SharpMedia GUI Library Design

© 2008Žiga Osolin – SharpMedia project, document version: DRAFT

Table Of Contents

[About 3](#_Toc197308466)

[Goals 3](#_Toc197308467)

[Overview 4](#_Toc197308468)

[Metrics 4](#_Toc197308469)

[Styles 5](#_Toc197308470)

[Substyles 5](#_Toc197308471)

[State Styles 5](#_Toc197308472)

[Style 5](#_Toc197308473)

[Style Animation 7](#_Toc197308474)

[Changing rules 7](#_Toc197308475)

[Themes 8](#_Toc197308476)

[Widget Rendering 8](#_Toc197308477)

[Animations 9](#_Toc197308478)

[IAnimation and AnimationProcess 9](#_Toc197308479)

[Animation Types 9](#_Toc197308480)

[Validation 10](#_Toc197308481)

[Layouting 11](#_Toc197308482)

[Widgets 11](#_Toc197308483)

[Changing Mechanism 11](#_Toc197308484)

[Serialization 11](#_Toc197308485)

[Font Style and Text Interactions 12](#_Toc197308486)

[IContainer 12](#_Toc197308487)

[Widget Listing 13](#_Toc197308488)

[Area 14](#_Toc197308489)

[Label 16](#_Toc197308490)

[DataEntry 17](#_Toc197308491)

[Button 18](#_Toc197308492)

[VectorImage 19](#_Toc197308493)

[AnimatableVectorImage 20](#_Toc197308494)

[TextInput 21](#_Toc197308495)

[Container 22](#_Toc197308496)

[HBox (VBox) 24](#_Toc197308497)

[WindowSkeleton 25](#_Toc197308498)

[Table 26](#_Toc197308499)

[ListBase 28](#_Toc197308500)

[Menu, DateChoser, Spacer, HSlider, List, , CheckBox, RadioButton, VRuler, RenderArea, ConsoleArea 29](#_Toc197308501)

[XML Format 31](#_Toc197308502)

# About

A GUI Library is built upon Vector graphics library and provides common facilities for Graphical User Interface (GUI). It is highly customizable (through themes) and extendable (through inheritance). It also provides XML->C# translation facilities for fast content creation.

# Goals

The goals of GUI library are:

* Common widgets with correct user interactions;
* Highly customizable appearance of widgets through themes (where rendering and default styles are specified);
* Style framework that allow style inheritance, immutability and more;
* Dynamic, easy to use layouting containers;
* Smooth transitions between styles states, smooth transitions of changes (if required);
* Easy to use and extend API;
* Usable as windows interface or HUD API;
* Easy Shell integration (this is actually implemented by Shell);
* Fast content creation through XML->C# translation or GUI Designer (later as a tool).

# Overview

SharpMedia GUI Library is organized into several namespaces, each belonging to specific aspect of GUI.

GUI Library depends on Vector Graphics. The following features still need to be implemented:

* Clipping regions (you can specify clipping regions for object, through ICanvas.ClippingRegions)
* Multiple transforms per batch (using matrix skinning)
* Multiple clipping regions per batch (using region »skinning«)

# Metrics

We use three type of units for metrics for maximum flexibility. You can describe any position or size by combination of three coordinate systems (that are then all converted to canvas units):

* Canvas units, usually in range [0,1]x[0,1] (but can be defined otherwise by ICanvas);
* Pixel units, as canvas interprets pixel (usually corresponds to render target's pixel size);
* Parent units, in range [0,1]x[0,1] (based on parent bounding rectangle).

This system allows very flexible layouting. Metrics defines **GuiScalar**, **GuiVector2** and **GuiRect**, each with its obvious meaning and implementing this three-unit system. Besides all that, rectangle allow different kinds of parameter meaning: min/max or one corner and size.

# Styles

Styles define *rendering* of widget. Each widget has its own style type which defines its look. Styles can be inherited and default styles are provided by themes[[1]](#footnote-2).

## Substyles

A substyle defines a sub-element of rendering, such as border, background, font etc. All substyles support inheritance (that is, take the properties of parent substyle if not contained in its definition). They must be easily cloneable and pre-change signalizer.

Basic substyles are: **FontSubstyle[[2]](#footnote-3), BackgroundSubstyle, BorderSubstyle**.

## State Styles

Styles define how specific widget is rendered for specific state of widget. For example, **Button** may have different rendering, depending on its state – is mouse over and clicked or not. Each state style is specific to widget it is assigned to, for example **Button** has **ButtonStyle**.

A state style can have several substyles. For example, a label may have the following substyles:

* Border;
* Background;
* SelectedBackground;
* TextFont;
* SelectedTextFont;

Each state style must support cloning, inheritance and pre-change firing.

## Style

**Style** contains state – state style pairs. It allows you to retrieve style that corresponds to certain state of widget. It is also responsible for providing the default style if state style for specific state is not available. The default linking is presented in the following hierarchy:

*All state styles of Style are of the same type.* Styles also support cloning, inheritance and pre-change firing.

## Style Animation

**StyleAnimationController** is usually used by widgets to identify their styles. The controller allows you to perform style interpolation if this kind of rendering is supported by current theme. This means that two styles and weight between them define the look of widget.

Style animation controller let's you specify current style, or specify transition. The transition between styles takes the time of animation, next state and **IUniformInterpolator** as arguments. This means that you can simply create non-linear interpolations. The default interpolation technique is smooth-step (no sharp edges). Through this class, you can map style change pairs to specific interpolators and transition times. This will be used if transition with only traget state argument is used.

## Changing rules

Every mutable object in style hierarchy (substyles, state styles and styles) has changing rules. Once style is assigned to a widget, it is forbidden to be changed. In order to change it, you must enter change context[[3]](#footnote-4) of all widgets that use it, and then you can change it. If you do not do this, exception will be thrown. This allows for style sharing. This means that styles that will be changed very often should be assigned on one-per-widget basis and then, they can be quite easily manipulated.

Styles provided by theme cannot be changed in their lifetime.

Implementation of changing rules is done through delegates. Whenever a dependency is added to style (state style), its pre-changed event is bound to style's method and style is signaled before any changes are made to this state style. The chaining is done in the same way for substyles. Parents are chained in the same manner, and widgets chain themselves to styles. When dependency is relinked, the assigned delegate to its pre-change event is also removed.

# Themes

A theme defines rendering of widgets. A theme must provide the following:

* Default styles (immutable) for widgets it supports;
* Default renderers (**ISkinRenderer**) for each widget it supports.

A theme is usually assigned to the whole GUI heirarchy of widgets, but this is not the rule. You can have several themes in the same GUI hierarchy, overwritten by custom styles and also custom renderers.

## Widget Rendering

Widgets are usually rendered in the following manner (example is for **Area**, which has no special »features«):

1. We first obtain both state styles (one for previous state and one for state we are going to, with weight). If the weight is 1.0 (meaning transition was done), the previous state may be set to the current;
2. We get and merge **Fill**s for both background substyles. If fill is null, we skip rendering, otherwise we render background. We also take into account mapping transforms and mapper;
3. We get and merge **Pen**s of both border substyles (Pen.Merge) and render border. If border pen is null (no border), we simply do not render it. The shape of border is specified by widget. We also take into account mapping transforms and mapper.

# Animations

This section deals with animations of GUI elements, not styles. Style animation is more or less automatic and controlled by **StyleAnimationController**.

## IAnimation and AnimationProcess

An animation is interface of type **IAnimation** that contains only two members; how long it lasts (e.g. AnimationTime property) and **Update** method. Update is automatically called by widget. It takes two arguments; animation time and object to apply animation to. This means that animation object is independant of its destination (object it affects) and can be freely shared.

Widget actually wraps each **IAnimation** with **AnimationProcess** that also tracks time of animation. **IAnimation** usually performs its update through reflection provided **PropertyInfo**/**FieldInfo**. This makes animations very generic; you can apply specific animation to all properties that deal with the same type.

## Animation Types

We split animations into several groups, based on most common GUI variables:

* GuiScalar animations – allow interpolations, loopers, inverse loopers and more;
* GuiVector2 animations – allow interpolations, loopers, inverse loopers, and composite of two GuiScalar animations;
* GuiRect animations – allow interpolations, loopers, inverse loopers and composite of two GuiVector2 animations;
* Float animations – allows interpolations, loopers, inverse loopers;
* Int animations – allows interpolations, loopers, inverse loopers;
* String animations – allow time based (Int animation) character addition;
* ...

# Validation

Validation is added to specific widgets. A result of validation can be *Valid, PartlyValid* or *NotValid*. The widget itself must then act acordingly. For example, the widget may not allow NotValid state, or just not allow leaving focus if state is invalid. Widget may also choose to jump to different style state if validation is invalid etc.

Most common validators are **RegexValidator, NumberValidator and \*InRangeValidator**, mostly implemented by regex expressions.

# Layouting

Layouting is done through **ILayoutNegotiation** interface. Interface provides methods that allow the widget to describe itselft to parent layouting element. The parent layouting widget is free to use any of those »hints«. Each layouting widget may interpret it in its own way. The actual rules will be described by the layout itself.

The properties are: minimum size, preferred size, margin (left, right, top, bottom), preferred rectange, should ratio width/height be preserved and layout anchor (docking properties).

# Widgets

A widget is composite of:

* **IPositionable –** allows obtaining and setting bounding box (the latter can only be done by layouting). Also contains shape and border of widget;
* **IUserInteractive –** user events are propagated through this interface. This includes mouse enter/leave, mouse move, mouse button and keyboard events;
* **IAnimatable –** allows attaching animations (**IAnimation**) to objects;
* **ILayoutNegototiation –** described by layouting;
* **IDisplayObject –** adds Style, Renderer, Z-Order and StyleAnimationController getters.

Besides those interface, widget also contains parent link and changing mechanism.

## Changing Mechanism

All widgets can only be changed inside change context. This means that you must first enter change context, change the widget in context and exit the change context. This ensures atomicy of this operation (rendering can render only in pre-enter or post-exit state).

Before any change, you must call **EnterChange()** and when you are done, you should call **EndChange()**. Each EnterChange must have an EndChange() pair. They can be scoped (multiple Enters, but then the same count of exits). While in change state, you can set any properties. When not in change state, all such calls will throw and **InvalidOperationException**.

## Serialization

The whole GUI hiearchy is serializable. Only internal data is serialized, no events. The widget can do custom deserialization/serialization actions using serialization API. The user can control the serialization through **WidgetSerializations**:

* PersistState – perists states of of widget
* PersistEvents – persists event bindings (this is not recommended)
* PersistNonThemeStyles – should non-theme styles be persisted
* PersistAllStyles – should all styles be persisted

## Font Style and Text Interactions

It is vital where certain character is positioned when rendered not only visually, but also for user events. For maximum flexibility, all text rendering is intercepted so each character is exactly positioned and this is known to the widget.

Interception is done through **IFont**. Each rendering must use the font interception provided by widget. This interception can be obtained by simply casting widget to **IFontIntercept**.

## IContainer

**IContainer** is a widget that contains children and organizes them according to layouting strategy. A container has additional properties for appearance (e.g. is vertical/horizonal slider allowed, is drawing out ob bounding box allowed (for drop-down lists) etc.). A container can be **IMarginable**. This means that margins can be set and is also used by widgets.

# Widget Listing

A widget and container listing with description and implementation issues follow. All widgets are implemented with seperate **[WidgetName]Event**class that serves all events allowed by widget. Each widget exposes its properties directly, but note that setting them is allowed only in change context. This does not apply to events.

## Area

*Extends*: **IWidget**

*Description*: Area is base class (directly or indirectly) of all widgets implemented by GUI Library. It contains basic interface implementations with default handlings and allowed overwrites. This class makes widgets easy to implement. It also sets standards for widget API.

*Events:*

* MouseOver
* MouseLeave
* MouseWheel
* MouseMove
* MousePressed
* MouseReleased
* KeyPress
* KeyRelease
* FocusGain
* FocusLost
* DescriptionShow
* DescriptionHide
* StyleStateChange

*Methods:*

* *GetBoundingRect*
* *AddAnimation*
* *RemoveAnimation*
* *EnterChange*
* *ExitChange*

*Properties:*

* *Left/Right/Top/BottomMargin*
* *PreferredRect*
* *MinSize*
* *MaxSize*
* *LayoutAnchor*
* *PreferredSize*
* StyleAnimationController – is is allowed to change style state outlide widget change context
* Manager – obtains the root GuiManager;
* *Z-Order*
* *Style*
* *Skin*
* *Shape*
* *Outline*
* IsVisible – is the widget visible. If it is not visible, it is not processed (not even by layouts).
* ContainedDescriptionTime – the time that triggers the contained description event.
* ContainedDescriptionPositioning – how is the contained description positioned
* ContainedDescription – the description widget when pointer is over this area for contained description time

*Style*: **AreaStyle** consists of border and background substyles.

*Style states:* Normal, PointerOver, Focused

An area with gradient background and border, contained withing bounding rect.

## Label

*Extends*: **Area**

*Description*: A text written in certain area.

*Events:*

* TextSelect
* TextDeselect
* TextCopy
* CursorPositionChange
* TextChange

*Properties:*

* Text
* TextSelectedRange – the text selection range, start and end offset.
* IsEnabled – this specifies whether text can be selected.
* CursorTextPosition

*Style*: **LabelStyle**, in addition to **AreaStyle** consists of text, selected text and selected background substyles. Selected text and text substyles must match in everything but fill (e.g. all the flags, sizes etc.).

*Style states:* Normal, PointerOver, Focused

A label without border and without background fill, only text. The text marked as red is selected. No special background selection colour is set.

Some text as label (centered)

## DataEntry

*Extends*: **Area**

*Description*: An abstract class that is base for all input based classes.

*Events:*

* DataChange – when data is changed.
* DataValidation – whenever validation is performed.

*Properties:*

* IsEnabled – if disabled, cannot obtain focus.
* ValidationType – one of None, WhenFocusLost, WhenDataReceived

*Style*: **DataEntryStyle**, no specific data apart from **AreaStyle** members.

*Style states:* abstract class

## Button

*Extends*: **Area**

*Description*: A button is an element that allows clicking. This container will not forward user events but will handle it internally. ContainedObject is used only for rendering representation. **Button uses (in priority order) preferred rect or preferred size for layouting (those two are exclusive).** If none of those is set, the contained element takes the whole bounding rectangle. Margins are included in all cases. Positioning is centered if not using preferred rectangle.

*Events:*

* ButtonClicked – when mouse button is pressed and released over button.
* ContainedObjectChange

*Properties:*

* ContainedObject – object contained inside button (representation).

*Style*: **ButtonStyle** has no special members.

*Style states:* Normal, PointerOver, Clicked, Focused

Style state is transfered to bottom widget with the following mapping:   
Normal -> Normal; PointerOver -> PointerOver, Clicked -> Clicked

A button with border and background fill, internal element has preferred size of (70%,70%) and it is centered inside the button (in this case, it is area with gradient fill).

## VectorImage

*Extends*: **Area**

*Description*: A vector image widget. It is described as array of { **IShapef/IPathf, Fill/Pen, Mapper, Transform** }. The theme is responsible to draw them. If you need even more customization, you can append your own renderer.

*Events:*

* ImageDataChange – when image data is changed.

*Properties:*

* ImageData – array of data to render. You must call this even if data pointer is not changed but the actual rendering data is changed. This will force dirty state.
* ImageRect – the source rectangle where image is contained, default is [0,1]x[0,1].

*Style*: **VectorImageStyle** has no special members.

*Style states:* Normal, PointerOver   
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
A vector graphics drawn as image. Note that paths and fills can be combined, with different fills/pens.

## AnimatableVectorImage

*Extends*: **VectorImage**

*Description*: Animatable vector image provides wrappers for easily updating animation data.

*Events:*

* ImageDataUpdate – triggered when image data should be updated.

*Methods:*

* ResetImageDataAnimation – sets animation time to 0.
* StopImageDataAnimation
* StartImageDataAnimation

*Properties:*

* ImageDataUpdateInterval – the interval when image data should be changed.
* ImageDataAnimationTime – the time of running animation.

*Style*: **AnimatableVectorImageStyle** has no special members.

## TextInput

*Extends*: **DataEntry**

*Description*: A single-line text input.

*Events:*

* TextSelect
* TextDeselect
* TextCopy
* CursorPositionChange
* TextChange

*Properties:*

* Text
* MaxCharacters – maximum number of characters by input
* Validator
* TextSelectedRange – the text selection range, start and end offset
* IsEnabled – this specifies whether text can be edited
* CursorTextPosition
* ValidationUserInterface – one of None or Style-based. You can use None and do some sort of external validation (image based for instance).

*Style*: **TextInputStyle**, in addition to DataEntryStyle, consists of text font, selected text font and selected background substyles. Selected text and text substyles must match in everything but fill (e.g. all the flags, sizes etc.).

*Style states:* Normal, PointerOver, Focused (when typing data) and optional styles (if style based validation) NormalInvalid, PointerOverInvalid and FocusedInvalid.

A text is can be edited in this box. In this example, it has border, but no background fill (or perhaps white fill). Text can be edited when mouse is clicked. It can also be copied in the same way as with Label.

Text being written ...

## **Container**

*Extends*: **Area**

*Description*: A basic container that does no inteligent layouting. All sub-widgets are required to specify **preferred rectangle** and they are positioned as specified by this rectangle (they are clipped to parent of course). If widget does not provide this rectangle, it is processed as if it were not visible (zero bounding rectangle is set). The container allows the children to span over each other, this should be carefully designed by the rectangels themselves (such coordinates so they cannot occupy the same space).

*Events:*

* ChildAdded
* ChildRemoved
* WidgetMoved – fired when widget is moved

*Methods:*

* AddChild – adds a widget to container. Child can be any object.
* RemoveChild – removes child from container.
* AddChildAt – adds child at index
* RemoveChildAt – removes child from index
* GetChildAt – obtains child at index
* GetChildAtPosition – obtains child at certain position
* GetChildrenInBounds – obtains children in specific bounds (rectangle)

*Properties:*

* *Children – all children as object*
* *WidgetChildren – only widget children*
* *ContainerOptions – bit field of HSlider, VSlider,HDropDown, VDropDown. Does not set unsupported by container.*
* WidgetMoved – fired when widget is moved

*Style*: **ContainerStyle**, no additional information apart **AreaStyle**.

*Style states:* Normal, PointerOver, Focused

Some upper left aligned text in a Container with fixed position by preferred rectangle.

A container with two children and bounding boxes as specified by the widgets.

## HBox (VBox[[4]](#footnote-5))

*Extends*: **Container, IMarginable**

*Description*: A container that aligns children into single horizontal row. It uses the **PreferredRect** (this is favoured over preferred size) or **PreferredSize** and **Margins** to calculate positions. The height is calculate as the maximum preferred height plus the margins, or shrinked (to minimum height plus margins) if this is required by parent. Margins are never violated. Before horizontal slider is used, it tries to pack children using **MinSize's** width (if available - it takes a compromise to fit all children in available span). If this is not enough, a slider is added and upper strategy is used. The width/height ratio is preserved if required.

HBox also allows the widgets to be moved by drag&drop. If widgets are equally spaced, you can »span« widget over multiply grids if you add it multiple times as children (one after another).

*Events:*

* HSliderSpawn – fired when slider is used (before it is used, can customize it here)
* HSliderDisappear – slider not neede anymore

*Properties:*

* HSlider –a horizontal slider. Can be set and will be used when needed
* IsHSliderUsed – is the slider used (this is the same as checking if HSlider.IsVisible), may force to use it (otherwise automatic)
* IsEquallySpaced – should all data be equally spaced
* MinSizeOfChildArea – minimum size of area

*Style*: **HBoxStyle**, adds a border style for gridding the container.

*Style states:* Normal, PointerOver, Focused  
  
  
  
  
  
  
  
  
  
  
  
A non-equally spaced Hbox without HSlider, but with non-null border style for gridding.

Some other label with more text

Some label as second

Some label as first

## WindowSkeleton[[5]](#footnote-6)

*Extends*: **Area**

*Description*: A window skeleton implements the outer, usually Shell controlled part of window. This consists of two parts: upper part (usually a HBox with applications's image, followed by label with title and at last 1-4 buttons). The lower part is a **ContainedObject**, which is not used by Shell (Shell copies window data from texture to this region). The upper object defines the width and minimum height.  
All user events are forwarded to appropriate object.

*Events:*

* ContainedObjectChange
* UpperObjectChange

*Properties:*

* *UpperObject*
* *ContainedObject*

*Style*: **WindowSkeletonStyle** adds no substyles.

*Style states:* Normal, PointerOver, Focused

An upper object area

Contained object area

A simple window skeleton construction, with outer border and no fill. Upper object is bordered, contained object is simple label without border/fill.

# Table

*Extends*: **Container, IMarginable**

*Description*: A container that organizes children in grid (e.g. table). It uses **PreferredRect** (favoured) or **PreferredSize** and **Margins** to construct table. If table does not fit in appropriate area, it uses **MinSize** (if available) before using any sliders to contain data. If this is not enough, sliders are automatically added.  
A child can span over more grids, this is fully defined by (x, y, width, height), all unsigned integers.

Table can be used in interactive mode with support for drag&drop actions and element extend actions. Drag&drop is done if dragging is issued and extending is done by double clicking on border that should be »freed«.

*Events:*

* HSliderSpawn – fired when slider is used (before it is used, can customize it here)
* HSliderDisappear – slider not needed anymore
* VSliderSpawn – fired when slider is used (before it is used, can customize it here)
* VSliderDisappear – slider not needed anymore
* ChildMove – a child was moved to another cell, may disallow this here
* ChildExtend – a child was extended to another cell, may disallow this here

*Methods:*

* AddChild – adds a child at specific position and span, overloads for multi-row span and more
* ExtendChild – extends a specific child
* MoveChild – moves a child to other cell

*Properties:*

* HSlider –a horizontal slider. Can be set and will be used when needed
* IsHSliderUsed – is the slider used (this is the same as checking if HSlider.IsVisible), may force to use it (otherwise automatic)
* IsHEquallySpaced – should data be equally spaced horizontally
* VSlider –a vertical slider. Can be set and will be used when needed
* IsVSliderUsed – is the slider used (this is the same as checking if VSlider.IsVisible), may force to use it (otherwise automatic)
* IsVEquallySpaced – should data be equally spaced vertically
* MinSizeOfChildArea – minimum grid size
* AllowDragDrop – allows drag&drop actions of elements in table
* AllowExtend – allows extending and shrinking children over multiple cells
* RowCount – row count, may be unspecified (calculated)
* ColumnCount – coloums count, may be unspecified (calculated)

*Style*: **TableStlye**, adds a border style for gridding the container (seperate vertical and horizontal)

*Style states:* Normal, PointerOver, Focused  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
A sample table. Table features different widget children, a multi-column span, equally spacing in rows and non-equal spacing in columns. Cells can also be empty. If the table is interactive, you could expand (for example) the arrow to middle cell. You could also move the arrow there.

The cell on left is empty, this text is alligned to be fully fitted in this cell (with the space available by cell.

**Some text in Cell (2,0) spanning over more cells**

Some text in Cell (0,0)

# ListBase

*Extends*: **DataEntry**

*Description*: A **abstract** base class for all list-based containers. All lists present data in some form and allow data selection. Multiple-selection, drop down, moving, ... are all supported by lists.

*Events:*

* DataSelect – triggerred everytime data is selected
* DataDeselect – trigerred when data is deselected
* DataClick – data was clicked
* DataDoubleClick – data was double clicked
* DataMove – data was moved (if moving is allowed)

*Properties:*

* IsMultiSelection – is multi-selection allowed
* IsMoveAllowed – can the data be moved
* SelectedData – gets indices of data selected (or sets)
* ForwardStrategy – one of None, AutomaticStyleStates, ForwardInteraction

*Style*: **ListBaseStyle** has no special members.

*Style states:* Abstract

# Menu, DateChoser, Spacer, HSlider, List, , CheckBox, RadioButton, VRuler, RenderArea, ConsoleArea

# XML Format

1. More about this in Themes section. [↑](#footnote-ref-2)
2. Note that FontSubStyle not only defines the look, but also the interaction (where exactly is the letter). [↑](#footnote-ref-3)
3. More about this in Widgets section. [↑](#footnote-ref-4)
4. VBox is the same, only that VSlider is used and the organisation is vertically, not horizontally. [↑](#footnote-ref-5)
5. We need a more appropriate name. [↑](#footnote-ref-6)