R3 GUI Faces

Author: Carl Sassenrath, Saphirion AG

Revision History
22-Apr-2011 A

Table of Contents

1. Concepts

A `face` is an instance of a `style`.

A style holds the default attributes, variables, and functions of a GUI element, but a face object stores the specific values for that `instance` of the style.

For example, a `button` style may specify a default size, but its actual size as displayed on screen is stored in its face object. Its current position as well as its "clicked" state are also stored in its face object.

Layouts are collections of related faces used for a specific part of the user interface. The faces within a layout are created during the `layout` stage of processing where the GUI language (a dialect) is interpreted and its styles create actual faces. A layout is itself a face, and a GUI is created from one or more layers of layouts, each of which holds faces of its own.

This method of design helps minimize memory requirements. The `static` variables of a graphical element are kept in its style object to be shared by many instances of that style. Only the `dynamic`

variables of the style need be stored in a face object. The style object also stores the code related to the face, which is reused for every face.

2. Using Faces

Faces are created during the construction of layout. The GUI dialect specifies the styles and attributes required for the GUI, and those are used to construct the faces of the layout.

For example, the GUI dialect code below uses the `view` function to create a single layout that is displayed in a window.

```
view [
        title "Example GUI"
        field
        button "Submit" submit
]
```

The `title`, `field`, and `button` styles are specified, and the result will be a layout containing three faces, each associated with those style.

Faces can also be created with the `make-face` function. This function takes a style and an attribute block as its arguments, and returns a new face object. This provides a functional method of creating GUIs (but, it's rarely used, because the GUI dialect is more concise.)

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

In order to be useful, this new face must be inserted into a layout to be displayed. See the R3 GUI Layouts and for details.

2.1. Primary Functions

These functions are the most common for faces:

Function	Description
view	Create a layout from a dialect block, and display it in a window.
unview	Close a window opened with view.
layout	Create a layout of faces. The layout can be displayed later.
make-face	Creates a new face object based on a given style with the given attributes.
get-face	Get the primary value of the face, or a specific state variable of the face.
set-face	Set the primary value of the face, or a specific state variable of the face, then redraw the face.

2.2. Advanced Functions

These functions are used mainly for style implementation or to create special results within a GUI.

Function	Description
get-facet	Get a named facet of a face. If not present in the face, get the value from the style of the face. Multiple values can be fetch at the same time in a block.
set-facet	Set a named facet of a face. If not defined, create it.
do-face	Evaluate the reactor functions for a face. This is provided to allow reactor code from one face to relay actions to another face.
draw-face	Generate the rendering block for a face, and refresh the face on screen.
extend-face	Add a new field to the face. This is used by special styles during face creation.
do-style	Call a style actor function for the face. Used to decouple the face, style, and actor.
has-actor?	Return true if the face's style supports the given actor.
do-related	If there are related faces within the same parent layout, call their related actors. This is how faces inter-connect.
find-face-actor	Find the next (or prior) face that responds to a given actor. This is also used to inter-connect faces by nearness.

2.3. Face Names

Each face can have a name. Within a layout, it is useful to refer to a face by its name. For example, a layout that requests a user name and email address might need to refer to those faces within its code.

```
name: field
email: field
button "Submit" do [
        unless verify get-face name [warn "invalid name"]
        unless verify get-face email [warn "invalid email address"]
]
```

The `name` and `email` words are layout-local variables that refer to their respective face objects.

Here's another example that attaches a slider to a progress bar. The `prog` name is used by the slider to refer to the progress bar face and change it as necessary:

```
text "Drag slider to see progress bar change:"
slider attach 'prog
prog: progress
```

The more about face names is provided below and in the R3 GUI Layouts page.

2.4. Face Values

Faces can have one or more `values`. In the example above, both the `slider` and the `progress` faces hold a percentage as their value.

The get-face and set-face functions get and set the values of faces. Because the majority of faces have only a single `primary value` of interest, get-face and set-face refer to the value by default.

For example, you can write:

```
slid: slider
button "Show" do [print get-face slid]
button "Full" do [set-face slid 100%]
button "Empty" do [set-face slid 0%]
```

However, it should be noted that there is an even easier way to set the slider that does not require the set-face function:

```
button "Full" set 'slid 100%
button "Empty" set 'slid 0%
```

These face reactors are shorthand for the set-face function.

3. Face Object

Users don’t need to know much about the structure of the face object just to create GUI’s. However, if you plan to implement your own styles, you’ll need to know a few things about faces and how they store information related to the style.

Some styles may require more storage than others; therefore, face objects are extensible (vary in size.) This minimizes the memory required for faces that do not require extra fields.

Every face must have these standard fields:

Field	Datatype	Description
style	word!	The name of the style for this face.
facets	object!	Attributes that are unique to this face.
state	object!	Storage for state-related variables (not attributes).
gob	gob!	The graphical object used to display this face.
options	object!	Optional attributes specified for this instance.

Depending on how the face was defined, these fields may also be present:

Field	Datatype	Description

name	word!	A name used to refer to this specific face. (For example, to get or set its value.)
reactors	block!	A block of user actions for this face.
intern	object!	A context that holds *static* values shared between all face instances of the style.

Some styles may add other fields. For example, a layout adds these fields:

Field	Datatype	Description
facets/intern	object!	Internal parameters used for layout calculations.
trigger-faces	block!	Trigger faces needed for the layout.

The next few sections are detailed descriptions of the fields above.

3.1. Style Field

The `style` field just holds the name of the style, not a direct reference to the style object itself. It's done this way (as a word) to avoid long listings during debugging.

3.2. Facets Field

The `facets` object stores attributes that are unique to the instance of the face. That is, it only holds attribute values that are different from the style itself or that may change based on variations in the state of the object (for example a change to a color when the mouse is hovering over a button.)

For example, a button might have these face facets:

```
init-size: 130x24
pen-color: 100.50.50 ; set by on-draw
area-fill: 200.200.200 ; set by on-draw
```

.

Note

Note that the face/facets field is merged with the style/facets field during the face creation. The face/facets always take precedence over the style/facets.

3.3. State Field

The `state` object provides storage for the non-attribute values of a style. For example, a button might keep track of its `over` and `selected` states. For a scroll-bar, the scroll value and delta percentages must be stored.

An example of the state object for a scroll-bar:

```
value: 0%   ; scroller offset
delta: 10%  ; page size increment
```

.

Note

Note that the state object is normally small, but can be extended as necessary for specific styles that require extra fields. If you need to store your own internal, non-attribute variables for a face, it's best to store them in this location.

3.4. GOB Field

The `gob` field refers to the lower level graphical object that implements face rendering. It is normally a GOB of the `draw` type; however, for special text sections, it can be a `text` GOB (richtext).

Note that within a face gob the `gob/data` field is a reference back to the face object itself. This allows the display and event sub-systems to process gobs efficiently.

Some faces will use a `face/gob/pane` to holds sub-gobs to implement various parts of their display. For example, a scroll bar contains several such sub-gobs for its various components. These sub-gobs may or may not be actual faces, depending on how the style is implemented.

It should be noted that the `gob/offset` and `gob/size` fields are only `approximations` of `the actual graphic image area`. This is due to the anti-aliased graphics engine which requires that the gob be large enough to include pixels that are part of the anti-aliased edges of the face.

3.5. Options Field

The `options` field holds the optional attributes that were provided to the face within the GUI dialect when the face was defined. For example, in the line:

```
button "Submit"
```

The submit string is an option, and it will be stored in the `face/options` object:

```
text-body: "Submit"
```

What gets stored in the options object is determined by the options field of the style.

It is permitted for the `face/options` object to be used to reset a face back to its initial value. For example, when resetting the fields of a form. If a style requires this ability, it should always copy its option series values in order to be able to restore them later.

.

Note

Note that face actions (reactors) are not stored in this options object.

3.6. Name Field

When a face is named, the `name` field holds that name (as a word). Such names can later be used to reference the face object from other faces and functions.

The name field comes from the set-word used in the dialect. For example, the area face created in this line:

```
summary: area
```

will have the name `summary`. The name can now be used to reset or clear the area:

```
button "Reset" reset 'summary
button "Clear" clear 'summary
```

For user coding convenience, the face name spans more than just the layout where it was defined. For example, this code works as you would expect, even though the buttons are part of a sub-layout, not the layout in which the field name was defined:

```
vpanel [
        user: field
        hpanel [
                button "Reset" reset 'user
                button "Clear" clear 'user
        ]
]
```

In case of compound styles where local name-space is neccessary, this can be done with the `style/facets/names` field. See the layouts document for more information.

3.7. Reactors Field

The `reactors` field is a block of names and values for special actions on a face, called reactors.

For example, the line below specifies a `reset` reactor with an argument of `summary`:

```
button "Reset" reset 'summary
```

This information is stored in the reactors block of the face. Multiple reactors may be specified.

Whenever specific events occur, the block is processed to perform various actions. For example, to evaluate code when a button is clicked.

4. Debugging Techniques

There will be times when you create a GUI but do not understand why it displays or acts a certain way. You can experiment around and try different methods, and often you'll find a good solution. At other times, it may get frustrating, and you're not sure what's going on.

Sometimes, you really want to know more about what's going on "under the hood". Fortunately, because the R3 GUI system is relatively simple. You can display more information about the GUI and gain an understanding what that information means.

4.1. Examining Faces

If you plan to write your own styles, you need to understand face objects fairly well. One easy way to learn about faces is to use the `make-face` function with different styles and options.

For example, here's the face that is created for a text style:

```
&gt;&gt; probe make-face 'text [text-body: "test"]
make object! [
    style: 'text
    facets: make object! [
        border-color: none
        border-size: [0x0 0x0]
        bg-color: none
        margin: [0x0 0x0]
        padding: [0x0 0x0]
        spacing: 0x0
        init-size: 400x20
        min-size: 0x0
        max-size: 3.402823e38x3.402823e38
        align: 'left
        valign: 'top
        resizes: true
        box-model: 'tight
        gob: make gob! [offset: 0x0 size: 100x100 alpha: 0]
        gob-size: none
        space: [0x0 0x0]
        margin-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        border-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        padding-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        viewport-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        text-body: "test"
        text-style: 'base
    ]
```

```
    state: make object! [
        mode: 'up
        over: false
        value: none
    ]
    gob: make gob! [offset: 0x0 size: 100x100 alpha: 0]
    options: make object! [
        text-body: "test"
    ]
    tags: make map! [
        info true
    ]
    intern: none
]
```

4.2. Face Debug

If you want to dump (display to the console) a face object created during a layout, just add the word debug on the line.

The GUI code:

```
button "Test" debug do [test: true]
```

will dump debug information to the console:

```
-- debug-face[button:make]: make object! [
    style: 'button
    facets: make object! [
        init-size: 130x24
        min-size: 24x24
        max-size: 260x24
        align: 'left
        valign: 'top
        resizes: true
        box-model: 'tight
        gob: make gob! [offset: 0x0 size: 100x100 alpha: 0]
        gob-size: none
        space: [0x0 0x0]
        margin-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        border-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        padding-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
```

```
        ]
        viewport-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        bg-color: 80.100.120
        border-color: 0.0.0.128
        pen-color: 'chrome
        area-fill: 'chrome
        material: 'chrome
        focus-color: 255.0.0
        draw-mode: 'normal
        materials: none
        face-width: none
        text: "Test"
        text-style: 'button
        text-size-pad: 20x0
    ]
    state: make object! [
        mode: 'up
        over: false
        value: none
        validity: none
    ]
    gob: make gob! [offset: 0x0 size: 100x100 alpha: 0]
    options: make object! [
        text: "Test"
    ]
    tags: make map! [
        action true
        tab true
    ]
    intern: none
    debug: [make]
]
```

Now you can review it and determine all of the facets and other variables are set as you would expect.

.

Note

Keep in mind that this dump occurs just after the face has been made and before any layout-related resizing occurs.

4.3. Red-line Mode

For face styles that render graphics (contain a DRAW), the face debug method above will also show up as a red box at the edges of the face's clip area (the GOB).

Example:

```
view [button "Test" debug do [test: true]]
```

Notes:

- If your face draws or fills the entire GOB clip area, the red line will not be seen. (It is underneath the draw graphics.)

- Styles that dynamically modify their draw blocks using `on-draw` should be aware that the face debug mode will copy and modify the draw block each time it is rendered (in order to add the red box).

4.4. Style Debug

When defining new styles you can examine the object that is created in the processby adding a debug field to the style definition:

```
stylize [
        thing: [
                facets: [
                        init-size: 100x100
                        bg-color: blue
                ]

                draw: [
                        circle
                ]

                debug: [style]
        ]
]
```

When the "thing" style is defined, you will see:

```
-- debug-style [thing]: make object! [
    name: 'thing
    facets: make object! [
        init-size: 100x100
        min-size: 0x0
        max-size: 3.402823e38x3.402823e38
        align: 'left
        valign: 'top
        resizes: true
        box-model: 'tight
        gob: none
        gob-size: none
        space: [0x0 0x0]
        margin-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        border-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        padding-box: [
```

```
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        viewport-box: [
            top-left: 0x0
            top-right: 0x0
            bottom-left: 0x0
            bottom-right: 0x0
            center: 0x0
        ]
        bg-color: 0.0.255
    ]
    draw: [
        circle
    ]
    actors: make map! [
        locate: make function! [[face arg
            /local
        ][
            arg/offset
        ]]
        on-resize: make function! [[face arg
            /local
        ][
            if any [face/facets/resizes face/gob/size &lt;&gt; 0x-1] [
                if all [
                    in face/facets 'intern
                    in face/facets/intern 'update?
                    face/facets/intern/update?
                ] [do-style face 'on-update none]
                set-draw-keywords-in face/facets arg
            ]
            unless face/gob/size = 0x-1 [face/gob/size: face/facets/gob-size]
        ]]
        on-get: make function! [[face arg
            /local
        ][
            select face/state arg
        ]]
    ]
    tags: make map! [
    ]
    options: make object! [
    ]
    parent: none
    state: none
    intern: none
    content: none
    about: "Not documented."
    debug: [style]
]
```

Note that you can also use this same method to output the face object each time it is used in a layout. To do so change the debug line to:

```
debug: [style make]
```

4.5. Runtime Debug

You can globally enable debugging throughout the GUI system for monitoring various functions and events.

The GUI module object includes a debug field that can be set to a block of debug flag words. They are:

Flag	Description
make-style	when stylize creates a new style
do-style	when a style actor is invoked
draw	when a draw block is reduced
events	to monitor events as they occur
dialect	to watch layout GUI dialect

`These can always be found simply by scanning source modules for "debug-gui" calls.`

For example, the line:

```
guie/debug: [events]
```

will show all events. Output will look like this:

```
-- events move 336x102
-- events move 320x93
-- events move 301x85
-- events move 285x83
-- events move 268x83
```

4.6. Debug Functions

Two internal debug functions can be called from your code:

Function	Description
dump-face	print information about a face
dump-layout	print information about a layout

An example of `dump-face` is:

```
view [
        example: button "Test" do [dump-face face]
]
```

When you click on the button, you will see:

```
button: 5x108 size: 221x24 example "Test"
```

This shows the button style location, size, name, and data.

An example of `dump-layout` is:

```
view [
        pan: hpanel [
                button "Dump" do [dump-layout pan]
                button "Quit" quit
        ]
]
```

and it will output:

```
hpanel: 5x405 size: 271x30 pan "*"
button: 3x3 size: 130x24 * "Dump"
button: 138x3 size: 130x24 * "Quit"
```

It shows the layout face and all of its sub-faces, even if they are sub-layouts.

The asterisk means that the field has not been set.