SUIT Documentation

Release 1.0

Matthias Richter

Jun 21, 2018

Contents

Simple User Interface Toolkit for LÖVE.

CHAPTER 1

SUIT up

You can download SUIT and view the code on github: vrld/SUIT. You may also download the sourcecode as a zip or tar file.

Otherwise, use Git:

```
git clone git://github.com/vrld/SUIT
```

Update:

```
git pull
```

Hello, Suit:

```
suit = require 'suit'

local show_message = false
function love.update(dt)
    -- Put a button on the screen. If hit, show a message.
    if suit.Button("Hello, World!", 100,100, 300,30).hit then
        show_message = true
    end

    -- if the button was pressed at least one time, but a label below
    if show_message then
        suit.Label("How are you today?", 100,150, 300,30)
    end
end

function love.draw()
    suit.draw()
end
```

CHAPTER 2

Read on

2.1 Getting Started

Before actually getting started, it is important to understand the motivation and mechanics behind SUIT:

- **Immediate mode is better than retained mode**
- **Layout should not care about content**
- **Less is more**

2.1.1 Immediate mode?

With classical (retained) mode libraries you typically have a stage where you create the whole UI when the program initializes. This includes what happens when events like button presses or slider changes occur. After that point, the GUI is expected to not change very much. This is great for word processors where the interaction is consistent and straightforward, but bad for games, where everything changes all the time.

With immediate mode libraries, on the other hand, the GUI is created every frame from scratch. Because that would be wasteful, there are no widget objects. Instead, widgets are created by functions that react to UI state and present some data. Where this data comes from and how it is maintained does not concern the widget at all. This is, after all, your job. This gives great control over what is shown where and when. The widget code can be right next to the code that does what should happen if the widget state changes. The layout is also very flexible: adding a widget is one more function call, and if you want to hide a widget, you simply don't call the corresponding function.

This separation of data and behaviour is great when a lot of stuff is going on, but takes a bit of time getting used to.

What SUIT is

SUIT is simple: It provides only a few basic widgets that are important for games:

- *Buttons* (including *Image Buttons*)
- *Text Labels*

- *Checkboxes*
- *Text Input*
- *Value Sliders*

SUIT is comfortable: It has a straightforward, yet effective row/column-based layout engine.

SUIT is adaptable: It is possible to change the color scheme, single drawing functions for all widget or the whole theme.

SUIT is hackable: Custom widgets can leverage the extensive *core library*.

SUIT is good at games!

What SUIT is not

SUIT is not a complete GUI library: It does not provide dropdowns, table views, menu bars, modal dialogs, etc.

SUIT is not a complete GUI library: It does not have a markup language or tools to design a user interface.

SUIT is not a complete GUI library: It does not take control of the runtime. You have to do everything yourself[1].

SUIT is not good at processing words!

2.1.2 Hello, World!

SUITing up is is straightforward: Define your GUI in `love.update()`, and draw it in `love.draw()`:

```
suit = require 'suit'

local show_message = false
function love.update(dt)
    -- Put a button on the screen. If hit, show a message.
    if suit.Button("Hello, World!", 100,100, 300,30).hit then
        show_message = true
    end

    -- if the button was pressed at least one time, but a label below
    if show_message then
        suit.Label("How are you today?", 100,150, 300,30)
    end
end

function love.draw()
    suit.draw()
end
```

This will produce this UI:

The two widgets (the button and the label) are each created by a function call (*suit.Button* and *suit.Label*). The first argument to a widget function always defines the *payload* of the widget. Different widgets expect different payloads. Here, both *suit.Button* and *suit.Label* expect a string. The last four arguments of a widget function define the position and dimension of the widget. The function returns a table that indicates the UI state of the widget. Here, the state `hit` is used to figure out if the mouse was clicked and released on the button. See *Widgets* for more info on widget states.

[1] But it thinks you can handle that.

2.1.3 Mutable state

Widgets that mutate some state - input boxes, checkboxes and sliders - expect a table as their payload, e.g.:

```
local slider = {value = 1, min = 0, max = 2}
function love.update(dt)
    suit.Slider(slider, 100,100, 200,20)
    suit.Label(tostring(slider.value), 300,100, 200,20)
end
```

The widget function updates the payload when some user interaction occurs. In the above example, `slider.value` may be changed by the *`Slider()`* widget. The value is then shown by a *`Label()`* next to the slider.

2.1.4 Options

You can define optional, well, options after the payload. Most options affect how the widget is drawn. For example, to align the label text to the left:

```
local slider = {value = 1, max = 2}
function love.update(dt)
    suit.Slider(slider, 100,100, 200,30)
    suit.Label(tostring(slider.value), {align = "left"}, 300,100, 200,30)
end
```

What options are available and what they are doing depends on the widget and the theme. See *Widgets* for more info on widget options.

2.1.5 Keyboard input

The *`Input()`* widget requires that you forward the `keypressed` and `textinput` events to SUIT:

```
local input = {text = ""}
function love.update(dt)
    suit.Input(input, 100,100,200,30)
    suit.Label("Hello, "..input.text, {align="left"}, 100,150,200,30)
end

-- forward keyboard events
function love.textinput(t)
    suit.textinput(t)
end

function love.keypressed(key)
    suit.keypressed(key)
end
```

The *`Slider()`* widget can also react to keyboard input. The mouse state is automatically updated, but you can provide your own version of reality if you need to. See the *Core functions* for more details.

2.1.6 Layout

It is tedious to calculate the position and size of each widget you want to put on the screen. Especially when all you want is to put three buttons beneath each other. SUIT implements a simple, yet effective layout engine. All the engine does is put cells next to each other (below or right). It does not care what you put into those cells, but assumes that you probably need them for widgets. Cells are reported by four numbers (left, top, width and height) that you can directly pass as the final four arguments to the widget functions. If you have ever dabbled with Qt's `QBoxLayout`, you already know 89%[2] of what you need to know.

Hello, World! can be rewritten as follows:

```
suit = require 'suit'

local show_message = false
function love.update(dt)
    -- put the layout origin at position (100,100)
    -- cells will grow down and to the right of the origin
    -- note the colon syntax
    suit.layout:reset(100,100)

    -- put 10 extra pixels between cells in each direction
    suit.layout:padding(10,10)

    -- construct a cell of size 300x30 px and put the button into it
    if suit.Button("Hello, World!", suit.layout:row(300,30)).hit then
        show_message = true
    end

    -- add another cell below the first cell
    -- the size of the cell is the same as the first cell
    if show_message then
        suit.Label("How are you today?", suit.layout:row())
    end
end

function love.draw()
    suit.draw()
end
```

At the beginning of each frame, the layout origin (and some internal layout state) has to be reset. You can also define optional padding between cells. Cells are added using `layout:row(w,h)` (which puts the new cell below the old cell) and `layout:col(w,h)` (which puts the new cell to the right of the old cell). If omitted, the width and height of the new cell are copied from the old cell. There are also special identifiers that calculate the size from the sizes of all cells that were created since the last `reset()`: `max`, `min` and `median`. They do what you expect them to do.

It is also possible to nest cells and to let cells dynamically fill the available space (but you have to tell how much space there is beforehand). Refer to the *Layout* documentation for more information.

2.1.7 Widget ids

Each widget is identified by an `id`[4]. Internally, this `id` is used to figure out which widget should handle user input like mouse clicks and keyboard presses. Unless specified otherwise, the `id` is the same as the payload, i.e., the `id`

[2] Proportion determined by rigorous scientific experiments[3].
[4] Welcome to the tautology club!

of `Button("Hello, World!", ...)` will be the string `"Hello, World!"`. In almost all of the cases, this will work fine and you don't have to worry about this `id` business.

Well, almost. Problems arise when two widgets share the same id, like here:

```
local suit = require 'suit'

function love.update()
    suit.layout:reset(100, 100)
    suit.layout:padding(10)

    if suit.Button("Button", suit.layout:row(200, 30)).hit then
        love.graphics.setBackgroundColor(255,255,255)
    end
    if suit.Button("Button", suit.layout:row()).hit then
        love.graphics.setBackgroundColor(0,0,0)
    end
end

function love.draw()
    suit:draw()
end
```

If the first button is hovered, both buttons will be highlighted, and if it pressed, both actions will be carried out. Hovering the second button will not affect the first, and clicking it will highlight both buttons, but only execute the action of the second button[5].

Luckily, there is a fix: you can specify the `id` of any widget using the `id` option, like so:

```
local suit = require 'suit'

function love.update()
    suit.layout:reset(100, 100)
    suit.layout:padding(10)

    if suit.Button("Button", {id=1}, suit.layout:row(200, 30)).hit then
        love.graphics.setBackgroundColor(255,255,255)
    end
    if suit.Button("Button", {id=2}, suit.layout:row()).hit then
        love.graphics.setBackgroundColor(0,0,0)
    end
end

function love.draw()
    suit:draw()
end
```

Now, events from one button will not propagate to the other. Here, the both `id` s are numbers, but you can use any Lua value except `nil` and `false`.

[5] Immediate mode is to blame: When the second button is processed, the first one is already fully evaluated. Time can not be reversed, not even by love.

2.1.8 Themeing

SUIT lets you customize how any widget (except *ImageButton()*) is drawn. Each widget (except, *you know*) is drawn by a function in the table `suit.theme`. Conveniently, the name of the function responsible for drawing a widget is named after it, so, a button is drawn by the function `suit.theme.Button`. If you want to change how a button is drawn, simply overwrite the function. If you want to redecorate completely, it might be easiest to start from scratch and swap the whole table.

However, if you just don't like the colors, the default theme is open to change. It requires you to change the background (`bg`) and foreground (`fg`) color of three possible widget states: `normal`, when nothing out of the ordinary happened, `hovered`, when the mouse hovers above a widget, and `active`, when the mouse hovers above, and the mouse button is pressed (but not yet released) on the widget. The colors are saved in the table `suit.theme.color`. The default color scheme is this:

```
suit.theme.color = {
    normal  = {bg = { 66, 66, 66}, fg = {188,188,188}},
    hovered = {bg = { 50,153,187}, fg = {255,255,255}},
    active  = {bg = {255,153,  0}, fg = {225,225,225}}
}
```

You can also do minimally invasive surgery:

```
function love.load()
    suit.theme.color.normal.fg = {255,255,255}
    suit.theme.color.hovered = {bg = {200,230,255}, fg = {0,0,0}}
end
```

2.1.9 GUI Instances

Sometimes you might feel the need to separate parts of the GUI. Maybe certain should always be drawn before or after other UI elements, or maybe you don't want the UI state to "leak" (e.g., from a stacked pause gamestate to the main gamestate).

For this reason, SUIT allows you to create GUI instances:

```
local dress = suit.new()
```

The IO and layout state of `dress` is totally contained in the instance and does not affect any other instances (including the "global" instance `suit`). In particular, `suit.draw()` will not draw anything from `dress`. Luckily, you can do that yourself:

```
dress:draw()
```

Notice that instances require that you use the colon syntax. This is true for every *core* <*core*> function as well as the widgets. To create a button, for example, you have to write:

```
dress:Button("Click?", dress.layout:row())
```

Instance Theme

Unlike UI and layout state, themes **are** shared among instances. The reason is that the `suit.theme` and `dress.theme` are **references**, and point to the same table (unless you make either of them point somewhere else). Usually this is a feature, but please still consider this

Warning: Changes in a shared theme will be shared across GUI instances.

If this is an issue—for example because you only want to change the color scheme of an instance—you can either deep-copy the theme table or use some metatable magic:

```
dress.theme = setmetatable({}, {__index = suit.theme})

-- NOTE: you have to replace the whole color table. E.g., replacing only
--       dress.theme.color.normal will also change suit.theme.color.normal!
dress.theme.color = {
    normal   = {bg = {188,188,188}, fg = { 66, 66, 66}},
    hovered  = {bg = {255,255,255}, fg = { 50,153,187}},
    active   = {bg = {255,255,255}, fg = {225,153,  0}}
}

function dress.theme.Label(text, opt, x,y,w,h)
    -- draw the label in a fancier way
end
```

2.2 Widgets

Note: Still under construction...

2.2.1 Immutable Widgets

Button (*text*[, *options*], *x*, *y*, *w*, *h*)

Arguments

- **text** (*string*) – Button label.
- **options** (*table*) – Optional settings (see below).
- **x,y** (*numbers*) – Upper left corner of the widget.
- **w,h** (*numbers*) – Width and height of the widget.o

Returns Return state (see below).

Creates a button widget at position `(x,y)` with width `w` and height `h`.

Label (*text*[, *options*], *x*, *y*, *w*, *h*)

Arguments

- **text** (*string*) – Label text.
- **options** (*table*) – Optional settings (see below).
- **x,y** (*numbers*) – Upper left corner of the widget.
- **w,h** (*numbers*) – Width and height of the widget.o

Returns Return state (see below).

Creates a label at position `(x,y)` with width `w` and height `h`.

ImageButton (*normal*, *options*, *x*, *y*)

Arguments

- **normal** (*mixed*) – Image of the button in normal state.
- **options** (*table*) – Widget options.
- **x,y** (*numbers*) – Upper left corner of the widget.

Returns Return state (see below).

Creates an image button widget at position `(x,y)`. Unlike all other widgets, an `ImageButton` is not affected by the current theme. The argument `normal` defines the image of the normal state as well as the area of the widget. The button activates when the mouse enters the area occupied by the widget. If the option `mask` defined, the button activates only if the mouse is over a pixel with non-zero alpha. You can provide additional `hovered` and `active` images, but the widget area is always computed from the `normal` image. You can provide additional `hovered` and `active` images, but the widget area is always computed from the `normal` image.

Additional Options:

mask Alpha-mask of the button, i.e. an `ImageData` of the same size as the `normal` image that has non-zero alpha where the button should activate.

normal Image for the normal state of the widget. Defaults to widget payload.

hovered Image for the hovered state of the widget. Defaults to `normal` if omitted.

active Image for the active state of the widget. Defaults to `hovered` if omitted.

Note: `ImageButton` does not recieve width and height parameters. As such, it does not necessarily honor the cell size of a *Layout*.

Note: Unlike other widgets, `ImageButton` is tinted by the currently active color. If you want the button to appear untinted, make sure the active color is set to white before adding the button, e.g.:

```
love.graphics.setColor(255,255,255)
suit.ImageButton(push_me, {hovered=and_then_just, active=touch_me},
                 suit.layout:row())
```

2.2.2 Mutable Widgets

Checkbox (*checkbox*[, *options*], *x*, *y*, *w*, *h*)

Arguments

- **checkbox** (*table*) – Checkbox state.
- **options** (*table*) – Optional settings (see below).
- **x,y** (*numbers*) – Upper left corner of the widget.
- **w,h** (*numbers*) – Width and height of the widget.o

Returns Return state (see below).

Creates a checkbox at position `(x,y)` with width `w` and height `h`.

State:

`checkbox` is a table with the following components:

`checked` `true` if the checkbox is checked, `false` otherwise.

`text` Optional label to show besides the checkbox.

`Slider`(*slider*[, *options*], *x*, *y*, *w*, *h*)

Arguments

- **`slider`** (*`table`*) – Slider state.
- **`options`** (*`table`*) – Optional settings (see below).
- **`x,y`** (*`numbers`*) – Upper left corner of the widget.
- **`w,h`** (*`numbers`*) – Width and height of the widget.o

Returns Return state (see below).

Creates a slider at position `(x,y)` with width `w` and height `h`. Sliders can be horizontal (default) or vertical.

State:

`value` Current value of the slider. Mandatory argument.

`min` Minimum value of the slider. Defaults to `min(value, 0)` if omitted.

`max` Maximum value of the slider. Defaults to `min(value, 1)` if omitted.

`step` Value stepping for keyboard input. Defaults to `(max - min)/10` if omitted.

Additional Options:

`vertical` Whether the slider is vertical or horizontal.

Additional Return State:

`changed` `true` when the slider value was changed, `false` otherwise.

`Input`(*input*[, *options*], *x*, *y*, *w*, *h*)

Arguments

- **`input`** (*`table`*) – Checkbox state
- **`options`** (*`table`*) – Optional settings (see below).
- **`x,y`** (*`numbers`*) – Upper left corner of the widget.
- **`w,h`** (*`numbers`*) – Width and height of the widget.o

Returns Return state (see below).

Creates an input box at position `(x,y)` with width `w` and height `h`. Implements typical movement (arrow keys, home and end key) and editing (deletion with backspace and delete) facilities.

State:

`text` Current text inside the input box. Defaults to the empty string if omitted.

`cursor` Cursor position. Defined as the position before the character (including EOS), so `1` is the position before the first character, etc. Defaults to the end of `text` if omitted.

Additional Return State:

`submitted` `true` when enter was pressed while the widget has keyboard focus.

2.2.3 Common Options

id Identifier of the widget regarding user interaction. Defaults to the first argument (e.g., `text` for buttons) if omitted.

font Font of the label. Defaults to the current font (`love.graphics.getFont()`).

align Horizontal alignment of the label. One of `"left"`, `"center"`, or `"right"`. Defaults to `"center"`.

valign Vertical alignment of the label. On of `"top"`, `"middle"`, or `"bottom"`. Defaults to `"middle"`.

color A table to overwrite the color. Undefined colors default to the theme colors.

cornerRadius The corner radius for boxes. Overwrites the theme corner radius.

draw A function to replace the drawing function. Refer to *Themeing* for more information about the function signatures.

2.2.4 Common Return States

id Identifier of the widget.

hit `true` if the mouse was pressed and released on the button, `false` otherwise.

hovered `true` if the mouse is above the widget, `false` otherwise.

entered `true` if the mouse entered the widget area, `false` otherwise.

left `true` if the mouse left the widget area, `false` otherwise.

2.3 Layout

Note: Still under construction...

2.3.1 Immediate Mode Layouts

reset([*x*, *y*[, *pad_x*[, *pad_y*]]])

Arguments

- **x,y** (*numbers*) – Origin of the layout (optional).
- **pad_x,pad_y** – Cell padding (optional).

Reset the layout, puts the origin at `(x,y)` and sets the cell padding to `pad_x` and `pad_y`.

If `x` and `y` are omitted, they default to `(0,0)`. If `pad_x` is omitted, it defaults to 0. If `pad_y` is omitted, it defaults to `pad_x`.

padding([*pad_x*[, *pad_y*]])

Arguments

- **pad_x** – Cell padding in x direction (optional).
- **pad_y** – Cell padding in y direction (optional, defaults to `pad_x`).

Returns The current (or new) cell padding.

Get and set the current cell padding.

If given, sets the cell padding to `pad_x` and `pad_y`. If only `pad_x` is given, set both padding in `x` and `y` direction to `pad_x`.

size()

Returns `width,height` - The size of the last cell.

Get the size of the last cell.

nextRow()

Returns `x,y` - Upper left corner of the next row cell.

Get the position of the upper left corner of the next cell in a row layout. Use for mixing precomputed and immediate mode layouts.

nextCol()

Returns `x,y` - Upper left corner of the next column cell.

Get the position of the upper left corner of the next cell in a column layout. Use for mixing precomputed and immediate mode layouts.

push([*x*, *y*])

Arguments

- **x,y** (*numbers*) – Origin of the layout (optional).

Saves the layout state (position, padding, sizes, etc.) on a stack, resets the layout with position `(x,y)`.

If `x` and `y` are omitted, they default to `(0,0)`.

Used for nested row/column layouts.

pop()

Restores the layout parameters from the stack and advances the layout position according to the size of the popped layout.

Used for nested row/column layouts.

row(*w*, *h*)

Arguments

- **w,h** (*mixed*) – Cell width and height (optional).

Returns Position and size of the cell: `x,y,w,h`.

Creates a new cell below the current cell with width `w` and height `h`. If either `w` or `h` is omitted, the value is set the last used value. Both `w` and `h` can be a string, which takes the following meaning:

max Maximum of all values since the last reset.

min Mimimum of all values since the last reset.

median Median of all values since the last reset.

Used to provide the last four arguments to a widget, e.g.:

```
suit.Button("Start Game", suit.layout:row(100,30))
suit.Button("Options", suit.layout:row())
suit.Button("Quit", suit.layout:row(nil, "median"))
```

down(*w*, *h*)

An alias for *layout:row()*.

col (*w*, *h*)

Arguments

- **w,h** (*mixed*) – Cell width and height (optional).

Returns Position and size of the cell: `x,y,w,h`.

Creates a new cell to the right of the current cell with width `w` and height `h`. If either `w` or `h` is omitted, the value is set the last used value. Both `w` and `h` can be a string, which takes the following meaning:

max Maximum of all values since the last reset.

min Mimimum of all values since the last reset.

median Median of all values since the last reset.

Used to provide the last four arguments to a widget, e.g.:

```
suit.Button("OK", suit.layout:col(100,30))
suit.Button("Cancel", suit.layout:col("max"))
```

right (*w*, *h*)

An alias for *layout:col()*.

up (*w*, *h*)

Arguments

- **w,h** (*mixed*) – Cell width and height (optional).

Returns Position and size of the cell: `x,y,w,h`.

Creates a new cell above the current cell with width `w` and height `h`. If either `w` or `h` is omitted, the value is set the last used value. Both `w` and `h` can be a string, which takes the following meaning:

max Maximum of all values since the last reset.

min Mimimum of all values since the last reset.

median Median of all values since the last reset.

Be careful when mixing `up()` and *layout:row()*, as suit does no checking to make sure cells don't overlap. e.g.:

```
suit.Button("A", suit.layout:row(100,30))
suit.Button("B", suit.layout:row())
suit.Button("Also A", suit.layout:up())
```

left (*w*, *h*)

Arguments

- **w,h** (*mixed*) – Cell width and height (optional).

Returns Position and size of the cell: `x,y,w,h`.

Creates a new cell to the left of the current cell with width `w` and height `h`. If either `w` or `h` is omitted, the value is set the last used value. Both `w` and `h` can be a string, which takes the following meaning:

max Maximum of all values since the last reset.

min Mimimum of all values since the last reset.

median Median of all values since the last reset.

Be careful when mixing `left()` and *layout:col()*, as suit does no checking to make sure cells don't overlap. e.g.:

```
suit.Button("A", suit.layout:col(100,30))
suit.Button("B", suit.layout:col())
suit.Button("Also A", suit.layout:left())
```

2.3.2 Precomputed Layouts

Apart from immediate mode layouts, you can specify layouts in advance. The specification is a table of tables, where each inner table follows the convention of *row()* and *col()*. The result is a layout definition object that can be used to access the cells.

There are almost only two reasons to do so: (1) You know the area of your layout in advance (say, the screen size), and want certain cells to dynamically fill the available space; (2) You want to animate the cells.

Note: Unlike immediate mode layouts, precomputed layouts **can not be nested**. You can mix immediate mode and precomputed layouts to achieve nested layouts with precomputed cells, however.

Layout Specifications

Layout specifications are tables of tables, where the each inner table corresponds to a cell. The inner tables define the width and height of the cell according to the rules of *row()* and *col()*, with one additonal keyword:

`fill` Fills the available space, determined by `min_height` or `min_width` and the number of cells with property `fill`.

For example, this row specification makes the height of the second cell to `(300 - 50 - 50) / 1 = 200`:

```
{min_height = 300,
    {100, 50},
    {nil, 'fill'},
    {nil, 50},
}
```

This column specification divides the space evenly among two cells:

```
{min_width = 300,
    {'fill', 100}
    {'fill'}
}
```

Apart from `min_height` and `min_width`, layout specifications can also define the position (upper left corner) of the layout using the `pos` keyword:

```
{min_width = 300, pos = {100,100},
    {'fill', 100}
    {'fill'}
}
```

You can also define a padding:

```
{min_width = 300, pos = {100,100}, padding = {5,5},
    {'fill', 100}
```

(continues on next page)

(continued from previous page)

```
    {'fill'}
}
```

Layout Definition Objects

Once constructed, the cells can be accessed in two ways:

- Using iterators:

```
for i, x,y,w,h in definition() do
    suit.Button("Button "..i, x,y,w,h)
end
```

- Using the `cell(i)` accessor:

```
suit.Button("Button 1", definition.cell(1))
suit.Button("Button 3", definition.cell(3))
suit.Button("Button 2", definition.cell(2))
```

There is actually a third way: Because layout definitions are just tables, you can access the cells directly:

```
local cell = definition[1]
suit.Button("Button 1", cell[1], cell[2], cell[3], cell[4])
-- or suit.Button("Button 1", unpack(cell))
```

This is especially useful if you want to animate the cells, for example with a tween:

```
for i,cell in ipairs(definition)
    local destination = {[2] = cell[2]} -- save cell y position
    cell[2] = -cell[4] -- move cell just outside of the screen

    -- let the cells fall into the screen one after another
    timer.after(i / 10, function()
        timer.tween(0.7, cell, destination, 'bounce')
    end)
end
```

Constructors

rows (*spec*)

Arguments

- **spec** (*table*) – Layout specification.

Returns Layout definition object.

Defines a row layout.

cols (*spec*)

Arguments

- **spec** (*table*) – Layout specification.

Returns Layout definition object.

Defines a column layout.

2.4 Core Functions

The core functions can be divided into two parts: Functions of interest to the user and functions of interest to the (widget) developer.

2.4.1 External Interface

Drawing

draw()

Draw the GUI - call in `love.draw`.

theme

The current theme. See *Themeing*.

Mouse Input

updateMouse(*x*, *y*, *buttonDown*)

Arguments

- **x, y** (*number*) – Position of the mouse.
- **buttonDown** (*boolean*) – Whether the mouse button is down.

Update mouse position and button status. You do not need to call this function, unless you use some screen transformation (e.g., scaling, camera systems, ...).

Keyboard Input

keypressed(*key*)

Arguments

- **key** (*KeyConstant*) – The pressed key.

Forwards a `love.keypressed(key)` event to SUIT.

textinput(*char*)

Arguments

- **char** (*string*) – The pressed character

Forwards a `love.textinput(key)` event to SUIT.

GUI State

anyHovered()

Returns `true` if any widget is hovered by the mouse.

Checks if any widget is hovered by the mouse.

isHovered(*id*)

Arguments

- **id** (*mixed*) – Identifier of the widget.

Returns `true` if the widget is hovered by the mouse.

Checks if the widget identified by `id` is hovered by the mouse.

wasHovered(*id*)

Arguments

- **id** (*mixed*) – Identifier of the widget.

Returns `true` if the widget was in the hovered by the mouse in the last frame.

Checks if the widget identified by `id` was hovered by the mouse in the last frame.

anyActive()

Returns `true` if any widget is in the `active` state.

Checks whether the mouse button is pressed and held on any widget.

isActive(*id*)

Arguments

- **id** (*mixed*) – Identifier of the widget.

Returns `true` if the widget is in the `active` state.

Checks whether the mouse button is pressed and held on the widget identified by `id`.

anyHit()

Returns `true` if the mouse was pressed and released on any widget.

Check whether the mouse was pressed and released on any widget.

isHit(*id*)

Arguments

- **id** (*mixed*) – Identifier of the widget.

Returns `true` if the mouse was pressed and released on the widget.

Check whether the mouse was pressed and released on the widget identified by `id`.

2.4.2 Internal Helpers

getOptionsAndSize(...)

Arguments

- **...** (*mixed*) – Varargs.

Returns `options, x,y,w,h`.

Converts varargs to option table and size definition. Used in the widget functions.

registerDraw(*f*, ...)

Arguments

- **f** (*function*) – Function to call in `draw()`.
- **...** (*mixed*) – Arguments to f.

Registers a function to be executed during *draw()*. Used by widgets to make themselves visible.

enterFrame()

Prepares GUI state when entering a frame.

exitFrame()

Clears GUI state when exiting a frame.

Mouse Input

mouseInRect(*x*, *y*, *w*, *h*)

Arguments

- **x,y,w,h** (*numbers*) – Rectangle definition.

Returns `true` if the mouse cursor is in the rectangle.

Checks whether the mouse cursor is in the rectangle defined by `x,y,w,h`.

registerMouseHit(*id*, *ul_x*, *ul_y*, *hit*)

Arguments

- **id** (*mixed*) – Identifier of the widget.
- **ul_x, ul_y** (*numbers*) – Upper left corner of the widget.
- **hit** (*function*) – Function to perform the hit test.

Registers a hit-test defined by the function `hit` for the widget identified by `id`. Sets the widget to `hovered` if th hit-test returns `true`. Sets the widget to `active` if the hit-test returns `true` and the mouse button is pressed.

The hit test receives coordinates in the coordinate system of the widget, i.e. `(0,0)` is the upper left corner of the widget.

registerHitbox(*id*, *x*, *y*, *w*, *h*)

Arguments

- **id** (*mixed*) – Identifier of the widget.
- **x,y,w,h** (*numbers*) – Rectangle definition.

Registers a hitbox for the widget identified by `id`. Literally this function:

```
function registerHitbox(id, x,y,w,h)
    return registerMouseHit(id, x,y, function(u,v)
        return u >= 0 and u <= w and v >= 0 and v <= h
    end)
end
```

mouseReleasedOn(*id*)

Arguments

- **id** (*mixed*) – Identifier of the widget.

Returns `true` if the mouse was released on the widget.

Checks whether the mouse button was released on the widget identified by `id`.

getMousePosition()

Returns Mouse positon `mx, my`.

Get the mouse position.

Keyboard Input

getPressedKey()

Returns KeyConstant

Get the currently pressed key (if any).

grabKeyboardFocus(*id*)

Arguments

- **id** (*mixed*) – Identifier of the widget.

Try to grab keyboard focus. Successful only if the widget is in the `active` state.

hasKeyboardFocus(*id*)

Arguments

- **id** (*mixed*) – Identifier of the widget.

Returns `true` if the widget has keyboard focus.

Checks whether the widget identified by `id` currently has keyboard focus.

keyPressedOn(*id*, *key*)

Arguments

- **id** (*mixed*) – Identifier of the widget.
- **key** (*KeyConstant*) – Key to query.

Returns `true` if `key` was pressed on the widget.

Checks whether the key `key` was pressed while the widget identified by `id` has keyboard focus.

2.4.3 Instancing

new()

Returns Separate UI state.

Create a separate UI and layout state. Everything that happens in the new state will not affect any other state. You can use the new state like the "global" state `suit`, but call functions with the colon syntax instead of the dot syntax, e.g.:

```
function love.load()
    dress = suit.new()
end

function love.update()
    dress.layout:reset()
    dress:Label("Hello, World!", dress.layout:row(200,30))
    dress:Input(input, dress.layout:row())
end

function love.draw()
    dress:draw()
end
```

Warning: Unlike UI and layout state, the theme might be shared with other states. Changes in a shared theme will be shared across all themes. See the *Instance Theme* subsection in the *Getting Started* guide.

2.5 Themeing

Note: Under construction.

2.6 License

CHAPTER 3

Example code

The following code will create this UI:

```
local suit = require 'suit'

-- generate some assets (below)
function love.load()
    snd = generateClickySound()
    normal, hovered, active, mask = generateImageButton()
    smallerFont = love.graphics.newFont(10)
end

-- data for a slider, an input box and a checkbox
local slider= {value = 0.5, min = -2, max = 2}
local input = {text = "Hello"}
local chk = {text = "Check?"}

-- all the UI is defined in love.update or functions that are called from here
function love.update(dt)
    -- put the layout origin at position (100,100)
    -- cells will grown down and to the right from this point
    -- also set cell padding to 20 pixels to the right and to the bottom
    suit.layout:reset(100,100, 20,20)

    -- put a button at the layout origin
    -- the cell of the button has a size of 200 by 30 pixels
    state = suit.Button("Click?", suit.layout:row(200,30))

    -- if the button was entered, play a sound
    if state.entered then love.audio.play(snd) end

    -- if the button was pressed, take damage
    if state.hit then print("Ouch!") end
```

(continues on next page)

(continued from previous page)

```
    -- put an input box below the button
    -- the cell of the input box has the same size as the cell above
    -- if the input cell is submitted, print the text
    if suit.Input(input, suit.layout:row()).submitted then
        print(input.text)
    end

    -- put a button below the input box
    -- the width of the cell will be the same as above, the height will be 40 px
    if suit.Button("Hover?", suit.layout:row(nil,40)).hovered then
        -- if the button is hovered, show two other buttons
        -- this will shift all other ui elements down

        -- put a button below the previous button
        -- the cell height will be 30 px
        -- the label of the button will be aligned top left
        suit.Button("You can see", {align='left', valign='top'}, suit.layout:row(nil,
↪30))

        -- put a button below the previous button
        -- the cell size will be the same as the one above
        -- the label will be aligned bottom right
        suit.Button("...but you can't touch!", {align='right', valign='bottom'},
                                               suit.layout:row())
    end

    -- put a checkbox below the button
    -- the size will be the same as above
    -- (NOTE: height depends on whether "Hover?" is hovered)
    -- the label "Check?" will be aligned right
    suit.Checkbox(chk, {align='right'}, suit.layout:row())

    -- put a nested layout
    -- the size of the cell will be as big as the cell above or as big as the
    -- nested content, whichever is bigger
    suit.layout:push(suit.layout:row())

        -- change cell padding to 3 pixels in either direction
        suit.layout:padding(3)

        -- put a slider in the cell
        -- the inner cell will be 160 px wide and 20 px high
        suit.Slider(slider, suit.layout:col(160, 20))

        -- put a label that shows the slider value to the right of the slider
        -- the width of the label will be 40 px
        suit.Label(("%.02f"):format(slider.value), suit.layout:col(40))

    -- close the nested layout
    suit.layout:pop()

    -- put an image button below the nested cell
    -- the size of the cell will be 200 by 100 px,
    --      but the image may be bigger or smaller
    -- the button shows the image `normal' when the button is inactive
    -- the button shows the image `hovered` if the mouse is over an opaque pixel
    --      of the ImageData `mask`
```

(continues on next page)

(continued from previous page)

```
    -- the button shows the image `active` if the mouse is above an opaque pixel
    --      of the ImageData `mask` and the mouse button is pressed
    -- if `mask` is omitted, the alpha-test will be swapped for a test whether
    --      the mouse is in the area occupied by the widget
    suit.ImageButton(normal, {mask = mask, hovered = hovered, active = active}, suit.
↪layout:row(200,50))

    -- if the checkbox is checked, display a precomputed layout
    if chk.checked then
        -- the precomputed layout will be 3 rows below each other
        -- the origin of the layout will be at (400,100)
        -- the minimal height of the layout will be 300 px
        rows = suit.layout:rows{pos = {400,100}, min_height = 300,
            {200, 30},    -- the first cell will measure 200 by 30 px
            {30, 'fill'}, -- the second cell will be 30 px wide and fill the
                          -- remaining vertical space between the other cells
            {200, 30},    -- the third cell will be 200 by 30 px
        }

        -- the first cell will contain a witty label
        -- the label will be aligned left
        -- the font of the label will be smaller than the usual font
        suit.Label("You uncovered the secret!", {align="left", font = smallerFont},
                                                rows.cell(1))

        -- the third cell will contain a label that shows the value of the slider
        suit.Label(slider.value, {align='left'}, rows.cell(3))

        -- the second cell will show a slider
        -- the slider will operate on the same data as the first slider
        -- the slider will be vertical instead of horizontal
        -- the id of the slider will be 'slider two'. this is necessary, because
        --     the two sliders should not both react to UI events
        suit.Slider(slider, {vertical = true, id = 'slider two'}, rows.cell(2))
    end
end

function love.draw()
    -- draw the gui
    suit.draw()
end

function love.textinput(t)
    -- forward text input to SUIT
    suit.textinput(t)
end

function love.keypressed(key)
    -- forward keypressed to SUIT
    suit.keypressed(key)
end

-- generate assets (see love.load)
function generateClickySound()
    local snd = love.sound.newSoundData(512, 44100, 16, 1)
    for i = 0,snd:getSampleCount()-1 do
        local t = i / 44100
```

(continues on next page)

(continued from previous page)

```
        local s = i / snd:getSampleCount()
        snd:setSample(i, (.7*(2*love.math.random()-1) + .3*math.sin(t*9000*math.pi))␣
↪* (1-s)^1.2 * .3)
    end
    return love.audio.newSource(snd)
end

function generateImageButton()
    local metaballs = function(t, r,g,b)
        return function(x,y)
            local px, py = 2*(x/200-.5), 2*(y/100-.5)
            local d1 = math.exp(-((px-.6)^2 + (py-.1)^2))
            local d2 = math.exp(-((px+.7)^2 + (py+.1)^2) * 2)
            local d = (d1 + d2)/2
            if d > t then
                return r,g,b, ((d-t) / (1-t))^.2
            end
            return 0,0,0,0
        end
    end

    local normal, hovered, active = love.image.newImageData(200,100), love.image.
↪newImageData(200,100), love.image.newImageData(200,100)
    normal:mapPixel(metaballs(.48, .74,.74,.74))
    hovered:mapPixel(metaballs(.46, .2,.6,.6))
    active:mapPixel(metaballs(.43, 1,.6,0))
    return love.graphics.newImage(normal), love.graphics.newImage(hovered), love.
↪graphics.newImage(active), normal
end
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

W