:

```
<Area>
  <Dim type="LeftEdge">
    <ImageDim
      imageset="TaharezLook"
      image="ButtonLeftNormal"
      dimension="Width"
    />
  </Dim>
  ...
</Area>
```

Now comes the fun part. Due to the fact we want the skin to operate correctly without knowing ahead of time how large the images are, we must use mathematical calculations in order to establish the required width of the middle section. If we knew for sure the image sizes, this could all be pre-calculated and we could simply use AbsoluteDim to tell the system the width we require. Unfortunately we are not this lucky. We are lucky, however, in the fact that the system provides a means for us to specify, within the XML, what calculations should be done to arrive at the final value for a dimension. The DimOperator element is what provides this ability.

Before going further we should look at what we need to calculate. The width of the middle section is basically the width of the widget, minus the combined width of the two end sections:

$$middleWidth = widgetWidth - (leftEndWidth + rightEndWidth)$$

However, due to the fact that the area can accept either a width or right edge co-ordinate,

we can simplify this a little by specifying the right edge co-ordinate instead of the width. The right edge location for this middle image will be equal to the width of the widget, minus the width of the right end image. So the final calculation we need to do is this:

rightEdge = widgetWidth – rightEndWidth

The result from both calculations is the same, so wherever possible we will use the simpler option. To specify this calculation in XML we first start off with our widget width:

```
<Dim type="RightEdge>
  <UnifiedDim scale="1" type="Width>

  </UnifiedDim>
</Dim>
```

Since we need to perform some calculation on this, we embed a DimOperator element that specifies the mathematical operation that we need to perform:

```
<Dim type="RightEdge>
  <UnifiedDim scale="1" type="Width>
    <DimOperator op="Subtract">

    </DimOperator>
  </UnifiedDim>
</Dim>
```

To complete the dimension specification we just insert a second *Dim element to tell the system what to subtract. In this case it's the width of the image for the right end, so we will use the ImageDim element for this purpose. The final specification for this dimension looks as follows:

```
<Dim type="RightEdge>
  <UnifiedDim scale="1" type="Width>
    <DimOperator op="Subtract">
      <ImageDim
        imageset="TaharezLook"
        image="ButtonRightNormal"
        dimension="Width"
      />
    </DimOperator>
  </UnifiedDim>
</Dim>
```

It is possible to chain further mathematical operations within the dimension specification. It would have been possible to do our original width calculation using two DimOperator elements chained together, however this leads us to two small oddities of the system:

First, pairs of dimension operands are taken in *reverse* order; working from the innermost operation to the outermost operation.

And second, there is no operator precedence. All operations are performed linearly starting at the innermost pair of values and working outwards.

To explain this further, the example from the reference section will be used. Basically, in that example we want to take the height of the widget font, add four pixels and multiply the result by two. This might lead us to write the following, wrong, specification:

```
...
<FontDim type="LineSpacing">
  <DimOperator op="Add">
    <AbsoluteDim value="4">
      <DimOperator op="Multiply">
        <AbsoluteDim value="2" />
      </DimOperator>
    </AbsoluteDim>
  </DimOperator>
</FontDim>
...
```

The operation that the above would perform is as follows:

```
(4 * 2) + LineSpacing
```

Obviously this is not what we were after. We need to switch the operands around so that the *pairs* are reversed:

```
...
<AbsoluteDim value="2">
  <DimOperator op="Multiply">
    <AbsoluteDim value="4">
      <DimOperator op="Add">
        <FontDim type="LineSpacing" />
      </DimOperator>
    </AbsoluteDim>
  </DimOperator>
</AbsoluteDim>
...
```

The operations this will perform, in order, are:

```
tmp = 4 + LineSpacing
```

Followed by:

```
2 * tmp
```

Giving us what we wanted which was:

```
(2 * (4 + LineSpacing))
```

Note also lack of 'normal' operator precedence, you might have been surprised to find the operation was not:

```
((2 * 4) + LineSpacing)
```

Anyway, I digress. Lets get back to our button imagery. We now have enough information to define the middle section of the button, which looks like this:

```
<ImageryComponent>
  <Area>
    <Dim type="LeftEdge">
      <ImageDim
        imageset="TaharezLook"
        image="ButtonLeftNormal"
        dimension="Width"
      />
    </Dim>
    <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
    <Dim type="RightEdge">
      <UnifiedDim scale="1" type="Width">
        <DimOperator op="Subtract">
          <ImageDim
            imageset="TaharezLook"
            image="ButtonRightNormal"
            dimension="Width"
          />
        </DimOperator>
      </UnifiedDim>
    </Dim>
    <Dim type="Height"><UnifiedDim scale="1" type="Height" /></Dim>
  </Area>
  <Image imageset="TaharezLook" image="ButtonMiddleNormal" />
  <VertFormat type="Stretched" />
  <HorzFormat type="Stretched" />
</ImageryComponent>
```

This completes the imagery within the "normal_imagery" section. You can now add in the other two sections in the same manner, just replacing the image names used for the hover and pushed imagery as appropriate – everything else will be exactly the same as what you've done for the normal imagery.

Now we can add references to the imagery sections to the elements that define the states. The imagery for state imagery elements must be specified in 'layers'. It is possible to specify multiple imagery sections to use within each layer, and for most simple cases, you'll only need one layer.

Here we've added the imagery specification for the Normal state:

```
<StateImagery name="Normal">
  <Layer>
    <Section section="normal_imagery" />
  </Layer>
</StateImagery>
```

The Hover and Pushed states are defined in a similar fashion; just replace the "normal_imagery" section with the name of the appropriate imagery section for the state.

The Disabled state is somewhat different though; we do not have any specific imagery for this state, so instead we will re-use the "normal_imagery" but specify some colours that will

be applied to make the button appear darker. This is done by embedding a Colours element within the Section element, as demonstrated here:

```
<StateImagery name="Disabled">
  <Layer>
    <Section section="normal_imagery">
      <Colours
        topLeft="FF7F7F7F"
        topRight="FF7F7F7F"
        bottomLeft="FF7F7F7F"
        bottomRight="FF7F7F7F"
    </Section>
  </Layer>
</StateImagery>
```

Now we have a nice button with imagery defined for all the required states. There's just one thing missing – we need to put some label text on the button.

To specify text, you use the TextComponent element, which goes in an ImagerySection the same as the ImageryComponent elements do. We could have put a TextComponent in each of the imagery sections we defined to display the label, however this is wasteful repetition. A better approach is to define a imagery section what contains the label by itself, then we can re-use that for all of the states.

So, start by defining the containing ImagerySection:

```
...
<ImagerySection name="label">
  <TextComponent>

  </TextComponent>
</ImagerySection>
...
```

The definition of the TextComponent is extremely similar to that of ImageryComponent. We specify an area for the text and the formatting that we require. We can also optionally specify a Text element which is used to explicitly set the font and / or text string to be drawn. Without these explicit settings, these items will be obtained from the base widget itself.

We want our label to be centred within the entire area of the widget, so we need to use the area that defines the entire widget (this was shown above, so will not be repeated here).

We also need to set the formatting options for the text. For the vertical formatting we will use:

```
<VertFormat type="CentreAligned" />
```

And for the horizontal formatting:

```
<HorzFormat type="WordWrapCentreAligned" />
```

The final definition for our label imagery section looks like this:

```
<ImagerySection name="label">
  <TextComponent>
    <Area>
      <Dim type="LeftEdge"><AbsoluteDim value="0" /></Dim>
      <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
      <Dim type="Width"><UnifiedDim scale="1" type="Width" /></Dim>
      <Dim type="Height"><UnifiedDim scale="1" type="Height" /></Dim>
    </Area>
    <VertFormat type="CentreAligned" />
    <HorzFormat type="WordWrapCentreAligned" />
  </TextComponent>
</ImagerySection>
```

Now all that is left is to add this to the layer specification for the state imagery. Normal state now looks like this (with Hover and Pushed being very similar):

```
<StateImagery name="Normal">
  <Layer>
    <Section section="normal_imagery" />
    <Section section="label" />
  </Layer>
</StateImagery>
```

And for Disabled we again specify some additional colours:

```
<StateImagery name="Disabled">
  <Layer>
    <Section section="normal_imagery">
      <Colours
        topLeft="FF7F7F7F"
        topRight="FF7F7F7F"
        bottomLeft="FF7F7F7F"
        bottomRight="FF7F7F7F"
      />
    </Section>
    <Section section="label">
      <Colours
        topLeft="FF7F7F7F"
        topRight="FF7F7F7F"
        bottomLeft="FF7F7F7F"
        bottomRight="FF7F7F7F"
      />
    </Section>
  </Layer>
</StateImagery>
```

This concludes the introduction tutorial. For full examples of this, and all the other WidgetLook specifications, see the example 'looknfeel' files in the cegui_mk2 distribution, in the directory:

cegui_mk2/Samples/datafiles/looknfeel/

Part Two

Reference Information

Falagard XML Reference

The following pages contain reference material for the XML elements defined for the Falagard skin definition files.

The reference for each element is arranged into sections, as described below:

Purpose:

This section describes what the elements general purpose is within the specifications.

Attributes:

This section describes available attributes for the elements, and whether they are required or optional.

Usage:

Describes where the element may appear, whether the element may have sub-elements, and other important usage information.

Examples:

For many elements, this section will contain brief examples showing the element used in context.

<AbsoluteDim> Element

Purpose:

The <AbsoluteDim> element is used to define a component dimension for an area rectangle. <AbsoluteDim> is used to specify absolute pixel values for a dimension.

Attributes:

- value – specifies the a number of pixels. Required attribute.

Usage:

- The <AbsoluteDim> element may contain a single <DimOperator> element in order to form a dimension calculation.
- The <AbsoluteDim> element can appear as a sub-element in <Dim> to form a dimension specification for an area.
- The <AbsoluteDim> element can appear as a sub-element of <DimOperator> to specify the second operand for a dimension calculation.

Examples:

The following shows <AbsoluteDim> used to define an area rectangle. In the example, all four component dimensions of the area rectangle are specified using <AbsoluteDim>:

```
<Area>
  <Dim type="LeftEdge" >
    <AbsoluteDim value="10" />
  </Dim>
  <Dim type="TopEdge" >
    <AbsoluteDim value="50" />
  </Dim>
  <Dim type="Width" >
    <AbsoluteDim value="290" />
  </Dim>
  <Dim type="Height" >
    <AbsoluteDim value="250" />
  </Dim>
</Area>
```

The following shows <AbsoluteDim> in use as part of a dimension calculation sequence. In the example the left edge is being set to the width of the child widget 'myWidget' minus two pixels:

```
<Area>
  <Dim type="LeftEdge" >
    <WidgetDim widget="myWidget" dimension="Width" >
      <DimOperator op="Subtract" >
        <AbsoluteDim value="2" />
      </DimOperator>
    </WidgetDim>
  </Dim>
  ...
</Area>
```

Finally, we see <AbsoluteDim> as the starting dimension for a dimension calculation sequence. In the example, we are adding the value of some window property to the starting absolute value of six:

```
<Area>
  ...
  <Dim type="Height" >
    <AbsoluteDim value="6">
      <DimOperator op="Add" >
        <PropertyDim name="someHeightProperty" />
      </DimOperator>
    </AbsoluteDim>
  </Dim>
</Area>
```

<Area> Element

Purpose:

The <Area> element is a simple container element for the <Dim> dimension elements, or a single <AreaProperty> element, in order to form a rectangular area. <Area> is generally used to define target regions which are to be used for rendering imagery, text, to place a component child widget, or to form 'named' areas required by the base widget.

Attributes:

<Area> has no attributes.

Usage:

- The <Area> element must contain either:
  - A single <AreaProperty> element that describes a URect type property where the final area can be obtained.
  - Four <Dim> elements:
    - One <Dim> element must define the left edge or x position.
    - One <Dim> element must define the top edge or y position.
    - One <Dim> element must define either the right edge or width.
    - One <Dim> element must define either the bottom edge or height.

The <Area> element may appear in any of the following elements:

- <Child> to define the target area to be occupied by a child widget.
- <ImageryComponent> to define the target rendering area of an image.
- <NamedArea> to define an area which can be retrieved by name.
- <TextComponent> to define the target rendering area of some text.
- <FrameComponent> to define the target rendering area for a frame.

Examples:

In this example we can see a named area being defined:

```
<NamedArea name="exampleArea" >
    <Area>
        <Dim type="LeftEdge"><AbsoluteDim value="0" /></Dim>
        <Dim type="TopEdge"><AbsoluteDim value="0" /></Dim>
        <Dim type="Width"><UnifiedDim scale="1.0" /></Dim>
        <Dim type="Height"><UnifiedDim scale="1.0" /></Dim>
    </Area>
</NamedArea>
```

<AreaProperty> Element

Purpose:

The <AreaProperty> element is intended to allow the system to access a property on the target window to obtain the final target area of a component being defined.

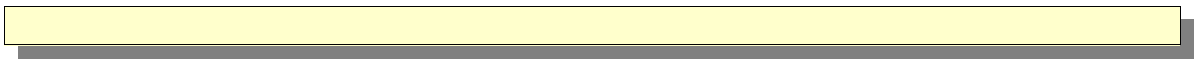
Attributes:

- name – specifies the name of the property to access. The named property must access a URect value. Required attribute.

Usage:

- The <AreaProperty> element may not contain sub-elements.
- The <AreaProperty> element may appear as a sub-element only within the main <Area> element.

Examples:



<Child> Element

Purpose:

The <Child> element defines a component widget that will be created and added to each instance of any window using the <WidgetLook> being defined. Some base widgets have requirements for <Child> element definition that must be provided.

Attributes:

- type – specifies the widget type to create. Required attribute.
- nameSuffix – specifies a suffix which will be used when naming the widget. The final name of the child widget will be that of the parent with this suffix appended. Required attribute.
- look – specifies the name of a widget look to apply to the child widget. You should only use this if 'type' specifies a Falagard base widget type. Optional attribute.

Usage:

Note: the sub-elements should appear in the order that they are defined here.

- The <Child> element must contain an <Area> element that defines the location of the child widget in relation to the component being defined.
- You may optionally specify a single <VertAlignment> element to set the vertical alignment for the child.
- You may optionally specify a single <HorzAlignment> element to set the horizontal alignment for the child.
- You may specify any number of <Property> elements to set default values for any property supported by the widget type being used for the child.
- The <Child> element may only appear within the <WidgetLook> element.

Examples:

In this example, taken from TaharezLookSkin.looknfeel, we see how the title bar child widget required by the frame window type is defined:

```
<WidgetLook name="TaharezLook/FrameWindow">
  ...
  <Child type="TaharezLook/Titlebar" nameSuffix="__auto_titlebar__">
    <Area>
      <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
      <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
      <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
      <Dim type="Height" >
        <FontDim type="LineSpacing">
          <DimOperator op="Multiply">
            <AbsoluteDim value="1.5" />
          </DimOperator>
        </FontDim>
      </Dim>
    </Area>
    <Property name="AlwaysOnTop" value="False" />
  </Child>
```

```
  ...
</WidgetLook>
```

<ColourProperty> Element

Purpose:

The <ColourProperty> element is intended to allow the system to access a property on the target window to obtain colour information to be used when drawing some part of the component being defined.

Attributes:

- name – specifies the name of the property to access. The named property must access a single colour value. Required attribute.

Usage:

- The <ColourProperty> element may not contain sub-elements.

The <ColourProperty> element may appear as a sub-element within any of the following elements:

- <ImageryComponent> to specify a modulating colour to be applied when rendering the image.
- <ImagerySection> to specify a modulating colour to be applied to all imagery components within the imagery section as it is rendered.
- <Section> to specify a modulating colour to be applied to all imagery in the named section as it is rendered.
- <TextComponent> to specify a colour to use when rendering the text component.
- <FrameComponent> to specify a colour to use when rendering the text frame.

Examples:

The following example, listing imagery for a button in the “Normal” state, shows the <ColourProperty> element in use to specify a property where colours to be used when rendering the ImagerySection named “label” can be found:

```
<StateImagery name="Normal">
  <Layer>
    <Section section="normal" />
    <Section section="label">
      <ColourProperty name="NormalTextColour" />
    </Section>
  </Layer>
</StateImagery>
```

<ColourRectProperty> Element

Purpose:

The <ColourRectProperty > element is intended to allow the system to access a property on the target window to obtain colour information to be used when drawing some part of the component being defined.

Attributes:

- name – specifies the name of the property to access. The named property must access a ColourRect value. Required attribute.

Usage:

- The <ColourRectProperty> element may not contain sub-elements.

The <ColourRectProperty> element may appear as a sub-element within any of the following elements:

- <ImageryComponent> to specify a modulating ColourRect to be applied when rendering the image.
- <ImagerySection> to specify a modulating ColourRect to be applied to all imagery components within the imagery section as it is rendered.
- <Section> to specify a modulating ColourRect to be applied to all imagery in the named section as it is rendered.
- <TextComponent> to specify a ColourRect to use when rendering the text component.
- <FrameComponent> to specify a colour to use when rendering the text frame.

Examples:

```
...
<StateImagery name="SpecialState">
  <Layer>
    <Section section="special_main">
      <ColourRectProperty name="SpecialColours" />
    </Section>
  </Layer>
</StateImagery>
...
```

<Colours> Element

Purpose:

The <Colours> element is used to explicitly specify values for a ColourRect that should be used when rendering some part of the component being defined.

Attributes:

- topLeft – specifies a hex colour value, of the form "AARRGGBB", to be used for the top-left corner of the ColourRect. Required attribute.
- topRight – specifies a hex colour value, of the form "AARRGGBB", to be used for the top-right corner of the ColourRect. Required attribute.
- bottomLeft – specifies a hex colour value, of the form "AARRGGBB", to be used for the bottom-left corner of the ColourRect. Required attribute.
- bottomRight – specifies a hex colour value of the form "AARRGGBB", to be used for the bottom-right corner of the ColourRect. Required attribute.

Usage:

- The <Colours> element may not contain sub-elements.

The <Colours> element may appear as a sub-element within any of the following elements:

- <ImageryComponent> to specify a modulating ColourRect to be applied when rendering the image.
- <ImagerySection> to specify a modulating ColourRect to be applied to all imagery components within the imagery section as it is rendered.
- <Section> to specify a modulating ColourRect to be applied to all imagery in the named section as it is rendered.
- <TextComponent> to specify a ColourRect to use when rendering the text component.
- <FrameComponent> to specify a colour to use when rendering the text frame.

Examples:

In this example, we see the <Colours> element used to specify the value 0xFFFFFF00 as the colour for all four corners of the colour rect to be used when rendering the image being defined:

```
...
<ImageryComponent>
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><AbsoluteDim value="12" /></Dim>
    <Dim type="Height" ><AbsoluteDim value="24" /></Dim>
  </Area>
  <Image imageset="newImageset" image="FunkyComponent" />
  <Colours
    topLeft="FFFFFF00"
    topRight="FFFFFF00"
    bottomLeft="FFFFFF00"
    bottomRight="FFFFFF00"
  />
  <VertFormat type="Stretched" />
  <HorzFormat type="Stretched" />
</ImageryComponent>
...
```

<Dim> Element

Purpose:

The <Dim> element is intended as a container element for a single dimension of an area rectangle.

Attributes:

- type – specifies what the dimension being defined represents. This attribute should be set to one of the values defined for the DimensionType enumeration (see below). Required attribute.

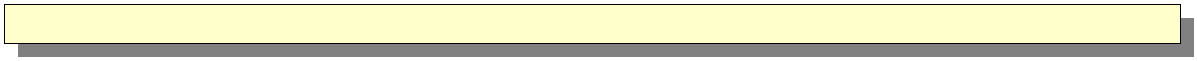
Usage:

- The <Dim> element may only appear within the <Area> element.

The <Dim> element may contain any of the following specialised dimension elements:

- <AbsoluteDim>
- <FontDim>
- <ImageDim>
- <PropertyDim>
- <UnifiedDim>
- <WidgetDim>

Examples:



<DimOperator> Element

Purpose:

The <DimOperator> element allows you to combine two of the specialised dimension specifier elements via a simple mathematical operator. Since the dimension used as the second operand may also contain a <DimOperator> it is possible to create quite complex operations.

Of important note is the fact that in a large chain of operations, the calculations are done in *reverse* order. Also, there is no operator precedence as such, all operations are applied linearly.

Attributes:

- op – specifies one of the vales from the DimensionOperator enumeration indicating the mathematical operation to be performed. Required attribute.

Usage:

A single <DimOperator> element may appear as a sub-element within any of the following specialised dimension elements:

- <AbsoluteDim>
- <FontDim>
- <ImageDim>
- <PropertyDim>
- <UnifiedDim>
- <WidgetDim>

The <DimOperator> element may contain any of the following specialised dimension elements:

- <AbsoluteDim>
- <FontDim>
- <ImageDim>
- <PropertyDim>
- <UnifiedDim>
- <WidgetDim>

Examples:

The following multiplies two simple AbsoluteDim dimensions:

```
...
<AbsoluteDim value="10">
  <DimOperator op="Multiply">
    <AbsoluteDim value="4" />
  </DimOperator>
</AbsoluteDim>
...
```

The next example takes the height of the font used for the target window, adds four pixels and multiplies the result by two.

Note the effectively reversed order and lack of 'normal' operator precedence, the operation performed will be:

*(2 * (4 + LineSpacing))*

and not:

*((2 * 4) + LineSpacing)*

```
...
<AbsoluteDim value="2">
  <DimOperator op="Multiply">
    <AbsoluteDim value="4">
      <DimOperator op="Add">
        <FontDim type="LineSpacing" />
      </DimOperator>
    </AbsoluteDim>
  </DimOperator>
</AbsoluteDim>
...
```

<Falagard> Element

Purpose:

The <Falagard> element is the root element in Falagard XML skin definition files. The element serves mainly as a container for <WidgetLook> elements

Attributes:

- Element <Falagard> has no attributes.

Usage:

- The <Falagard> element is the root element for Falagard skin files.
- The <Falagard> element may contain any number of <WidgetLook> elements.
- No element may contain <Falagard> elements as a sub-element.

Examples:

Here we just see the general structure of a Falagard XML file, notice that the <Falagard> element just serves as a container for multiple <WidgetLook> elements:

```
<?xml version="1.0" ?>
<Falagard>
  <WidgetLook name="TaharezLook/Button">
    ...
  </WidgetLook>
  <WidgetLook ... >
    ...
  </WidgetLook>
  ...
</Falagard>
```

<FontDim> Element

Purpose:

The <FontDim> element is used to take some measurement of a Font, and use it as a dimension component of an area rectangle.

Attributes:

- widget – specifies the name suffix of a child window to access when automatically obtaining the font or text string to be used when calculating the dimension's value. The final name used to access the widget will be that of the target window with this suffix appended. If this suffix is not specified, the target window itself is used. Optional attribute.
- type – specifies the type of font metric / measurement to use for this dimension. This should be set to one of the values from the FontMetricType enumeration. Required attribute.
- font – specifies the name of a font. If no font is given, the font will be taken from the target window at the time the dimension's value is taken. Optional attribute.
- string – For horizontal extents measurement, specifies the string to be measured. If no explicit string is given, the window text for the target window at the time the dimension's value is taken will be used instead. Optional attribute.
- padding – an absolute pixel 'padding' value to be added to the font metric value. Optional attribute.

Usage:

- The <FontDim> element may contain a single <DimOperator> element in order to form a dimension calculation.
- The <FontDim> element can appear as a sub-element in <Dim> to form a dimension specification for an area.
- The <FontDim> element can appear as a sub-element of <DimOperator> to specify the second operand for a dimension calculation.

Examples:

This first example just gets the line spacing for the window's current font:

```
<Dim type="Height">
  <FontDim type="LineSpacing" />
</Dim>
```

Now we take an extents measurement of the windows current text, using a specified font, and pad the result by ten pixels:

```
<Dim type="Width">
  <FontDim type="HorzExtent" font="Roman-14" padding="10" />
</Dim>
```

<FrameComponent> Element

Purpose:

The <FrameComponent> element is used to define an imagery frame using a maximum of eight images for the corners and edges, and a single, formatted, image for the background. Any of the images may be omitted if not required.

Attributes:

- No attributes are currently defined for the <FrameComponent> element.

Usage:

Note: the sub-elements should appear in the order that they are defined here.

- <Area> defining the target area for this frame.
- Up to nine <Image> elements specifying the images to be drawn and in what positions.
- Optionally specifying the colours for the entire frame, one of the colour elements:
  - <Colours>
  - <ColourProperty>
  - <ColourRectProperty>
- Optionally, to specify the vertical formatting to use for the frame background, either of:
  - <VertFormat>
  - <VertFormatProperty>
- Optionally, to specify the horizontal formatting to use for the frame background, either of:
  - <HorzFormat>
  - <HorzFormatProperty>

- The <FrameComponent> element may only appear as a sub-element of the element <ImagerySection>.

Examples:

The following defines a full frame and background. It is taken from the TaharezLook skin specification for the Listbox widget:

```
<FrameComponent>
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
    <Dim type="Height" ><UnifiedDim scale="1" type="Height" /></Dim>
  </Area>
  <Image type="TopLeftCorner"
    imageset="TaharezLook" image="ListboxTopLeft"
  />
  <Image type="TopRightCorner"
    imageset="TaharezLook" image="ListboxTopRight"
  />
  <Image type="BottomLeftCorner"
    imageset="TaharezLook" image="ListboxBottomLeft"
  />
  <Image type="BottomRightCorner"
    imageset="TaharezLook" image="ListboxBottomRight"
  />
  <Image type="LeftEdge"
    imageset="TaharezLook" image="ListboxLeft"
  />
  <Image type="RightEdge"
    imageset="TaharezLook" image="ListboxRight"
  />
  <Image type="TopEdge"
    imageset="TaharezLook" image="ListboxTop"
  />
  <Image type="BottomEdge"
    imageset="TaharezLook" image="ListboxBottom"
  />
  <Image type="Background"
    imageset="TaharezLook" image="ListboxBackdrop"
  />
</FrameComponent>
```

<HorzAlignment> Element

Purpose:

The <HorzAlignment> element is used to specify the horizontal alignment option required for a child window element.

Attributes:

- type – specifies one of the values from the HorizontalAlignment enumeration indicating the desired horizontal alignment.

Usage:

- The <HorzAlignment> element may only appear as a sub-element of the <Child> element.
- The <HorzAlignment> element may not contain any sub-elements.

Examples:

This example defines a scrollbar type child widget. We have used the <HorzAlignment> element to specify that the scrollbar appear on the far right edge of the component being defined:

```
...
<Child  type="MyLook/VertScrollbar" nameSuffix="__auto_vscrollbar__">
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><AbsoluteDim value="15" /></Dim>
    <Dim type="Height" ><UnifiedDim scale="1" type="Height" /></Dim>
  </Area>
  <HorzAlignment type="RightAligned" />
</Child>
...
```

<HorzFormat> Element

Purpose:

The <HorzFormat> element is used to specify the required horizontal formatting for an image, frame, or text component.

Attributes:

type – specifies the required horizontal formatting option. For use in ImageryComponents or FrameComponents, this will be one of the values from the HorizontalFormat enumeration. For use in TextComponents, this will one of the values form the HorizontalTextFormat enumeration.

Usage:

- The <HorzFormat> element may only appear as a sub-element of the <ImageryComponent>, <FrameComponent>, or <TextComponent> elements.
- The <HorzFormat> element may not contain any sub-elements.

Examples:

This first example shows an ImageryComponent definition. We use <HorzFormat> to specify that we want the image stretched to cover the entire width of the designated target area:

```
...
<ImageryComponent>
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><AbsoluteDim value="25" /></Dim>
    <Dim type="Height" ><AbsoluteDim value="25" /></Dim>
  </Area>
  <Image imageset="myImageset" image="coolImage" />
  <VertFormat type="Stretched" />
  <HorzFormat type="Stretched" />
</ImageryComponent>
...
```

This second example is for a TextComponent. You can see <HorzFormat> used here to specify that we want the text centred within the target area, and word-wrapped where required:

```
<TextComponent>
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="RightEdge" ><UnifiedDim scale="1" type="Width" /></Dim>
    <Dim type="Height" ><UnifiedDim scale="1" type="Height" /></Dim>
  </Area>
  <HorzFormat type="WordWrapLeftAligned" />
</TextComponent>
```

<HorzFormatProperty> Element

Purpose:

The <HorzFormatProperty> element is intended to allow the system to access a property on the target window to obtain the horizontal formatting to be used when drawing the component being defined.

Attributes:

- name – specifies the name of the property to access. The named property must access a string value that will be set to one of the enumeration values appropriate for the component being defined (so HorizontalTextFormat for TextComponent, and HorizontalFormat for FrameComponent and ImageryComponent). Required attribute.

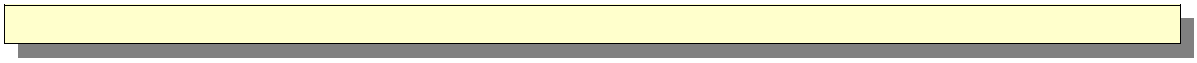
Usage:

- The <HorzFormatProperty> element may not contain sub-elements.

The <HorzFormatProperty> element may appear as a sub-element within any of the following elements:

- <ImageryComponent> to specify a horizontal formatting to be used the the image.
- <FrameComponent> to specify a horizontal formatting to be used for the frame background.
- <TextComponent> to specify a horizontal formatting to be used for the text.

Examples:



<Image> Element

Purpose:

The <Image> element is used to specify an Imageset and Image pair, and for FrameComponent images, how the image is to be used.

Attributes:

- imageset – specifies the name of an Imageset which contains the image to be used. Required attribute.
- image – specifies the name of the image from the specified Imageset to be used. Required attribute.
- type – ***Only for FrameComponent***. Specifies the part of the frame that this image is to be used for. One of the values from the FrameImageComponent enumeration. Required attribute.

Usage:

- The <Image> element may only appear as a sub-element of the <ImageryComponent> or <FrameComponent> elements.
- The <Image> element may not contain any sub-elements.

Examples:

Here you can see the <Image> element used to specify the image to render for an ImageryComponent being defined:

```
...
<ImageryComponent>
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><AbsoluteDim value="15" /></Dim>
    <Dim type="Height" ><UnifiedDim scale="1.0" type="Height" /></Dim>
  </Area>
  <Image imageset="FunkyLook" image="ButtonIcon" />
  <VertFormat type="CentreAligned" />
  <HorzFormat type="CentreAligned" />
</ImageryComponent>
...
```

<ImageDim> Element

Purpose:

The <ImageDim> element is used to define a component dimension for an area rectangle. <ImageDim> is used to specify some dimension of an image for use as an area dimension.

Attributes:

- imageset – specifies the name of an Imageset which contains the image to be used. Required attribute.
- image – specifies the name of the image from the specified Imageset to be used. Required attribute.
- dimension – specifies the image dimension to be used. This should be set to one of the values defined in the DimensionType enumeration. Required attribute.

Usage:

- The <ImageDim> element may contain a single <DimOperator> element in order to form a dimension calculation.
- The <ImageDim> element can appear as a sub-element in <Dim> to form a dimension specification for an area.
- The <ImageDim> element can appear as a sub-element of <DimOperator> to specify the second operand for a dimension calculation.

Examples:

This example shows a dimension that uses <ImageDim> to fetch the width of a specified image for use as the dimensions value:

```
...
<Area>
  <Dim type="LeftEdge">
    <ImageDim imageset="myImages" image="leftImage" dimension="width" />
  </Dim>
  ...
</Area>
...
```

<ImageryComponent> Element

Purpose:

The <ImageryComponent> element defines a single image to be drawn within a given ImagerySection. The ImageryComponent contains all information about which image is to be drawn, where it should be drawn, which colours are to be used and how the image should be formatted.

Attributes:

- No attributes are defined for the <ImageryComponent> element.

Usage:

Note: the sub-elements should appear in the order that they are defined here.

- <Area> defining the target area for this image.
- Either one of:
  - <Image> element specifying the image to be drawn.
  - <ImageProperty> element specifying a property defining the image to be drawn.
- Optionally specifying the colours for this single image, one of the colour elements:
  - <Colours>
  - <ColourProperty>
  - <ColourRectProperty>
- Optionally, to specify the vertical formatting to use, either of:
  - <VertFormat>
  - <VertFormatProperty>
- Optionally, to specify the horizontal formatting to use, either of:
  - <HorzFormat>
  - <HorzFormatProperty>

- The <ImageryComponent> element may only appear as a sub-element of the element <ImagerySection>.

Examples:

The following was taken from TaharezLookSkin.looknfeel and shows a full ImageryComponent definition:

```
<ImageryComponent>
  <Area>
    <Dim type="LeftEdge" ><UnifiedDim scale="0" type="LeftEdge" /></Dim>
    <Dim type="TopEdge" ><UnifiedDim scale="0.2" type="TopEdge" /></Dim>
    <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
    <Dim type="Height" ><UnifiedDim scale="0.3" type="Height" /></Dim>
  </Area>
  <Image imageset="TaharezLook" image="TextSelectionBrush" />
  <Colours
    topLeft="FFFFFF00"
    topRight="FFFFFF00"
    bottomLeft="FFFFFF00"
    bottomRight="FFFFFF00"
  />
  <VertFormat type="Tiled" />
  <HorzFormat type="Stretched" />
</ImageryComponent>
```

<ImageProperty> Element

Purpose:

The <ImageProperty> element is intended to allow the system to access a property on the target window to obtain the final image to be used when rendering the ImageryComponent being defined.

Attributes:

- name – specifies the name of the property to access. The named property must access a imageset & image value pair of the form:

  "set:<imageset_name> image:<image_name>"

  Required attribute.

Usage:

- The <ImageProperty> element may not contain sub-elements.
- The <ImageProperty> element may appear as a sub-element only within the <ImageryComponent> elements.

Examples:

<ImagerySection> Element

Purpose:

The <ImagerySection> element is used to group multiple <ImageryComponent> and <TextComponent> definitions into named sections which can then be specified for use as imagery in state definitions.

Attributes:

- name – specifies the name to be given to this ImagerySection. Names are per-WidgetLook, and specifying the same name more than once will replace the previous definition. Required attribute.

Usage:

Note: the sub-elements should appear in the order that they are defined here.

- To optionally specify colours to be modulated with the individual component's colours, the <ImagerySection> may contain one of the colour definition elements:
  - <Colours>
  - <ColourProperty>
  - <ColourRectProperty>
- Any number of <FrameComponent> elements may then follow.
- Followed by any number of <ImageryComponent> elements.
- Finally, any number of <TextComponent> elements may be given
- The <ImagerySection> element may only appear as a sub-element of the <WidgetLook> element.

Examples:

```
<ImagerySection name="example">
  <ImageryComponent>
    <Area>
      <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
      <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
      <Dim type="Width" ><AbsoluteDim value="15" /></Dim>
      <Dim type="Height" ><UnifiedDim scale="1.0" type="Height" /></Dim>
    </Area>
    <Image imageset="sillyImages" image="anotherImage" />
    <VertFormat type="Stretched" />
    <HorzFormat type="Stretched" />
  </ImageryComponent>
  <TextComponent>
    <Area>
      <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
      <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
      <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
      <Dim type="Height" ><UnifiedDim scale="1" type="Height" /></Dim>
    </Area>
  </TextComponent>
</ImagerySection>
```

<Layer> Element

Purpose:

The <Layer> element is used to define layers of imagery within the definition of a StateImagery section.

Attributes:

- priority – specifies the priority for the layer. Higher priorities appear in front of lower priorities. Default priority is 0. Optional attribute.

Usage:

- The <Layer> element may only appear as a sub-element of the <StateImagery> element.
- The <Layer> element may contain any number of <Section> sub-elements.

Examples:

Here we see a single layer with multiple sections included. This example was taken from the TaharezLook skin XML file (ListHeaderSegment widget):

```
<StateImagery name="Normal">
  <Layer>
    <Section section="segment_normal" />
    <Section section="splitter_normal" />
    <Section section="label" />
  </Layer>
</StateImagery>
```

<NamedArea> Element

Purpose:

Defines an area that can be accessed via it's name. Generally this this used by base widgets to obtain skin supplied areas for use in rendering or other widget specific operations.

Attributes:

- name – specifies a name for the area being defined. Required attribute.

Usage:

- The <NamedArea> element must contain only an <Area> sub-element defining the area rectangle for the named area.
- The <NamedArea> element may only appear as a sub-element within <WidgetLook> elements.

Examples:

This example defines a named area called “TextArea”. It is defined as being an area seven pixels inside the total area of the widget being defined:

```
<NamedArea name="TextArea">
  <Area>
    <Dim type="LeftEdge" >
      <AbsoluteDim value="7" />
    </Dim>
    <Dim type="TopEdge" >
      <AbsoluteDim value="7" />
    </Dim>
    <Dim type="RightEdge" >
      <UnifiedDim scale="1.0" offset="-7" type="RightEdge" />
    </Dim>
    <Dim type="BottomEdge" >
      <UnifiedDim scale="1.0" offset="-7" type="BottomEdge" />
    </Dim>
  </Area>
</NamedArea>
```

<Property> Element

Purpose:

The <Property> element is used to initialise a property on a window or widget being defined.

Attributes:

- name – specifies the name of the property to be initialised. Required attribute.
- value – specifies the value string to be used when initialising the property. Required attribute.

Usage:

- The <Property> element may not contain any sub-elements.
- The <Property> element may appear as a sub-element in <WidgetLook> elements to define property initialisers for the type being defined.
- The <Property> element may appear as a sub-element in <Child> elements to define property initialisers for the child widget being defined.

Examples:

In this extract from the definition for TaharezLook/Titlebar, we can see the <Property> element used to set the “CaptionColour” property; this establishes a default for all instances of this widget:

```
<WidgetLook name="TaharezLook/Titlebar">
  <Property name="CaptionColour" value="FFFFFFFF" />
  <ImagerySection name="main">
    <ImageryComponent>
      <Area>
        <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
        ...
      </Area>
    ...
    </ImageryComponent>
  </ImagerySection>
  ...
</WidgetLook>
```

<PropertyDefinition> Element

Purpose:

The <PropertyDefinition> element creates a new named property for the widget being defined. The defined property may be accessed via any means that a 'normal' property may.

Attributes:

- name – specifies the name to use for the new property. Required attribute.
- initialValue – specifies the initial value to be assigned to the property. Optional attribute.
- type – specifies the data type of the property. This should be one of the values defined for the PropertyType enumeration. Optional attribute.
- redrawOnWrite – boolean setting specifies whether writing a new value to this property should cause the widget being defined to redraw itself. Optional attribute.
- layoutOnWrite – boolean setting specifies whether writing a new value to this property should cause the widget being defined to re-layout it's defined child widgets. Optional attribute.

Usage:

- The <PropertyDefinition> element may not contain sub-elements.
- The <PropertyDefinition> element must appear as a sub-element within <WidgetLook> elements.

Examples:

In this example, within the WidgetLook we create a new property named “ScrollbarWidth”. We then use this property to control the width of a component child widget. This effectively gives the user control over the width of the child scrollbar via the property:

```
<WidgetLook name=”PropertyDefExample”>
  <PropertyDefinition
    name=”ScrollbarWidth”
    initialValue=”12”
    layoutOnWrite=”true”
  />
  ...
  <Child  type="MyVertScrollbar" nameSuffix="__auto_vscrollbar__">
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><PropertyDim name="ScrollbarWidth" /></Dim>
    <Dim type="Height" ><UnifiedDim scale="1" type="Height" /></Dim>
  </Area>
  <HorzAlignment type="RightAligned" />
</Child>
...
</WidgetLook>
```

<PropertyDim> Element

Purpose:

The <PropertyDim> element is used to define a component dimension for an area rectangle. <PropertyDim> is used to specify a floating point value, accessed via a window property, for use as an area dimension.

Attributes:

- widget – specifies the name suffix of a child window to access the property for.
  The final name used to access the widget will be that of the target window with this suffix appended. If this suffix is not specified, the target window itself is used. Optional attribute.
- name – specifies the name of the property that will provide the value for this dimension. The property named should access a simple numerical value.

Usage:

- The <PropertyDim> element may contain a single <DimOperator> element in order to form a dimension calculation.
- The <PropertyDim> element can appear as a sub-element in <Dim> to form a dimension specification for an area.
- The <PropertyDim> element can appear as a sub-element of <DimOperator> to specify the second operand for a dimension calculation.

Examples:

This example shows a dimension that uses <PropertyDim> to fetch a property value to use as the dimensions value. We are accessing the 'AbsoluteWidth' property from an attached widget with the name suffix '__auto_button__':

```
...
<Area>
  <Dim type="LeftEdge">
    <PropertyDim widget="__auto_button__" name="AbsoluteWidth" />
  </Dim>
  ...
</Area>
...
```

<Section> Element

Purpose:

The <Section> element is used to name an ImagerySection to be included for rendering within a StateImagery Layer definition.

Attributes:

- look – specifies the name of a WidgetLook that contains the ImagerySection to be referenced. If this is omitted, the WidgetLook currently being defined is used. Optional attribute.
- section – specifies the name of an ImagerySection from the chosen WidgetLook to be referenced. Required attribute.

Usage:

- The <Section> element may only appear as a sub-element within the <Layer> element.
- The <Section> element may specify colours to be modulated with any current colours used for each component within the named ImagerySection, by optionally specifying one of the colour elements as a sub-element:
  - <Colours>
  - <ColourProperty>
  - <ColourRectProperty>

Examples:

Here we see a state definition from a button widget. The state specifies to use the “normal” imagery section, and also the “label” imagery section. Colours for “label” will be modulated with the colour obtained from the “NormalTextColour” property of the target window:

```
...
<StateImagery name="Normal">
  <Layer>
    <Section section="normal" />
    <Section section="label">
      <ColourProperty name="NormalTextColour" />
    </Section>
  </Layer>
</StateImagery>
...
```

<StateImagery> Element

Purpose:

The <StateImagery> element defines imagery to be used when rendering a named state. The base widget type intended as a target for the WidgetLook being defined will specify which states are required to be defined.

Attributes:

- name – specifies the name of the state being defined. Required attribute.
- clipped – boolean setting that states whether imagery defined within this state should be clipped to the target window's defined area. If this is specified and set to false, the state imagery will only be clipped to the display area. Optional attribute.

Usage:

- The <StateImagery>element may contain any number of <Layer> sub-elements.
- The <StateImagery> element may only appear as a sub-element of the <WidgetLook> element.

Examples:

The following is an extract of the MenuItem definition from TaharezLookSkin.looknfeel. The except defines some of the states required for that widget. Note that, although not shown here, a required state can be empty if no rendering is needed for that state:

```
...
<StateImagery name="EnabledNormal">
  <Layer>
    <Section section="label" />
  </Layer>
</StateImagery>
<StateImagery name="EnabledHover">
  <Layer>
    <Section section="frame" />
    <Section section="label" />
  </Layer>
</StateImagery>
<StateImagery name="EnabledPushed">
  <Layer>
    <Section section="frame" />
    <Section section="label" />
  </Layer>
</StateImagery>
...
```

<Text> Element

Purpose:

The <Text> element is used to define font and text string information within a TextComponent.

Attributes:

- font – specifies the name of a font to use for this text. If this is omitted, the current font of the target window will be used instead. Optional attribute.
- string – specifies a text string to be rendered. If this is omitted, the current window text for the target window will be used instead. Optional attribute.

Usage:

- The <Text> element may not contain any sub-elements.
- The <Text> element should only appear as a sub-element within <TextComponent> elements.

Examples:

In this simple example, we define a TextComponent that renders some static text. The <Text> element is used to specify the font and string to be used:

```
...
<TextComponent>
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
    <Dim type="Height" ><UnifiedDim scale="1" type="Height" /></Dim>
  </Area>
  <Text font="Roman-18" string="Render this text!" />
</TextComponent>
...
```

<TextComponent> Element

Purpose:

The <TextComponent> element defines a single item of text to be drawn within a given ImagerySection. The TextComponent contains all information about the text that is to be drawn, where it should be drawn, which colours are to be used and how the text should be formatted within its area.

Attributes:

- The <TextComponent> element has no attributes defined.

Usage:

Note: the sub-elements should appear in the order that they are defined here.

- <Area> defining the target area for this image.
- <Text> optional element specifying the font to be used and text string to be drawn.
- Optionally specifying the colours for this text, one of the colour elements:
  - <Colours>
  - <ColourProperty>
  - <ColourRectProperty>
- Optionally, to specify the vertical formatting to use, either of:
  - <VertFormat>
  - <VertFormatProperty>
- Optionally, to specify the horizontal formatting to use, either of:
  - <HorzFormat>
  - <HorzFormatProperty>

- The <TextComponent> element may only appear as a sub-element of the element <ImagerySection>.

Examples:

The following example could be used to specify the caption text to appear within a Titlebar style widget:

```
...
<ImagerySection name="caption">
  <TextComponent>
    <Area>
      <Dim type="LeftEdge" ><AbsoluteDim value="10" /></Dim>
      <Dim type="TopEdge" ><AbsoluteDim value="2" /></Dim>
      <Dim type="Width" ><UnifiedDim scale="1" type="Width" /></Dim>
      <Dim type="Height" ><UnifiedDim scale="1" type="Height" /></Dim>
    </Area>
    <ColourProperty name="CaptionColour" />
    <VertFormat type="CentreAligned" />
    <HorzFormat type="WordWrapLeftAligned" />
  </TextComponent>
</ImagerySection>
...
```

<UnifiedDim> Element

Purpose:

The <UnifiedDim> element is used to define a component dimension for an area rectangle. <UnifiedDim> is used to specify a value using the 'unified' co-ordinate system.

Attributes:

- scale – specifies the relative scale component of the UDim. Optional attribute.
- offset – specifies the absolute pixel offset of the UDim. Optional attribute.
- type – specifies what the dimension represents. This is required so that the system knows how to interpret the 'scale' component. Required attribute.

Usage:

- The <UnifiedDim> element may contain a single <DimOperator> element in order to form a dimension calculation.
- The <UnifiedDim> element can appear as a sub-element in <Dim> to form a dimension specification for an area.
- The <UnifiedDim> element can appear as a sub-element of <DimOperator> to specify the second operand for a dimension calculation.

Examples:

This example shows a dimension that uses <UnifiedDim> to specify a UDim value to use as the dimension's value:

```
...
<Area>
  <Dim type="LeftEdge">
    <UnfiedDim scale="0.5" offset="-8" type="LeftEdge" />
  </Dim>
  ...
</Area>
...
```

<VertAlignment> Element

Purpose:

The <VertAlignment> element is used to specify the vertical alignment option required for a child window element.

Attributes:

- type – specifies one of the values from the VerticalAlignment enumeration indicating the desired vertical alignment.

Usage:

- The <VertAlignment> element may only appear as a sub-element of the <Child> element.
- The <VertAlignment> element may not contain any sub-elements.

Examples:

This example defines a scrollbar type child widget. We have used the <VertAlignment> element to specify that the scrollbar appear on the bottom edge of the component being defined:

```
...
<Child  type="MyLook/HorzScrollbar" nameSuffix="__auto_hscrollbar__">
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><UnifiedDim scale="1" type="Width” /></Dim>
    <Dim type="Height" ><AbsoluteDim value="15" /></Dim>
  </Area>
  <VertAlignment type="BottomAligned" />
</Child>
...
```

<VertFormat> Element

Purpose:

The <VertFormat> element is used to specify the required vertical formatting for an image, frame, or text component.

Attributes:

type – specifies the required vertical formatting option. For use in ImageryComponents and FrameComponents, this will be one of the values from the VerticalFormat enumeration. For use in TextComponents, this will one of the values form the VerticalTextFormat enumeration.

Usage:

- The <VertFormat> element may only appear as a sub-element of the <ImageryComponent>, <FrameComponent>, or <TextComponent> elements.
- The <VertFormat> element may not contain any sub-elements.

Examples:

This first example shows an ImageryComponent definition. We use <VertFormat> to specify that we want the image tiled to cover the entire width of the designated target area:

```
...
<ImageryComponent>
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="Width" ><AbsoluteDim value="25" /></Dim>
    <Dim type="Height" ><AbsoluteDim value="25" /></Dim>
  </Area>
  <Image imageset="myImageset" image="coolImage" />
  <VertFormat type="Tiled" />
  <HorzFormat type="Stretched" />
</ImageryComponent>
...
```

This second example is for a TextComponent. You can see <VertFormat> used here to specify that we want the text centred within the target area:

```
...
<TextComponent>
  <Area>
    <Dim type="LeftEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="TopEdge" ><AbsoluteDim value="0" /></Dim>
    <Dim type="RightEdge" ><UnifiedDim scale="1" type="Width" /></Dim>
    <Dim type="Height" ><UnifiedDim scale="1" type="Height" /></Dim>
  </Area>
  <VertFormat type="CentreAligned" />
</TextComponent>
...
```

<VertFormatProperty> Element

Purpose:

The <VertFormatProperty> element is intended to allow the system to access a property on the target window to obtain the vertical formatting to be used when drawing the component being defined.

Attributes:

- name – specifies the name of the property to access. The named property must access a string value that will be set to one of the enumeration values appropriate for the component being defined (so VerticalTextFormat for TextComponent, and VerticalFormat for FrameComponent and ImageryComponent). Required attribute.

Usage:

- The <VertFormatProperty> element may not contain sub-elements.

The <VertFormatProperty> element may appear as a sub-element within any of the following elements:

- <ImageryComponent> to specify a vertical formatting to be used the the image.
- <FrameComponent> to specify a vertical formatting to be used for the frame background.
- <TextComponent> to specify a vertical formatting to be used for the text.

Examples:

<WidgetDim> Element

Purpose:

The <WidgetDim> element is used to define a component dimension for an area rectangle. <WidgetDim> is used to specify some dimension of an attached child widget for use as an area dimension.

Attributes:

- widget – specifies a suffix which will be used when building the name of the widget to access. The final name of the child widget will be that of the parent with this suffix appended. If this is not specified, the target window itself is used. Optional attribute.
- dimension – specifies the widget dimension to be used. This should be set to one of the values defined in the DimensionType enumeration. Required attribute.

Usage:

- The <WidgetDim> element may contain a single <DimOperator> element in order to form a dimension calculation.
- The <WidgetDim> element can appear as a sub-element in <Dim> to form a dimension specification for an area.
- The <WidgetDim> element can appear as a sub-element of <DimOperator> to specify the second operand for a dimension calculation.

Examples:

This example shows using <WidgetDim> to obtain dimensions from an attached child widget "__auto_titlebar__", and also from the target window itself:

```
...
<Area>
    <Dim type="LeftEdge" >
      <AbsoluteDim value="0" />
    </Dim>
    <Dim type="TopEdge" >
      <WidgetDim widget="__auto_titlebar__" dimension="BottomEdge" />
    </Dim>
    <Dim type="Width" >
      <UnifiedDim scale="1" type="Width" />
    </Dim>
    <Dim type="BottomEdge" >
      <WidgetDim dimension="BottomEdge" />
    </Dim>
</Area>
...
```

<WidgetLook> Element

Purpose:

The <WidgetLook> element is the most important element within the system. It defines a complete widget 'look' which can be assigned to one of the Falagard base widget classes to create what is essentially a new widget type.

Attributes:

- name – specifies the name of the WidgetLook being defined. If a WidgetLook with this name already exists within the system, it will be replaced with the new definition. Required attribute.

Usage:

The <WidgetLook> element can contain the following sub-elements:

Note: the sub-elements should appear in the order that they are defined here.

- Any number of <PropertyDefinition> sub-elements defining new properties.
- Any number of <Property> sub-elements specifying default property values.
- Any number of <NamedArea> sub-elements defining areas within the widget.
- Any number of <Child> sub-elements defining component child widgets.
- Any number of <ImagerySection> sub-elements defining imagery for the widget.
- Any number of <StateImagery> sub-elements defining what to draw for widget states.

- The <WidgetLook> element may only appear as sub-elements of the root <Falagard> element.

Examples:

The following example is the complete definition for “TaharezLook/ListHeader”. This is a trivial example that actually does no rendering, it just specifies a required property:

```
<WidgetLook name="TaharezLook/ListHeader">
  <Property
    name="SegmentWidgetType"
    value="TaharezLook/ListHeaderSegment"
  />
  <StateImagery name="Enabled" />
  <StateImagery name="Disabled" />
</WidgetLook>
```

DimensionOperator Enumeration

- "Noop" – does nothing.
- "Add" – Adds two dimensions.
- "Subtract" – Subtracts two dimensions.
- "Multiply" – Multiplies two dimensions.
- "Divide" – Divides two dimensions.

DimensionType Enumeration

- "LeftEdge" – specifies the left edge of the target item.
- "TopEdge" – specifies the top edge of the target item.
- "RightEdge" – specifies the right edge of the target item.
- "BottomEdge" – specifies the bottom edge of the target item.
- "XPosition" – specifies the x position co-ordinate of the target item (same as “LeftEdge”).
- "YPosition" – specifies the y position co-ordinate of the target item (same as “TopEdge”).
- "Width" – specifies the width of the target item.
- "Height" – specifies the height of the target item.
- "XOffset" – specifies the x offset of the target item (only applies to Images).
- "YOffset" – specifies the y offset of the target item (only applies to Images).

FontMetricType Enumeration

- "LineSpacing" – gets the vertical line spacing value of the font.
- "Baseline" – get the vertical baseline value of the font.
- "HorzExtent" – gets the horizontal extent of a string of text.

FrameImageComponent Enumeration

- “TopLeftCorner” – specifies the image be used for the frame's top-left corner.
- “TopRightCorner” – specifies the image be used for the frame's top-right corner.
- “BottomLeftCorner” – specifies the image be used for the frame's bottom-left corner.
- “BottomRightCorner” – specifies the image be used for the frame's bottom-right corner.
- “LeftEdge” – specifies the image be used for the frame' left edge.
- “RightEdge” – specifies the image be used for the frame's right edge.
- “TopEdge” – specifies the image be used for the frame's top edge.
- “BottomEdge” – specifies the image be used for the frame bottom edge.
- “Background” – specifies the image be used for the frame's background (area formed within all edges).

HorizontalAlignment Enumeration

- "LeftAligned" – x position is an offset of element's left edges.
- "CentreAligned" – x position is an offset of element's horizontal centre points.
- "RightAligned" – x position is an offset of element's right edges.

HorizontalFormat Enumeration

- "LeftAligned" – Image is left aligned within the prescribed area.
- "CentreAligned" – Image is horizontally centred within the prescribed area.
- "RightAligned" – Image is right aligned within the prescribed area.
- "Stretched" – Image is horizontally stretched to fill the prescribed area.
- "Tiled" – Image is horizontally tiled to fill the prescribed area.

HorizontalTextFormat Enumeration

- "LeftAligned" – lines of text are left aligned within the prescribed area.
- "CentreAligned" – lines of text are horizontally centred within the prescribed area.
- "RightAligned" – lines of text are right aligned within the prescribed area.
- "Justified" – lines of text are justified to the prescribed area.
- "WordWrapLeftAligned" – text wraps, with lines left aligned within the prescribed area.
- "WordWrapCentreAligned" – text wraps, with lines horizontally centred in the prescribed area.
- "WordWrapRightAligned" – text wraps, with lines right aligned within the prescribed area.
- "WordWrapJustified" – text wraps, within lines justified to the prescribed area.

PropertyType Enumeration

- "Generic" – specifies a general purpose property.

VerticalAlignment Enumeration

- "TopAligned" – y position is an offset of element's top edges.
- "CentreAligned" – y position is an offset of element's vertical centre points.
- "BottomAligned" – y position is an offset of element's bottom edges.

VerticalFormat Enumeration

- "TopAligned" – Image is aligned with the top of the prescribed area.
- "CentreAligned" – Image is vertically centred within the prescribed area.
- "BottomAligned" – Image is aligned with the bottom of the prescribed area.
- "Stretched" – Image is vertically stretched to fill the prescribed area.

- "Tiled" – Image is vertically tiled to fill the prescribed area.

VerticalTextFormat Enumeration

- "TopAligned" – Text line block is aligned with the top of the prescribed area.
- "CentreAligned" – Text line block is vertically centred within the prescribed area.
- "BottomAligned" – Text line block is aligned with the bottom of the prescribed area.

Falagard Base Widgets Reference

Falagard/Button

General purpose push button widget class.

Assigned WidgetLook should provide the following:

- StateImagery definitions (missing states will default to 'Normal'):
  - Normal – Imagery used when the widget is neither pushed nor has the mouse hovering over it.
  - Hover – Imagery used when the widget has the mouse hovering over it, or when the widget is 'pushed' but the mouse has been moved out of the widget area.
  - Pushed – Imagery used when the widget is pushed and the mouse is over the widget.
  - Disabled – Imagery used when the widget is disabled.

Falagard/Checkbox

General purpose checkbox / toggle button widget class.

Assigned WidgetLook should provide the following:

- StateImagery definitions (missing states will default to 'Normal' or 'SelectedNormal'):
  - Normal – Imagery used when the widget is in the deselected / off state, and is neither pushed nor has the mouse hovering over it.
  - Hover – Imagery used when the widget is in the deselected / off state, and has the mouse hovering over it, or when the widget is in the deselected / off state, and is 'pushed' but the mouse has been moved out of the widget area.
  - Pushed – Imagery used when the widget is in the deselected / off state, and is pushed with the mouse over the widget.
  - Disabled – Imagery used when the widget is in the deselected / off state, and is disabled.
  - SelectedNormal – Imagery used when the widget is in the selected / on state, and is neither pushed nor has the mouse hovering over it.
  - SelectedHover – Imagery used when the widget is in the selected / on state, and has the mouse hovering over it, or when the widget is in the selected / on state, and is 'pushed' but the mouse has been moved out of the widget area.
  - SelectedPushed – Imagery used when the widget is in the selected / on state, and is pushed with the mouse over the widget.
  - SelectedDisabled – Imagery used when the widget is in the selected / on state, and is disabled.

Falagard/Combobox

Widget providing a text box and drop down list invoked via a button.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- Child widget definitions:
  - Editbox based widget with name suffix "__auto_editbox__"
  - ComboDropList based widget with name suffix "__auto_droplist__"
  - PushButton based widget with name suffix "__auto_button__"

Falagard/ComboDropList

Listbox style widget used as the component drop-down list for Combobox widgets.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- NamedArea definitions:
  - ItemRenderingArea – Target area where list items will appear when no scrollbars are visible (also acts as default area). Required.
  - ItemRenderingAreaHScroll – Target area where list items will appear when the horizontal scrollbar is visible. Optional.
  - ItemRenderingAreaVScroll – Target area where list items will appear when the vertical scrollbar is visible. Optional.
  - ItemRenderingAreaHVScroll – Target area where list items will appear when both the horizontal and vertical scrollbars are visible. Optional.

- Child widget definitions:
  - Scrollbar based widget with name suffix "__auto_vscrollbar__". This widget will be used to control vertical scroll position.
  - Scrollbar based widget with name suffix "__auto_hscrollbar__". This widget will be used to control horizontal scroll position.

Falagard/Editbox

General purpose single-line text box widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – Imagery used when widget is enabled.
  - Disabled – Imagery used when widget is disabled.
  - ReadOnly – Imagery used when widget is in 'Read Only' state.
  - ActiveSelection – Additional imagery used when a text selection is defined and the widget is active. The imagery for this state will be rendered within the selection area.
  - InactiveSelection – Additional imagery used when a text selection is defined and the widget is not active. The imagery for this state will be rendered within the selection area.

- NamedArea definitions:
  - TextArea – Defines the area where the text, carat, and any selection imagery will appear.

- ImagerySection definitions:
  - Carat – Additional imagery used to display the insertion position carat.

Falagard/FrameWindow

General purpose window type which can be sized and moved.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - ActiveWithTitleWithFrame – Imagery used when the widget has its title bar enabled, has its frame enabled, and is active.
  - InactiveWithTitleWithFrame – Imagery used when the widget has its title bar enabled, has its frame enabled, and is inactive.
  - DisabledWithTitleWithFrame – Imagery used when the widget has its title bar enabled, has its frame enabled, and is disabled.
  - ActiveWithTitleNoFrame – Imagery used when the widget has its title bar enabled, has its frame disabled, and is active.
  - InactiveWithTitleNoFrame – Imagery used when the widget has its title bar enabled, has its frame disabled, and is inactive.

- DisabledWithTitleNoFrame – Imagery used when the widget has its title bar enabled, has its frame disabled, and is disabled.
- ActiveNoTitleWithFrame – Imagery used when the widget has its title bar disabled, has its frame enabled, and is active.
- InactiveNoTitleWithFrame – Imagery used when the widget has its title bar disabled, has its frame enabled, and is inactive.
- DisabledNoTitleWithFrame – Imagery used when the widget has its title bar disabled, has its frame enabled, and is disabled.
- ActiveNoTitleNoFrame – Imagery used when the widget has its title bar disabled, has its frame disabled, and is active.
- InactiveNoTitleNoFrame – Imagery used when the widget has its title bar disabled, has its frame disabled, and is inactive.
- DisabledNoTitleNoFrame – Imagery used when the widget has its title bar disabled, has its frame disabled, and is disabled.

- NamedArea definitions:
  - ClientWithTitleWithFrame – Area that defines the clipping region for the client area when the widget has its title bar enabled, and has its frame enabled.
  - ClientWithTitleNoFrame – Area that defines the clipping region for the client area when the widget has its title bar enabled, and has its frame disabled.
  - ClientNoTitleWithFrame – Area that defines the clipping region for the client area when the widget has its title bar disabled, and has its frame enabled.
  - ClientNoTitleNoFrame – Area that defines the clipping region for the client area when the widget has its title bar disabled, and has its frame disabled.

- Child widget definitions:
  - Titlebar based widget with name suffix "__auto_titlebar__". This widget will be used as the title bar for the frame window.
  - SystemButton based widget with name suffix "__auto_closebutton__". This widget will be used as the close button for the frame window.

Falagard/Listbox

General purpose single column list widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- NamedArea definitions:
  - ItemRenderingArea – Target area where list items will appear when no scrollbars are visible (also acts as default area). Required.
  - ItemRenderingAreaHScroll – Target area where list items will appear when the horizontal scrollbar is visible. Optional.
  - ItemRenderingAreaVScroll – Target area where list items will appear when the vertical scrollbar is visible. Optional.
  - ItemRenderingAreaHVScroll – Target area where list items will appear when both the horizontal and vertical scrollbars are visible. Optional.

- Child widget definitions:
  - Scrollbar based widget with name suffix "__auto_vscrollbar__". This widget will be used to control vertical scroll position.
  - Scrollbar based widget with name suffix "__auto_hscrollbar__". This widget will be used to control horizontal scroll position.

Falagard/ListHeader

List header widget. Acts as a container for ListHeaderSegment based widgets. Usually used as a component part widget for multi-column list widgets.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- Property initialiser definitions:
  - SegmentWidgetType – specifies the name of a "ListHeaderSegment" based widget type; an instance of which will be created for each column within the header. (Required)

Falagard/ListHeaderSegment

Widget type intended for use as a single column header within a list header widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Disabled – Imagery to use when the widget is disabled.

- Normal – Imagery to use when the widget is enabled and the mouse is not within any part of the segment widget.
- Hover – Imagery to use when the widget is enabled and the mouse is within the main area of the widget (not the drag-sizing 'splitter' area).
- SplitterHover – Imagery to use when the widget is enabled and the mouse is within the drag-sizing 'splitter' area.
- DragGhost – Imagery to use for the drag-moving 'ghost' of the segment. This state should specify that its imagery be render unclipped.
- AscendingSortIcon – Additional imagery used when the segment has the ascending sort direction set.
- DescendingSortDown – Additional imagery used when the segment has the descending sort direction set.
- GhostAscendingSortIcon – Additional imagery used for the drag-moving 'ghost' when the segment has the ascending sort direction set.
- GhostDescendingSortDown – Additional imagery used for the drag-moving 'ghost' when the segment has the descending sort direction set.

- Property initialiser definitions:
  - MovingCursorImage – Property to define a mouse cursor image to use when drag-moving the widget. (Optional).
  - SizingCursorImage – Property to define a mouse cursor image to use when drag-sizing the widget. (Optional).

Falagard/Menubar

General purpose horizontal menu bar widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- NamedArea definitions:
  - ItemRenderingArea – Target area where menu items will appear.

Falagard/MenuItem

General purpose textual menu item widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - EnabledNormal – Imagery used when the item is enabled and the mouse is not within its area.
  - EnabledHover – Imagery used when the item is enabled and the mouse is within its area.
  - EnabledPushed – Imagery used when the item is enabled and user has pushed the mouse button over it.
  - EnabledPopupOpen – Imagery used when the item is enabled and attached popup menu is opened.
  - DisabledNormal – Imagery used when the item is disabled and the mouse is not within its area.
  - DisabledHover – Imagery used when the item is disabled and the mouse is within its area.
  - DisabledPushed – Imagery used when the item is disabled and user has pushed the mouse button over it.
  - DisabledPopupOpen – Imagery used when the item is disabled and attached popup menu is opened.
  - PopupClosedIcon – Additional imagery used when the item is attached to a popup menu widget and has a a 'sub' popup menu attached to itself, and that popup is closed.
  - PopupOpenIcon – Additional imagery used when the item is attached to a popup menu widget and has a a 'sub' popup menu attached to itself, and that popup is open.

Falagard/MultiColumnList

General purpose multi-column list / grid widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- NamedArea definitions:
  - ItemRenderingArea – Target area where list items will appear when no scrollbars are visible (also acts as default area). Required.
  - ItemRenderingAreaHScroll – Target area where list items will appear when the horizontal scrollbar is visible. Optional.
  - ItemRenderingAreaVScroll – Target area where list items will appear when the vertical scrollbar is visible. Optional.

- ItemRenderingAreaHVScroll – Target area where list items will appear when both the horizontal and vertical scrollbars are visible. Optional.

- Child widget definitions:
  - Scrollbar based widget with name suffix "__auto_vscrollbar__". This widget will be used to control vertical scroll position.
  - Scrollbar based widget with name suffix "__auto_hscrollbar__". This widget will be used to control horizontal scroll position.
  - ListHeader based widget with name suffix "__auto_listheader__". This widget will be used for the header (though technically, you can place it anywhere).

Falagard/MultiLineEditbox

General purpose multi-line text box widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – Imagery used when widget is enabled.
  - Disabled – Imagery used when widget is disabled.
  - ReadOnly – Imagery used when widget is in 'Read Only' state.

- NamedArea definitions:
  - TextArea – Target area where text lines will appear when no scrollbars are visible (also acts as default area). Required.
  - TextAreaHScroll – Target area where text lines will appear when the horizontal scrollbar is visible. Optional.
  - TextAreaVScroll – Target area where text lines will appear when the vertical scrollbar is visible. Optional.
  - TextAreaHVScroll – Target area where text lines will appear when both the horizontal and vertical scrollbars are visible. Optional.

- Child widget definitions:
  - Scrollbar based widget with name suffix "__auto_vscrollbar__". This widget will be used to control vertical scroll position.
  - Scrollbar based widget with name suffix "__auto_hscrollbar__". This widget will be used to control horizontal scroll position.

- ImagerySection definitions:
  - Carat – Additional imagery used to display the insertion position carat.

- Property initialiser definitions:
  - SelectionBrushImage – defines name of image that will be painted for the text selection (this is applied on a per-line basis).

Falagard/PopupMenu

General purpose popup menu widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- NamedArea definitions:
  - ItemRenderingArea – Target area where menu items will appear.

Falagard/ProgressBar

General purpose progress widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions (missing states will default to 'Normal' or 'SelectedNormal'):
  - Enabled – General imagery used when widget is enabled.
  - Disabled – General imagery used when widget is disabled.
  - EnabledProgress – imagery for 100% progress used when widget is enabled. The drawn imagery will appear in named area “ProgressArea” and will be clipped appropriately according to widget settings and the current progress value.
  - DisabledProgress – imagery for 100% progress used when widget is disabled. The drawn imagery will appear in named area “ProgressArea” and will be clipped appropriately according to widget settings and the current progress value.

- NamedArea definitions:
  - ProgressArea – Target area where progress imagery will appear.

- Property initialiser definitions:
  - VerticalProgress – boolean property. Determines whether the progress widget is horizontal or vertical. Default is horizontal. Optional.

- ReversedProgress – boolean property. Determines whether the progress grows in the opposite direction to what is considered 'usual'. Set to “True” to have progress grow towards the left or bottom of the progress area. Optional.

Falagard/RadioButton

General purpose radio button style widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions (missing states will default to 'Normal' or 'SelectedNormal'):
  - Normal – Imagery used when the widget is in the deselected / off state, and is neither pushed nor has the mouse hovering over it.
  - Hover – Imagery used when the widget is in the deselected / off state, and has the mouse hovering over it, or when the widget is in the deselected / off state, and is 'pushed' but the mouse has been moved out of the widget area.
  - Pushed – Imagery used when the widget is in the deselected / off state, and is pushed with the mouse over the widget.
  - Disabled – Imagery used when the widget is in the deselected / off state, and is disabled.
  - SelectedNormal – Imagery used when the widget is in the selected / on state, and is neither pushed nor has the mouse hovering over it.
  - SelectedHover – Imagery used when the widget is in the selected / on state, and has the mouse hovering over it, or when the widget is in the selected / on state, and is 'pushed' but the mouse has been moved out of the widget area.
  - SelectedPushed – Imagery used when the widget is in the selected / on state, and is pushed with the mouse over the widget.
  - SelectedDisabled – Imagery used when the widget is in the selected / on state, and is disabled.

Falagard/ScrollablePane

General purpose scrollable pane widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- NamedArea definitions:
  - ViewableArea – Target area where visible content will appear when no scrollbars are

visible (also acts as default area). Required.

- ViewableAreaHScroll – Target area where visible content will appear when the horizontal scrollbar is visible. Optional.
- ViewableAreaVScroll – Target area where visible content will appear when the vertical scrollbar is visible. Optional.
- ViewableAreaHVScroll – Target area where visible content will appear when both the horizontal and vertical scrollbars are visible. Optional.

- Child widget definitions:
  - Scrollbar based widget with name suffix "__auto_vscrollbar__". This widget will be used to control vertical scroll position.
  - Scrollbar based widget with name suffix "__auto_hscrollbar__". This widget will be used to control horizontal scroll position.

Falagard/Scrollbar

General purpose scrollbar widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- NamedArea definitions:
  - ThumbTrackArea – Target area in which thumb may be moved.

- Child widget definitions:
  - Thumb based widget with name suffix "__auto_thumb__". This widget will be used for the scrollbar thumb.
  - PsuhButton based widget with name suffix "__auto_incbtn__". This widget will be used as the increase button.
  - PushButton based widget with name suffix "__auto_decbtn__". This widget will be used as the decrease button.

- Property initialiser definitions:
  - VerticalScrollbar – boolean property. Indicates whether this scrollbar will operate in the vertical or horizontal direction. Default is for horizontal. Optional.

Falagard/Slider

General purpose slider widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- NamedArea definitions:
  - ThumbTrackArea – Target area in which thumb may be moved.

- Child widget definitions:
  - Thumb based widget with name suffix "__auto_thumb__". This widget will be used for the slider thumb.

- Property initialiser definitions:
  - VerticalSlider – boolean property. Indicates whether this slider will operate in the vertical or horizontal direction. Default is for horizontal. Optional.

Falagard/Spinner

General purpose numerical spinner widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- Child widget definitions:
  - Editbox based widget with name suffix "__auto_editbox__". This widget will be used as the text box / display portion of the widget.
  - PushButton based widget with name suffix "__auto_incbtn__". This widget will be used as the increase button.
  - PushButton based widget with name suffix "__auto_decbtn__". This widget will be used as the decrease button.

Falagard/StaticImage

Static widget that displays a configurable image.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.
  - EnabledFrame – Additional imagery used when the widget is enabled and the widget frame is enabled.
  - DisabledFrame – Additional imagery used when the widget is disabled and the widget frame is enabled.
  - EnabledBackground – Additional imagery used when the widget is enabled and the widget background is enabled.
  - DisabledBackground – Additional imagery used when the widget is disabled and the widget background is enabled.

- NamedArea definitions:
  - WithFrameImageRenderArea - Area to render image into when the frame is enabled (optional).
  - NoFrameImageRenderArea - Area to render image into when the frame is disabled (optional).

Falagard/StaticText

Static widget that displays configurable text.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.
  - EnabledFrame – Additional imagery used when the widget is enabled and the widget frame is enabled.
  - DisabledFrame – Additional imagery used when the widget is disabled and the widget frame is enabled.
  - EnabledBackground – Additional imagery used when the widget is enabled and the widget background is enabled.
  - DisabledBackground – Additional imagery used when the widget is disabled and the widget background is enabled.

- NamedArea definitions:
  - TextRenderArea – Target area where text will appear when no scrollbars are visible (also acts as default area). Required.
  - TextRenderAreaHScroll – Target area where text will appear when the horizontal scrollbar is visible. Optional.
  - TextRenderAreaVScroll – Target area where text will appear when the vertical scrollbar is visible. Optional.
  - TextRenderAreaHVScroll – Target area where text will appear when both the horizontal and vertical scrollbars are visible. Optional.

- Child widget definitions:
  - Scrollbar based widget with name suffix "__auto_vscrollbar__". This widget will be used to control vertical scroll position.
  - Scrollbar based widget with name suffix "__auto_hscrollbar__". This widget will be used to control horizontal scroll position.

Falagard/SystemButton

Specialised push button widget intended to be used for 'system' buttons appearing outside of the client area of a frame window style widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions (missing states will default to 'Normal'):
  - Normal – Imagery used when the widget is neither pushed nor has the mouse hovering over it.
  - Hover – Imagery used when the widget has the mouse hovering over it, or when the widget is 'pushed' but the mouse has been moved out of the widget area.
  - Pushed – Imagery used when the widget is pushed and the mouse is over the widget.
  - Disabled – Imagery used when the widget is disabled.

Falagard/TabButton

Special widget type used for tab buttons within a tab control based widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions (missing states will default to 'Normal'):
  - Normal – Imagery used when the widget is neither selected nor has the mouse hovering over it.

- Hover – Imagery used when the widget has the mouse hovering over it.
- Selected – Imagery used when the widget is the active / selected tab.
- Disabled – Imagery used when the widget is disabled.

Falagard/TabControl

General purpose tab control widget.

The current TabControl base class enforces a strict layout, so while imagery can be customised as desired, the general layout of the component widgets is, at least for the time being, fixed.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- Child widget definitions:
  - TabPane based widget with name suffix "__auto_TabPane__". This widget will be used as the content viewing pane.
  - DefaultWindow based widget with name suffix "__auto_TabPane__Buttons". This widget will be used as a container for the tab buttons. Optional.

- Property initialiser definitions:
  - TabButtonType – specifies a TabButton based widget type to be created each time a new tab button is required.

Falagard/TabPane

Special widget type used for the content pane of a tab control based widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

Falagard/Thumb

General purpose thumb class for use as a component in widgets that require a movable thumb or knob.

Assigned WidgetLook should provide the following:

- StateImagery definitions (missing states will default to 'Normal'):
  - Normal – Imagery used when the widget is neither pushed nor has the mouse hovering over it.
  - Hover – Imagery used when the widget has the mouse hovering over it, or when the widget is 'pushed' but the mouse has been moved out of the widget area.
  - Pushed – Imagery used when the widget is pushed and the mouse is over the widget.
  - Disabled – Imagery used when the widget is disabled.

Falagard/Titlebar

Title bar widget intended for use as the title bar of a frame window widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions (missing states will default to 'Normal'):
  - Active – Imagery used when the widget is active.
  - Inactive – Imagery used when the widget is inactive.
  - Disabled – Imagery used when the widget is disabled.

Falagard/Tooltip

General purpose tool-tip widget.

Assigned WidgetLook should provide the following:

- StateImagery definitions:
  - Enabled – General imagery for when the widget is enabled.
  - Disabled – General imagery for when the widget is disabled.

- NamedArea definitions:
  - TextArea – Typically this would be the same area as the TextComponent you define to receive the tool-tip text. This named area is used when deciding how to dynamically size the tool-tip so that text is not clipped.