R3 GUI Styles

Author: Carl Sassenrath, Saphirion AG

Revision History

Date	A
13-Jan-2013/15:50:55+1:00	

Table of Contents

1. Concepts

`Styles` define GUI elements.

A GUI is built from a collection of styles, such as `button`, `field`, `text`, and `image`. In essence a style defines a `class` of user interface objects, specifying the attributes, variables, and functions that are used to create and operate such an object. An `instance` of a style is a user interface object called a face, one or more of which are displayed in a layout.

Every style has a `name` that refers to the style during the construction of a GUI, within documentation, or later for debugging purposes. For example, `button` is the name for a style that looks and acts like a push-button.

Styles can be collected into a `style-sheet` that provides all of the definitions for the desired look and feel of a GUI. Style-sheets can be global across an entire application, or local to a specific section of the application. The GUI system supplies a default style-sheet that includes a standard set of useful styles.

2. Using Styles

GUI styles are used in two main ways:

- To create face objects to be displayed in the GUI. \in Styles create faces from a layout block (with the GUI dialect) provided to the `view` function, or by calling the `make-face` function directly. /in
- To create `new` styles from existing styles. \in The new styles can be simple graphical changes or deeper modifications to the behavior of the style. For example, a `cancel-button` style might be defined from a plain `button`, but with a different color, shape, or default action. /in

2.1. With a Layout

Here's an example of styles used within a layout block.

```
view [
        text "Enter your name:"
        field
        button "Submit"
]
```

The words `text`, `field`, and `button` are the names of styles. They are used here to create faces that will be shown in the GUI when the window is created.

2.2. With Make-Face

You can also create a face from a style by calling the `make-face` function directly and providing the style's name and specification block:

```
btn-face: make-face 'button [text: "Submit"]
```

For more examples, see the R3 GUI Faces section.

With regard to the creation of new styles from existing styles, that is the subject covered in detail below.

3. Predefined Styles

The system defines a standard set of styles that implement the basic set of graphical elements (the widgets) of a GUI.

!add full list of styles, this is just a start

Style	Description
area	text input area
button	expandable button
drawing	scalar vector graphic
field	text input field
image	bitmap image
h1	heading of level 1

h2	heading of level 2
scroller	scrollbar
slider	sliding controller
text	section of text
toggle	toggle button

4. Defining Styles

Styles are normally defined form existing styles. This approach saves time because you only need to define the additional attributes required by the new style.

When it comes to defining new styles, there are three common patterns, depending on how close existing styles are to what you need:

- Minor variations of another style. \in Often for commonly reused element of your GUI. These make simple changes to the color, size, or other basic attributes. /in
- Derived from another style. \in Starts with a basic foundation from another style, but makes substantial additions to its attributes, rendering, or actor functions. /in
- Entirely new style. \in When no other styles have what you need, you must build one from scratch. This will require greater knowledge of the system. /in

4.1. Style Functions

New styles are created by providing a definition block to the `stylize` or `make-style` functions:

Function	Description
stylize	Define new styles from a block of names and definitions. The specification of the style is provided in a block format, similar to that used by objects.
make-style	Define a new style of a given name. A parent style can be specified. The specification of the style is provided in a block format, similar to that used by objects.

Note that various other functions related to styles can be found on the R3 GUI Faces page.

4.2. Definition Format

In order to make style definitions easier to create and maintain, the `stylize` function accepts a special declaration format of the general syntax:

```
new-style: parent-style [specifications]
```

The specifications block consists of field names followed followed by their settings, attributes, or functions. The field names are those used by the style object, described in the next section.

Here's an example of the actual style definition for `button`:

```
button: clicker [

        about: "Single action button with text."

        tags: [action tab]

        facets: [
                init-size: 130x24
                text: "Button"
                text-style: 'button
                max-size: 260x24
                min-size: 24x24
                text-size-pad: 20x0
        ]

        options: [
                text: [string! block!]
                bg-color: [tuple!]
                init-size: [pair!]
                wide: [percent!]
                face-width: [integer!]
        ]

        actors: [
                on-set: [
                        if arg/1 = 'value [
                                face/facets/text: form any [arg/2 ""]
                                show-later face
                        ]
                ]
                on-get: [
                        if arg = 'value [
                                face/facets/text
                        ]
                ]
                on-draw: [
                        t: get-facet face 'text
                        ; limit-text-size modifies, so we need to copy
                        ; size is made 20px smaller to incorporate "..." (see text-size-
                        l: limit-text-size copy t face/gob/size - face/facets/text-size-
                        set-facet face 'text-body either equal? length? t length? l [t]
                        do-style/style face 'on-draw arg 'clicker
                ]
        ]
]
```

This simple style is based on the `clicker` style which defines the main actor functions, as reused for buttons. It adds a new `about` string, a few new `facets`, some `options` and few additional `actors`.

Many examples will be shown below.

4.3. Style Object

A style object includes at least these fields:

Field	Datatype	Description

name	word!	Identifies a style for construction or debugging purposes. The system finds styles by their unique names. Example names: button and text.
about	string!	A short summary of the style for display by help and documentation functions.
parent	word!	The name of the parent style. For example, the parent of button is a clicker.
facets	object!	Holds named attributes that are referenced within other parts of the style. For example the init-size or bg-color of the style.
options	object!	Optional attributes that are used inline within the GUI definition as shortcuts. For example, a pair value may specify the size of an object.
actors	map!	Functions that are called by the GUI system at specific times for specific purposes. For example, on-click and on-resize.
draw	block!	A block of draw commands and their arguments that are used to render the style into a displayed image.
state	block!	Holds the prototype definition for the face instance variables used by a style.
content	block!	The layout dialect for compound styles.

4.4. Standard Facet Names

Although you can add any facet name you want, many functions of the GUI depend on standard names. For example, the `bg-color` is used for creating the (gradient) backgrounds of most graphical elements.

Some of the common names are:

Name	Description
init-size	define
min-size	define

max-size	define
bg-color	define
pen-color	define
area-fill	define
edge-color	define

add complete list!

5. Example Definitions

Here are some examples of the most common style creation methods.

5.1. Simple Variation

Here is a very simple variation on a style. It's called `red-box`, and it is based on the `box` style. It makes only one change, to define a new facet for the `bg-color`. The rest of the style is inherited from the `box` style which is a base-style of the GUI system.

```
stylize [
        red-box: box [
                facets: [bg-color: maroon]
        ]
]
view [red-box]
```

It should be noted that to make a new style from an existing style, you must know the names the facets and how they are being used. In the code above, we know that `bg-color` controls the background color of the box. When you are not sure of what facets are used, you can review the definition of the parent style.

Here's another example of `stylize` that creates a new button style of a different color and default text label:

```
stylize [
        stop-button: button [
                about: "Implements a red stop button"
                facets: [
                        bg-color: 200.0.0
                        text: "Stop"
                ]
        ]
]
view [stop-button]
```

5.2. New Style

Here's an example of a completely new style called `circle` that draws a circle of a fixed size and color:

```
stylize [
        circle: [
                about: "A circle style"
                facets: [
```

```
                        init-size: 100x100
                ]
                draw: [
                        pen black
                        line-width 2.7
                        fill-pen maroon
                        circle 50x50 40
                ]
        ]
]
view [circle]
```

The facets block provides the initial size for the face and the draw block provides the drawing instructions.

Of course, this is just a static circle style with no variations, so let’s add a few.

5.3. Facet Attributes

Taking the above circle style, we want to `parameterize` its size and color. This allows its attributes to be specified later, when it is used within a GUI.

Running this code and resizing the window, we see:

```
stylize [
        circle: [
                about: "A resizable circle style"
                facets: [
                        init-size: 100x100
                        max-size: 1000x1000
                        fill-color: maroon
                        edge-color: black
                ]
                draw: [
                        pen edge-color
                        line-width 2.7
                        fill-pen fill-color
                        circle viewport-box/center
                        ((min viewport-box/bottom-right/y viewport-box/bottom-right/x) -
                ]
        ]
]
view [circle]
```

Several fields have been added to the facets, and you can see how they are used in the draw block.

Notice the `max-size` field; it allows the face to resize automatically. The `draw` block includes a bit of math to help compute the new values. Note that in the `draw` block, you can obtain the inner size of the face using `viewport-box/bottom-right`, which is a standard facet for all styles and thus not displayed in the facets block above.

When displayed, if a size is not provided, it will use its default size. However, it can be resized. If you enter this example, drag the window corner to see the circle resize automatically.

5.4. Options

An option is an attribute that can be specified directly within the GUI layout code.

For example, let's allow the circle color to be provided as an option. In addition, let's expand the code to use that color to make a gradient within the circle.

```
stylize [
        circle: [
                about: "A colorable resizable circle style"
                facets: [
                        init-size: 100x100
                        max-size: 1000x1000
                        fill-color: maroon
                        edge-color: black
                ]
                options: [
                        fill-color: [tuple!]
                ]
                draw: [
                        pen edge-color
                        line-width 2.7
                        fill-pen fill-color
                        circle viewport-box/center
                        ((min viewport-box/bottom-right/y viewport-box/bottom-right/x) -
                ]
        ]
]
view [
        circle red
        circle green
        circle blue
]
```

Resize the window, and you will see all three automatically resize.

5.5. Attributes and Actors

You will notice that the above example recomputes a few values in the draw block every time the object is redrawn. Normally, this is not a problem, but if those computations were more complex, they may add a lot of extra time to the redraw, slowing down the GUI.

Instead, you can move some of those computed values out of the draw block, and make them attributes of the face. Then, you add some code to compute the values only when necessary.

Here's an example:

```
stylize [
        circle: [
                about: "Draws a resizable circle."
                facets: [
                        init-size: 100x100
                        max-size: 1000x1000
                        fill-color: maroon
                        edge-color: black
                        area-fill: none
                        radius: 48
                ]
                options: [
                        fill-color: [tuple!]
                ]
                draw: [
```

```
                        pen edge-color
                        line-width 2.7
                        fill-pen fill-color
                        grad-pen radial viewport-box/center 0 radius area-fill
                        circle viewport-box/center radius
                ]
                actors: [
                        on-make: [
                                face/facets/area-fill: select make-material/facet face '
                        ]
                        on-resize: [; arg is the size
                                do-actor/style face 'on-resize arg 'face ;handle default
                                face/facets/radius: (min arg/y arg/x) - 5 / 2
                        ]
                ]
        ]
]
view [
        circle red
        circle green
        circle blue
]
```

The `actors` block defines a few actor functions that are called from the GUI as needed. The first computes the area-fill when the face is created. The second handles resizing, recomputing the size of the face and the diameter of the circle.

Notice that within the actor functions, the face variables must be referenced via the face object itself.

You should note the use of `get-facet` for obtaining the area-color. This allows the area color to be obtained from the style facets or from the face facets, depending on if it was changed in the options line. This is the general method for obtaining the attributes of a style.

6. View All Styles

.

Note

This section is under construction and the examples doesn’t work at the moment.

Here’s an example that’s a little bit extreme. It creates a three column window that shows all predefined viewable styles:

```
all-styles: find extract to-block guie/styles 2 'clicker
view repend [group 3] [all-styles]
```

We’ll let you run it for yourself to see the results (because the window is too large to show in this document.)

Here’s a more elaborate example that creates a list of the predefined styles, then let’s you view each one separately.

```
all-styles: find extract to-block guie/styles 2 'clicker
last-view: none
view/options [
        title "Pick a style:"
```

```
        text-list all-styles do [
                if last-view [unview last-view]
                style-name: pick all-styles value
                last-view: view/options reduce [
                        'title reform ["Example of a" style-name "style:"]
                        style-name
                ] [offset: 'center]
        ]
] [offset: 50x50 size: 200x400]
```

The `last-view` variable provides a way to close the prior windows, otherwise you'd end up with a many windows on top of each other.

7. Advanced Examples

A number of other example styles can be found in the source code to the GUI system. They range from simple styles of just a few lines (e.g. a button) to advanced styles that may require one or two pages (e.g. a scroller).

7.1. Progress Bar

A progress bar is just an output display bar that can be set from your program or from other GUI objects.

Here's an example using the predefined `progress` style:

```
view [
    prog: progress
    button "Set 50%" set 'prog 50%
]
```

Normally, you won't need to create your own progress bar style, but if you did here's an example of what it might look like:

```
stylize [
    my-progress: [
        about: "Progress bar."
        facets: [
            init-size: 200x22
            max-size: 1000x22
            edge-color: 96.96.96
            fill-color: 80.80.80.128
            bar-color: teal
            bar-size: 1x1 ; modified by the progress % value
            area-fill: bar-fill: none
        ]
        options: [
            bar-color: [tuple!]
            size: [pair!]
        ]
        draw: [
            pen edge-color
            line-width 1.5
            grad-pen 1x1 0 20 90 area-fill
            box 1x1 (viewport-box/bottom-right - 1.5) 3
            grad-pen 1x1 0 20 90 bar-fill
            box 1x1 bar-size 3
```

```
        ]
        actors: [
            on-make: [
                 face/facets/area-fill: select make-material/facet face 'piano 'fill-col
                 face/facets/bar-fill: select make-material/facet face 'piano 'bar-color
            ]
            on-set: [ ; arg: event
                ; Update the bar size from the face value.
                face/state/value: arg/2
                v: limit to percent! arg/2 0% 100%
                size: face/facets/viewport-box/bottom-right - 1.5
                face/facets/bar-size: as-pair v * size/x size/y
            ]
        ]
    ]
]
view [
    prog: my-progress
    button "Set 50%" set 'prog 50%
]
```

7.2. Slider bar (numeric value input)

Here's a much more advanced example that shows how the `slider` style was defined. A slider bar is an input device that for setting a value between 0% and 100%.

Here's an example using the predefined `slider` style:

```
view [
        slider attach 'prog
        prog: progress
]
```

Normally, you won't need to create your own slider style, but if you did here's an example of what it might look like:

```
stylize [
    my-slider: [
        about: "Slide-bar for numeric input (0% - 100%)"
        facets: [
            init-size: 200x22
            max-size: 1000x22
            edge-color: 96.96.96
            fill-color: 80.80.80
            axis: area-fill: none
            knob-color: red
            knob-xy:
            bias-xy: 8x0 ; pointer adjustment at ends
        ]
        options: [
            init-size: [pair!]
            knob-color: [tuple!]
        ]
        draw: [
            pen edge-color
            line-width .4
            grad-pen 1x1 0 4 90 area-fill
            box 1x1 viewport-box/bottom-right 3
            line-width 1.5
```

```
            fill-pen knob-color
            translate knob-xy
            triangle -6x16 0x2 6x16
        ]
        actors: [
            on-make: [
                face/facets/area-fill: select make-material/facet face 'piano 'fill-colo
            ]
            on-resize: [ ; arg: size
                do-style/style face 'on-resize arg 'face ;handle default resizing
                face/facets/axis: face-axis? face
                do-style face 'on-update none
            ]
            on-up:date: [
                ; Compute the knob offset from face/value:
                val: limit to percent! any [face/state/value 0] 0% 100%
                bias: face/facets/bias-xy
                size: face/facets/viewport-box/bottom-right - bias - bias
                face/facets/knob-xy: val * size * 1x0 + bias
            ]
            on-offset: [ ; arg: offset
                ; Compute face/value from knob offset:
                bias: face/facets/bias-xy
                arg:  max 0x0 arg - bias
                size: face/facets/viewport-box/bottom-right - bias - bias
                axis: pick [x y] 'y = get-facet face 'axis
                face/state/value: val: min 100%
                max 0% to-percent arg/:axis / size/:axis
                face/facets/knob-xy: val * size * 1x0 + bias
            ]
            on-click: [ ; arg: event
                do-style face 'on-offset arg/offset
                if arg/type = 'down [
                    draw-face face
                    return init-drag/only face arg/offset
                ]
                do-face face
                ; Click UP: compute percentage value from xy offset none
                ; handled event
            ]
            on-drag: [ ; arg: drag
                do-style face 'on-offset arg/delta + arg/base
                draw-face face
                do-face face
            ]
            on-set: [ ; arg: [field value]
                if all [
                    'value = first arg
                    number? second arg
                ][face/state/value: second arg]
                do-style face 'on-update none ; will clip value range
            ]
        ]
    ]
]

view [
    my-slider attach 'prog
    prog: progress
```

```
]
```

For details about the actor functions above, see the R3 GUI Actors section. Here is a quick summary of the ones used above:

Actor	Description
on-resize	changes the gob and area sizes, then calls update to recompute the knob location.
on-update	is a general update function that recomputes the knob location.
on-offset	is used during mouse input to convert the mouse position to the actual state value, which is a percentage between 0% and 100%. It also updates the knob location, just to save time.
on-click	handles the mouse button when it is clicked in the bar. First, the knob is moved to where the click occurred. Then, if the user starts to drag, the drag operation begins.
on-drag	takes care of the drag operation, updating the value as needed, depending on the location of the mouse.
on-set	handles setting the slider from another GUI object or from the program. Its main job is to make sure the value is valid, then update the slider.

Descriptions of various other functions can be found in the R3 GUI Faces section.