Scrupp 0.4 Manual

Table of Contents

1 Getting Started **1**
1.1 Architecture 1
1.2 Operating systems 1
1.2.1 Windows 1
1.2.2 Linux 1
1.2.3 Mac OS X 1
1.3 Using archives 1
1.4 Using the Command Line 2
1.5 Script arguments 2

2 File Access **4**
2.1 Overview 4
2.2 Startup 4
2.3 Functions 5

3 Callbacks **7**
3.1 Overview 7
3.2 Callbacks 7
3.3 Example 7

4 General Functions **9**
4.1 Version 9
4.2 Platform 9
4.3 General functions 9
4.4 Modifications of the render state 9

5 Drawing **11**
5.1 Drawing points, lines and polygons 11
5.2 Example 11

6 Image handling **13**
6.1 Loading 13
6.2 Methods 13
6.3 Example 15

7 Cairo support **16**
7.1 Additional or Modified Cairo Functions 16
7.2 Additional Scrupp Functions 16
7.3 Example 16

8 Font handling **18**
8.1 Note 18
8.2 UTF-8 18
8.3 Loading 18
8.4 Methods 18
8.5 Example 18

9 Sound handling **20**
9.1 Loading 20
9.2 Methods 20
9.3 Example 20

Table of Contents

10 Music handling.....21
10.1 Loading.....21
10.2 Methods.....21
10.3 Functions.....21
10.4 Example.....22

11 Movie handling.....23
11.1 How to create MPEG-1 movies.....23
11.2 Loading.....23
11.3 Methods.....23
11.4 Example.....25

12 Mouse handling.....26
12.1 Functions.....26
12.2 Callbacks.....26
12.3 Example.....26

13 Keyboard handling.....27
13.1 Functions.....27
13.2 Callbacks.....27
13.3 Key strings.....27
13.4 Example.....30

14 Network support.....32
14.1 Example.....32

15 Simple class implementation.....33
15.1 Functions.....33
15.2 Constructor.....33
15.3 Creation of instances.....33
15.4 Methods.....33
15.5 Example.....33

16 Animation class.....35
16.1 Requirements.....35
16.2 Creation.....35
16.3 Methods.....35
16.4 Example.....35

17 Colors.....37
17.1 Provided colors.....37
17.2 Usage.....37
17.3 Example.....37

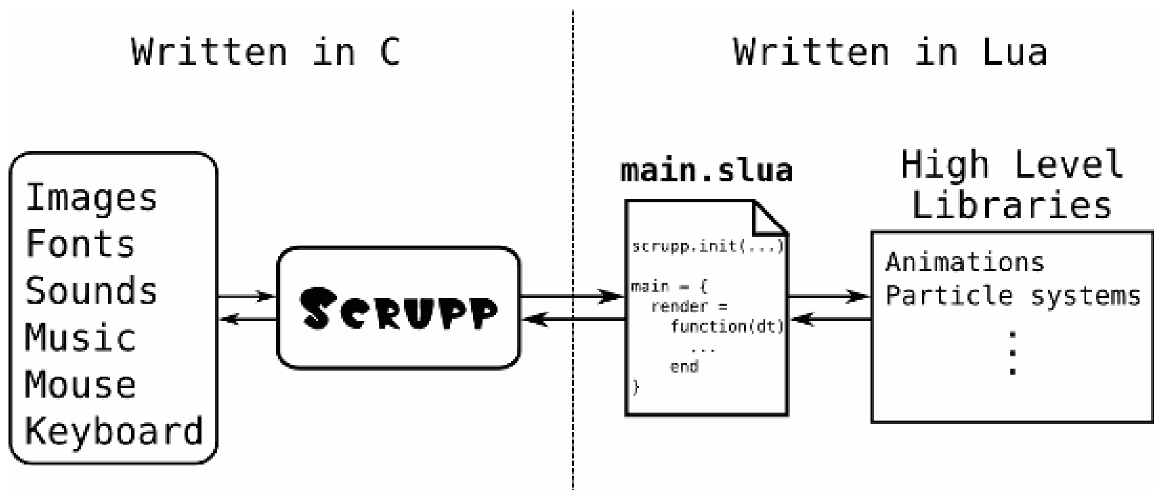
18 Font class.....39
18.1 UTF-8.....39
18.2 Kerning.....39
18.3 Requirements.....39
18.4 Creation.....39
18.5 Methods.....39
18.6 Example.....40

Table of Contents

19 Timer class....................41
19.1 Requirements....................41
19.2 Creation....................41
19.3 Methods....................41
19.4 Example....................41

1 Getting Started

1.1 Architecture



This figure shows the architecture of an application created with Scrupp. The whole development happens on the right side using Lua.

1.2 Operating systems

1.2.1 Windows

On Windows the easiest way to use Scrupp is to let the installer associate script files (*.slua) and archives (*.sar) with Scrupp. That way, a double-click on one of these filetypes automatically runs the file with Scrupp.

Another possibility is to drop either a script file, an archive, or a directory on the executable scrupp.exe. The dropped file or directory becomes the first command line argument. See Using the Command Line for details.

1.2.2 Linux

On Linux you can either configure your file manager to open the script files (*.slua) and archives (*.sar) with Scrupp, or use the command line.

Drag and drop is supported as well (see Windows).

1.2.3 Mac OS X

On Mac OS X the nescessary filetypes should be automatically associated with Scrupp. Thus, a double click on a script file (*.slua) or an archive (*.sar) should invoke Scrupp.

Drag and drop is supported as well (see Windows).

1.3 Using archives

Scrupp supports zip archives. You can compress all the files belonging to your application to a single zip file. The extension of the file is not important, but the default for a Scrupp archive is *.sar. Scrupp can load the archive and executes the main script file called main.slua. It is important that this file is located in the root of the zip file.

wrong:

```
archive.sar
  \-- myApp (folder)
        \-- main.slua
```

right:

```
archive.sar
  \-- main.slua
```

1.4 Using the Command Line

The following list shows the possibilitis for running Scrupp from the command line.

1. No command line arguments

   ```
   $ scrupp [-- arg1 arg2 ... argN]
   ```

   Scrupp tries to open the file `main.slua`. The working directory of the script will be the one containing the executable. If this file is not found, Scrupp will try to open the zip archive `main.sar` which has to contain the file `main.slua`. If both files are not found, an error will be thrown. Extra arguments to the script have to be separated from the executable's name by `--`.
2. A Lua file as argument

   ```
   $ scrupp <Lua file> [arg1 arg2 ... argN]
   ```

   Scrupp tries to execute the Lua file. The working directory will be the one containing the Lua file. It is possible to give extra arguments to the Lua file.
3. A zip archive as argument

   ```
   $ scrupp <zip archive> [arg1 arg2 ... argN]
   ```

   The archive is prepended to the search path. The directory that contains the archive becomes the working directory. After this Scrupp tries to execute `main.slua` which should be located inside this archive. It is possible to give extra arguments to the Lua file.
4. A directory as argument

   ```
   $ scrupp <directory> [arg1 arg2 ... argN]
   ```

   The directory is prepended to the search path and set as the working directory. After this Scrupp tries to execute `main.slua` which should be located inside this directory. It is possible to give extra arguments to the Lua file.

1.5 Script arguments

Before running a Lua file, Scrupp collects all command line arguments in the global table `arg`. The name of the chosen Lua file, directory or archive becomes the element at index 0 in that table. Additional arguments are stored in index 1, 2 and so on. The name of the executable goes to index -1. The script can access these arguments by using `arg[1], arg[2],` etc. or the vararg expression '...'.

```
-- print the name of the script
print("name of the script:", arg[0])

-- print the number of arguments to the script
print("number of arguments:", select('#', ...) )
```

```
-- print all arguments
print("arguments:", ...)

-- this prints the first argument
print("Argument 1: ", arg[1] or "none")

-- exiting Scrupp
scrupp.exit()
```

Just save this example code in a file called `argtest.lua` and run

```
$ scrupp argtest.lua arg1 arg2
```

This can be done with every example from the manual. To get started have a look at the General Functions, save the example in a file and run this file with Scrupp.

2 File Access

2.1 Overview

Scrupp uses a so called virtual filesystem (VFS) which is implemented using the PhysicsFS library. The VFS consists of a list of paths to directories and archives - the search path.

Everytime a file is accessed using one of the Scrupp functions (e.g. scrupp.addImage()) every entry of the search path is checked for the existance of the specified file. If the file exists in multiple paths, the first one is chosen.

The standard io facilities of Lua (all functions in the global io table) do **not** support the virtual filesystem. These functions modify files in the working directory.

2.2 Startup

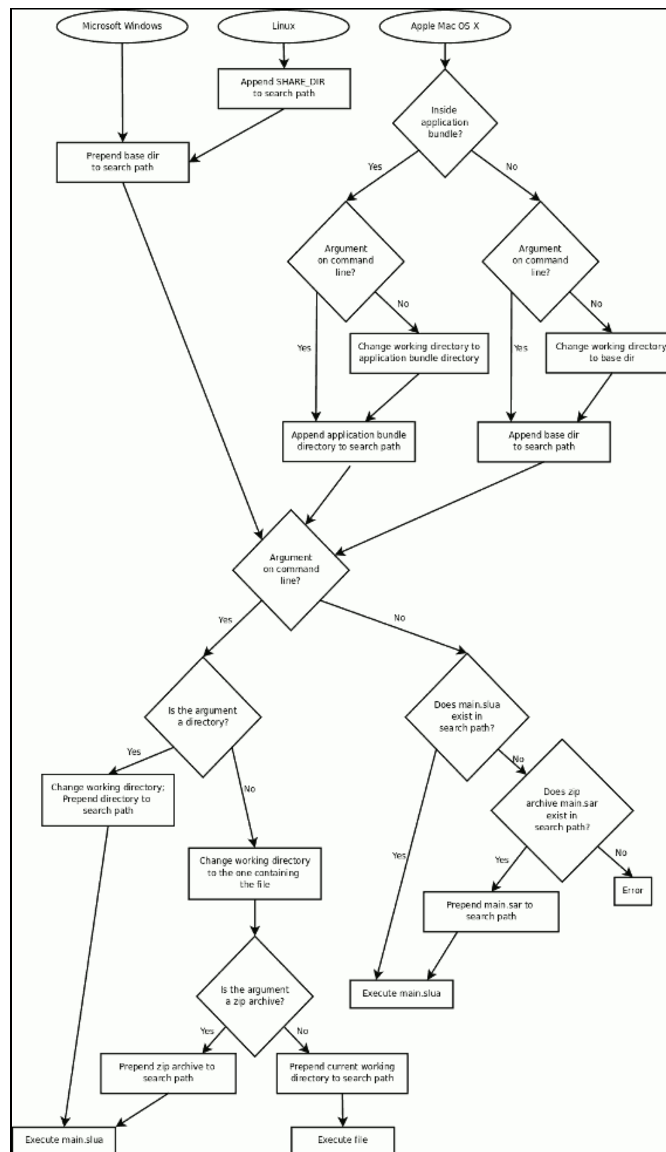
On startup, Scrupp constructs the search path and sets the working directory. In order to understand the way how that's done the following definitions are needed:

base directory
: The directory that contains the executable.

working directory
: This directory is important if you want to modify files using the input and output facilities of plain Lua.

`SHARE_DIR`
: On Linux, the SHARE_DIR defaults to `${prefix}/share/scrupp`. The prefix is defined by the configure script (default: /usr/local). During the installation, all support files ("main.slua" and the directories "examples", "fonts" and "scripts") are copied to this directory.

The following figure shows Scrupp's behaviour on startup, i. e. how it sets the search path and working directory. Click it for a bigger version.



2.3 Functions

require(modname)
: Loads the specified module. The first thing is to look for the module in the virtual filesystem using the new *path.scrupppath*. The default list of paths is "?.lua;scripts/?.lua;scripts/?/init.lua;scripts/socket/?.lua". That means the question mark in the semicolon separated list of paths is replaced by the *modname*. If the virtual filesystem contains a file with the resulting name, it is loaded. Otherwise the next path of the list is processed. The default list of paths can be overridden by the environment variable SCRUPP_PATH. If the module is not found in the virtual filesystem, require falls back to its default behaviour.

dofile(filename)
: Opens the named Lua file and executes its content. It looks for the file in the virtual filesystem. If the file is not found, it falls back to the default behaviour.

scrupp.getSearchPath()
: Returns an array with the paths of the current search path.

scrupp.setSearchPath(array)
: Sets the search path to the paths in the *array*.

scrupp.fileExists(filename)
: Returns true if the file exists in the virtual filesystem, false otherwise.

scrupp.isDirectory(path)

Returns true if the *path* points to a directory in the virtual filesystem, false otherwise.

scrupp.isSymbolicLink(filename)

Returns true if the file is a symbolic link, false otherwise.

3 Callbacks

3.1 Overview

At the beginning, Scrupp runs a Lua file which is chosen depending on the command line arguments. Read "Getting Started" for details. This file can run other Lua files, load any media type, define functions and so on. It has to define a table with the name 'main' which contains the callbacks.

3.2 Callbacks

render(delta)
: This callback function is mandatory. It is executed in every frame. Scrupp calls it with *delta* as the only argument. This parameter holds the time in milliseconds which has passed since the last call to this function. This time should be about 10 ms long because Scrupp tries to make constant 100 frames per second. This information can be used to time animations. The rendering of images should happen in this function.

resized(width, height)
: This callback function is optional. If a resizable window is created and it is resized, this function will be executed. It gets the new window width and height as arguments. This function should call *scrupp.init()* with these new values as arguments. Otherwise, it's not possible to render to new areas of the window.

keypressed(key)
: This callback function is optional. If a key is pressed, this function will be executed. Its only argument is the key constant. Read the section about Keyboard usage for details and an example.

keyreleased(key)
: This callback function is optional. If a key is released, this function will be executed. Its only argument is the key constant. Read the section about Keyboard usage for details and an example.

mousepressed(x, y, button)
: This callback function is optional. If a mouse button (including the mouse wheel) is pressed, this function will be executed. Read the section about Mouse usage for details and an example.

mousereleased(x, y, button)
: This callback function is optional. If a mouse button (including the mouse wheel) is released, this function will be executed. Read the section about Mouse usage for details and an example.

3.3 Example

```
-- create a resizable window
scrupp.init("Callback Test", 600, 400, 32, false, true)

local image = scrupp.addImage("path_to_image_file")

main = {
	render = function(dt)
		image:render(mouse.getX(), mouse.getY())
	end,

	resized = function(width, height)
		scrupp.init("Resized!", width, height, 32, false, true)
	end,

	mousepressed = function(x, y, button)
		print(button .." button pressed at "..x..", "..y)
	end,

	mousereleased = function(x, y, button)
		print(button .." button released at "..x..", "..y)
```

```
        end,

        keypressed = function(k)
                if k == "ESCAPE" then
                        scrupp.exit()
                end
                print("key '" .. k .. "' pressed")
        end,

        keyreleased = function(k)
                print("key '" .. k .. "' released")
        end
}
```

4 General Functions

4.1 Version

scrupp.VERSION
This variable contains a string with the version of Scrupp. The format is major.minor.

4.2 Platform

scrupp.PLATFORM
Contains a string describing the platform Scrupp was compiled on. Can be one of "Windows", "Mac OS X", "Linux" and "Unknown".

4.3 General functions

scrupp.init(title, width, height, bit_depth, fullscreen, [resizable])
Creates a window with the chosen *title*, a size in pixel chosen by *width* and *height* and the chosen *bit_depth*. The *fullscreen* parameter toggles between fullscreen (*true*) or window mode (*false*). If window mode is selected, the optional boolean parameter *resizable* enables or disables the resizing of the window.

scrupp.getWindowWidth()
Has to be called after *scrupp.init()*. Returns the window width in pixels.

scrupp.getWindowHeight()
Has to be called after *scrupp.init()*. Returns the window height in pixels.

scrupp.getWindowSize()
Has to be called after *scrupp.init()*. Returns the window width and height in pixels.

scrupp.getTicks()
Has to be called after *scrupp.init()*. Returns the number of milliseconds since the start of the application.

scrupp.setDelta(delta)
Sets the minimum delta in milliseconds between two frames. If *delta* equals 0, Scrupp runs as fast as possible.

scrupp.showCursor([show])
If called without argument, returns the current state of the cursor as boolean. Otherwise the state of the cursor is changed depending on the boolean argument *show*.

scrupp.exit()
A call to this function immediately exits the application.

4.4 Modifications of the render state

The following set of functions modify the render state of Scrupp. That means, they change the way everything is rendered afterwards. This applies to the rendering of images, movies and drawings.

It's important to know, that the three functions to translate, scale and rotate the view influence each other depending on the order in which they are executed. The recommended order that usually does what you want is:

1. scrupp.translateView()
2. scrupp.rotateView()
3. scrupp.scaleView()

scrupp.translateView(x, y)
: Moves the view by the specified number of pixels in *x* and *y* direction.
This function wraps the OpenGL function *glTranslate*.

scrupp.scaleView(sx, sy)
: Scales the view by the specified factor *sx* in x and *sy* in y direction.
This function wraps the OpenGL function *glScale*.

scrupp.rotateView(angle)
: Rotates the view by the specified *angle* in degrees. Positive values rotate clockwise, negative ones counter-clockwise.
This function wraps the OpenGL function *glRotate*.

scrupp.saveView()
: Saves the current view. Can be called multiple times. Every time the view is saved on a stack. The view on the top of the stack can be restored using *scrupp.restoreView()*.
This function wraps the OpenGL function *glPushMatrix*.

scrupp.restoreView()
: Restores the view that was saved most recently with *scrupp.saveView()*. Then this view is removed from the stack of saved views.
This function wraps the OpenGL function *glPopMatrix*.

scrupp.resetView()
: Replaces the current view with the default one.
This function wraps the OpenGL function *glLoadIdentity*.

5 Drawing

5.1 Drawing points, lines and polygons

scrupp.draw(table)

This function can draw points, lines and polygons. The array part of the *table* contains the list of x,y coordinate pairs of the points. The interpretation of those points depends on several settings made by the following key-value pairs. All of them are optional.

color [= {255, 255, 255, 255}]
: This table sets the color of the point(s), line(s) or polygon(s) to the given value. The first three elements are the red, green and blue components of the color. The optional fourth value is the alpha component. All four elements have to be in the range from 0 to 255. The default color is opaque white.

size [= 1]
: Sets the pixel size or line width to the chosen number. The default value is 1.

centerX [= 0] and centerY [= 0]
: These are the coordinates of the center of scaling and rotation.

scaleX [= 1] and scaleY [= 0]
: Define the scale factor of the x- and y-axis.

rotate [= 0]
: The angle in degrees to rotate the graphic.

relative [= false]
: If this boolean value is true, the first point in the array defines where the graphic should be moved to. All other points in the array are relative to the first one.

connect [= false]
: If this boolean value is true and if there are more than two points a line will be drawn between the first and the second, the second and the third and so on. The default value is false.

fill [= false]
: If this boolean value is true the created shape will be filled with the chosen *color*. The default value is false.

antialiasing [= false]
: This boolean value activates or deactivates the anti-aliasing. If it is true, lines will blend nicely with the background. The default value is false.

pixellist [= false]
: If the boolean value *pixellist* is true, a point will be drawn at each location given by the coordinate pairs. If it is false and there is more than one point, the interpretation of the coordinate pairs depends on the other options:

- *fill* is true: polygon is drawn
- *fill* is false and *connect* is true: connected lines are drawn
- *fill* is false and *connect* is false: unconnected lines are drawn; one between the first and the second point, one between the third and the fourth one and so on

Note:

- Pixel with a size greater than one cannot be rotated; use a filled rectangle instead
- Pixel disappear when the center leaves the viewable area
- The antialiasing of pixel with a size greater than one depends on the graphics card driver and the OpenGL implementation. Sometimes, they are rendered as circles and sometimes not.

5.2 Example

```
scrupp.init("Draw Test - Click to test!", 600, 400, 32, false)

-- prepare one table for big white pixels
```

```
local pixels = { antialiasing = false, size = 20, pixellist = true }
-- prepare one table for red connected lines
local lines = { color = {255,0,0}, antialiasing = true, connect = true }

main = {
	render = function(dt)
		scrupp.draw(lines)
		scrupp.draw(pixels)
	end,

	mousepressed = function(x, y, button)
		-- add the point where the mouse was pressed to the list of pixels
		pixels[#pixels+1] = x
		pixels[#pixels+1] = y

		-- add the point to the list of lines, too
		lines[#lines+1] = x
		lines[#lines+1] = y
	end
}
```

6 Image handling

Image support is implemented using SDL_image.

JPEG support requires the JPEG library:
    IJG Homepage
PNG support requires the PNG library:
    PNG Homepage
and the Zlib library:
    Zlib Homepage
TIFF support requires the TIFF library:
    SGI TIFF FTP Site

6.1 Loading

scrupp.addImage(filename)
    Loads the image file with the given *filename*. Supported formats: BMP, PNM (PPM/PGM/PBM), XPM, LBM, PCX, GIF, JPEG, PNG and TIFF. Returns an image object.
scrupp.addImageFromString(str)
    Loads the image from the string *str*. Supports the same formats as *scrupp.addImage()*. Returns an image object.

6.2 Methods

image:getWidth()
    Returns the image width.
image:getHeight()
    Returns the image height.
image:getSize()
    Returns the width and the height of the image.

image:isTransparent(x, y)
    Returns true if the pixel of the *image* with the coordinates *x* and *y* is not opaque, i.e. has an alpha value lower than 255. Returns false otherwise. This is not influenced by the alpha value of the whole image changed using *image:setAlpha()*.

image:setAlpha(alpha)
    Changes the alpha value of the image. *alpha* has to be between 0 (transparent) and 255 (opaque).
image:getAlpha()
    Returns the current alpha value of the image. This value is between 0 (transparent) and 255 (opaque).

image:setColor(r, g, b)
    Changes the color of the image. Think of this as a color filter so e.g. if you set it to red, only the red part of every pixel is rendered. *r, g, b* have to be numbers in the range from 0 to 255.
image:getColor()
    Returns three numbers in the range from 0 to 255, one for each color component (r, g, b). If no color was set using *image:setColor(r, g, b)* white (255, 255, 255) is returned.
image:clearColor()
    Removes any custom color from the image. After that it's like the color was white (255, 255, 255) so everything is rendered normally.

image:setCenterX(cx)
image:setCenterY(cy)

image:setCenter(cx, cy)
: These functions change the center of the image. The image is rendered with these coordinates at the position given to *image:render(x, y)*. Furthermore, the image is rotated around this point. The default center of an image is (0, 0), the top left corner.

image:getCenterX()
image:getCenterY()
image:getCenter()
: Returns the x-, the y- or both coordinates of the image's center.

image:setScaleX(sx)
image:setScaleY(sy)
image:setScale(sx, sy)
: These functions change the scaling of the image. The default scaling of an image is (1, 1), it's original size.

image:getScaleX()
image:getScaleY()
image:getScale()
: Returns the x-, the y- or both scaling factors of the image.

image:setRotation(angle)
: Changes the rotation of the image. The *angle* in degrees defaults to 0. Positive angles rotate clockwise.

image:getRotation()
: Returns the current rotation's angle of the image in degrees.

image:setRect(x, y, w, h)
: Changes the part of the image which should be rendered. You can define the top left corner of the rectangle (*x, y*) and the width and height (*w, h*).

image:getRect()
: Returns four numbers (*x, y, w, h*) describing the part of the image which should be rendered. If no rectangle was set using *image:setRect(x, y, w, h)* (0, 0, width of image, height of image) is returned.

image:clearRect()
: Removes any custom rectangle from the image. After that it's like the rectangle is (0, 0, width of image, height of image) so the whole image is rendered.

image:render(x, y)
: Renders the *image* at the window coordinates *x* and *y*.

image:render(table)
: This is nearly the same as the last one. This time a *table* contains the arguments. It's complete structure follows.

```
table = {
        x, -- x-coordinate of the image position -- mandatory
        y, -- y-coordinate of the image position -- mandatory
        centerX = [x-coordinate of the center for positioning, scaling and rotation], -- op
        centerY = [y-coordinate of the center for positioning, scaling and rotation], -- op
        scaleX = [scale factor for the x-axis], -- optional
        scaleY = [scale factor for the y-axis], -- optional
        rotate = [angle in degrees to rotate the image], -- optional
        rect = { x, y, width, height }, -- optional
        color = { red, green, blue, alpha } -- optional
}
```

The first two elements of the array part of the *table* have to be the x- and y-coordinate of the point that the image should be rendered at. The *table* may have an optional *rect* entry. This has to be a table describing the rectangle the image should be clipped to. It contains the x- and y-coordinate of the top

left corner and the width and height of the rectangle inside the image. The entries *centerX* and *centerY* are optional and default to zero, the top left corner of the image. *scaleX* and *scaleY* are optional as well and default to one. *rotate* has a default value of zero dregrees and is optional, too. The optional *color* entry is a table defining some kind of color filter. If you have a grayscale image, it will be colorized with this color. As always the table looks like this: {red, green, blue, [alpha]} with each value between 0 and 255. The default is opaque white, so the color of the image is not changed.

6.3 Example

```
scrupp.init("Image Test", 600, 400, 32, false)

-- load an image file
local image = scrupp.addImage("path_to_image_file")

-- get the dimension of the image
local width = image:getWidth()
local height = image:getHeight()

-- this has the same result:
local width, height = image:getSize()

if width > scrupp.getWindowWidth() or height > scrupp.getWindowHeight() then
        print("Please choose an image with smaller dimensions.")
        scrupp.exit()
end

-- calculate the x- and y-coordinate of the image
local x = (scrupp.getWindowWidth()-width)/2
local y = (scrupp.getWindowHeight()-height)/2

main = {
        render = function(dt)
                -- get the position of the mouse pointer
                mx, my = mouse:getPos()
                -- set the image alpha depending on the x-coordinate of the
                -- mouse pointer
                image:setAlpha(mx/scrupp.getWindowWidth()*200+50)
                -- if you click with the left mouse button,
                -- only a part of the image is rendered
                if mouse.isDown("left") then
                        image:render{
                                mx, my,
                                rect = { mx-x, my-y, width/2, height/2 }
                        }
                -- otherwise the whole image is rendered
                else
                        image:render(x,y)
                end
        end
}
```

7 Cairo support

Scrupp 0.4 contains version 1.2 of the Cairo binding lua-oocairo written by Geoff Richards.

By using this Cairo binding it is possible to *generate* images. Due to the runtime generation these images can be completely resolution independant.

Another possibility is the creation of PNG, PDF, PS and SVG files.

The original website of lua-oocairo (http://www.daizucms.org/lua/library/oocairo/) is no longer available on the web. That's why I have uploaded a copy on this website. It contains the documentation of lua-oocairo.

The doc directory of the Scrupp distribution contains the lua-oocairo documentation as well.

In order to ease the usage of Cairo from Scrupp, the binding was modified and new functions were added to Scrupp.

7.1 Additional or Modified Cairo Functions

Cairo.image_surface_create_from_png(file/filename)
: This function was modified to support the virtual filesystem provided by PhysFS. That means it can load the png from a zip archive. Apart from that, it behaves exactly the way described in the lua-oocairo documentation.

Cairo.image_surface_create_from_file(filename)
: Loads the image file with the specified *filename* and returns an image surface which can be used with Cairo. This function supports the same image formats as *scrupp.addImage()* and the virtual filesystem provided by PhysFS.
Note: It does not support images with an alpha channel! In this case, save the image as png and use *Cairo.image_surface_create_from_png()*.
The Cairo example *masking.slua* from the Scrupp distribution shows how to use this function.

7.2 Additional Scrupp Functions

scrupp.addImageFromCairo(surface)
: Returns a Scrupp image generated from the specified Cairo *surface*.

7.3 Example

```
-- load the Cairo library
local Cairo = require "oocairo"

-- create an image surface with a size of 200x200 pixel
local surface = Cairo.image_surface_create("rgb24", 200, 200)

-- create a Cairo context used for drawing
local cr = Cairo.context_create(surface)

-- create some arbitrary shapes
cr:set_source_rgb(1, 1, 1)
cr:paint()

cr:move_to(0, 0)
cr:line_to(190, 100)
cr:line_to(100, 185)
cr:line_to(200, 200)
```

```
cr:line_to(30, 130)
cr:close_path()
cr:set_source_rgb(0.8, 0.4, 1)
cr:fill()

cr:move_to(180, 30)
cr:line_to(100, 20)
cr:line_to(80, 120)
cr:set_source_rgb(0.5, 0.7, 0.3)
cr:fill_preserve()
cr:set_source_rgb(0, 0, 0)
cr:set_line_width(3)
cr:stroke()

-- use scrupp to display the surface
scrupp.init("Cairo: simple-example", 200, 200, 32, false)

-- convert the Cairo surface to a Scrupp image
local image = scrupp.addImageFromCairo(surface)

main = {
	render = function(dt)
		image:render(0,0)
	end,

	keypressed = function(key)
		if key == "ESCAPE" then
			scrupp.exit()
		end
	end
}
```

8 Font handling

8.1 Note

Have a look at the font plugin. This makes the usage of truetype fonts easier.

8.2 UTF-8

Scrupp supports UTF-8. If you want to use this feature, remember to edit your scripts using the UTF-8 mode of your editor. Save them with UTF-8 encoding (without BOM).

8.3 Loading

scrupp.addFont(filename, size)
: Loads a font file with the given *filename* and the given *size* in pixels. This can load TTF and FON formats. Returns a font object.

8.4 Methods

font:getTextSize(text)
: Returns the width and height of the resulting surface of the UTF-8 encoded *text* rendered using *font*. No actual rendering is done. The returned height is the same as returned by *font:getHeight()*.

font:getHeight()
: Returns the maximum pixel height of all glyphs of the given *font*.

font:getLineSkip()
: Returns the recommended pixel height of a rendered line of text of the loaded *font*. This usually is larger than the result of *font:getHeight()*.

font:generateImage(text)
: Returns an image containing the *text* rendered with the given *font*. The text color is white and the background is transparent. To use another color call *font:generateImage()* with a table as sole argument. The color of the created image can be changed by setting the appropriate option during rendering (see *image:render()*).

font:generateImage(table)
: This is nearly the same as the last one. This time a *table* contains the arguments.

```
table = {
        text,
        color = {red, green, blue, [alpha]} -- optional
}
```

The first element of the array part of the *table* has to be the text. The *table* may have an optional *color* entry. This has to be a table containing three numbers between 0 and 255, one for each color component (red, green, blue). An optional fourth entry is the transparency (0-255).
Remember: The color of an image can be changed during rendering.

8.5 Example

```
scrupp.init("Font Test", 600, 400, 32, false)

-- load a font
local font = scrupp.addFont("fonts/Vera.ttf", 20)
```

```
-- get the recommended line height in pixel
local lineSkip = font:getLineSkip()
-- get the text size of a sample text
local w, h = font:getTextSize("Hello World")

-- define a color (opaque blue)
local cBlue = {0, 0, 255, 255}
-- the transparency defaults to 255 (opaque), so this is the same:
local cBlue = {0, 0, 255}

-- generate an image containing some text using the default color
local image_1 = font:generateImage("Hello World")

-- generate an image using the defined color cBlue
-- -- because font:generateImage() gets only one parameter and this is a table
-- -- we can omit the parenthesises and just use the curly brackets of the table
-- -- constructor
local image_2 = font:generateImage{"Hello World", color = cBlue}
-- this has the same result:
local image_2 = font:generateImage{"Hello World", color = {0, 0, 255}}

main = {
	render = function(dt)
		-- render the already loaded images:
		image_1:render(100,50)
		image_2:render(100 + w, 50 + lineSkip)
	end
}
```

9 Sound handling

The sound support is implemented using SDL_mixer.

Note: If you load a sound and play it multiple times in parallel all sound manipulating methods will affect every instance of this sound!

9.1 Loading

scrupp.addSound(filename)
: Loads a sound file with the given *filename*. This can load WAV, AIFF, RIFF, OGG and VOC formats. Returns a sound object.

9.2 Methods

sound:setVolume(volume)
: Sets the volume of the sound to the specified value between 0 (mute) and 128 (maximum volume).

sound:getVolume()
: Returns the current volume of the sound. This value is between 0 (mute) and 128 (maximum volume).

sound:play([loops=1])
: Plays a sound file loaded with *scrupp.addSound()*. The optional parameter *loops* defines the number of times the sound will be played. 0 means infinite looping. The default is to play the sound once.

sound:pause()
: Pauses the playback of the sound. A paused sound may be stopped with *sound:stop()*. If the sound is not playing, this method will do nothing.

sound:resume()
: Unpauses the sound. This is safe to use on stopped, paused and playing sounds.

sound:stop()
: Stops the playback of the sound.

sound:isPlaying()
: Tells you if the sound is actively playing, or not.
: **Note:** Does not check if the sound has been paused, i.e. paused sounds return *true*. Only stopped sounds return *false*.

sound:isPaused()
: Tells you if the sound is paused, or not.
: **Note:** This state is only influenced by calls to *sound:pause()* and *sound:resume()*.

9.3 Example

```
scrupp.init("Sound Test", 600, 400, 32, false)

-- load a sound file
local sound = scrupp.addSound("path_to_sound_file")

main = {
	-- empty render function
	render = function(dt)
	end,

	-- play the sound every time a mouse button is pressed
	mousepressed = function(x, y, button)
		sound:play()
	end
}
```

10 Music handling

The music support is implemented using SDL_mixer.

Scrupp supports only one music file playing at a time. Because of that there is only one method for a music object: *play*. All the other music manipulating functions are independant from a special music object. They change whatever music file is playing at the moment.

10.1 Loading

scrupp.addMusic(filename)
: Loads a music file with the given *filename*. This can load WAV, MOD, XM, OGG and MP3. Returns a music object.
Note: Only MP3 files with id3v1 tag are supported. If you try to load a file with id3v2 tag, the error message will say: *Module format not recognized*. To strip a possible id3v2 tag you can use the command *id3convert* from the *id3lib*:

```
$ id3convert -s -2 <mp3 file>
```

10.2 Methods

music:play([loops=0], [fade_in_time=0])
: Plays a music file loaded with *scrupp.addMusic()*. The optional parameter *loops* defines the number of times the music will be played. 0 means inifite looping and is the default. The optional parameter *fade_in_time* defines the number of milliseconds the music will fade in, the default is 0 ms.

10.3 Functions

scrupp.setMusicVolume(volume)
: Sets the music volume to the specified value between 0 (mute) and 128 (maximum volume).

scrupp.getMusicVolume()
: Returns the current music volume. This value is between 0 (mute) and 128 (maximum volume).

scrupp.pauseMusic()
: Pauses the music playback. Paused music may be stopped with *scrupp.stopMusic()*.

scrupp.resumeMusic()
: Unpauses the music. This is safe to use on stopped, paused and playing music.

scrupp.stopMusic([fade_out_time=0])
: Stop the playback of music. The optional paramter *fade_out_time* gives the number of milliseconds the music will fade out. It defaults to 0 ms.

scrupp.rewindMusic()
: Rewinds the music to the start. This is safe to use on stopped, paused and playing music. This function only works for these streams: MOD, XM, OGG, MP3.

scrupp.musicIsPlaying()
: Tells you if music is actively playing, or not.
Note: Does not check if the music has been paused, i.e. paused music returns *true*. Only stopped music returns *false*.

scrupp.musicIsPaused()
: Tells you if music is paused, or not.
Note: This state is only influenced by calls to *scrupp.pauseMusic()* and *scrupp.resumeMusic()*.

10.4 Example

```
scrupp.init("Music Test", 600, 400, 32, false)

-- load some music
local music = scrupp.addMusic("path_to_music_file")
-- start music
music:play()

main = {
	-- empty render function
	render = function(dt)
	end
}
```

11 Movie handling

Movie support is implemented using SMPEG.

Scrupp **only** supports MPEG-1 videos! Furthermore automatic aspect ratio correction is not implemented. For example, a video file has a resolution of 352x240 (= 1.47:1) but the aspect ratio is defined by the header as 1.34:1. Therefore the video has to be resized in order to correct the aspect ratio. A resolution of 352x262 satisfies the ratio. The scale factor in vertical direction calculates to 262/240 = 1.09.

11.1 How to create MPEG-1 movies

The recommended way to convert video files to the MPEG-1 format is using MEncoder, a command line program which is a part of MPlayer. It is available for all major platforms.

I got good results with the following command:

```
mencoder -oac lavc -ovc lavc \
  -of mpeg -mpegopts format=mpeg1 \
  -vf harddup -srate 44100 -af lavcresample=44100 \
  -lavcopts vcodec=mpeg1video:acodec=mp2 -ofps 25 -mc 0 -noskip \
  -o output.mpeg movie.file
```

11.2 Loading

scrupp.addMovie(filename)
: Loads the MPEG-1 movie file with the given *filename*. Returns a movie object.

11.3 Methods

movie:play([loop])
: Starts the movie playback. If the movie has an audio channel, music playback is paused and the audio of the movie is played back using the music channel. The optional boolean parameter *loop* enables (=true) or disables (=false, default) looping of the movie. Remember to call *movie:render()* in the render callback function!

movie:pause()
: Pauses the movie playback.

movie:resume()
: Resumes the movie playback.

movie:rewind()
: Rewinds the movie.

movie:stop()
: Stops the movie and gives the music channel free. After that, music can be played back again. Music that has been paused by the movie playback will be resumed.

movie:getWidth()
: Returns the width of the movie.

movie:getHeight()
: Returns the height of the movie.

movie:getSize()
: Returns the width and the height of the movie.

movie:getInfo()
: Returns a table containing information about the movie. This is the structure of the returned table:

```
table = {
```

```
        hasAudio          = boolean,
        hasVideo          = boolean,
        currentFrame      = integer,
        currentFPS        = number,  -- frames per second
        audioString       = string,  -- e.g. "MPEG-1 Layer 2 128kbit/s 44100Hz stereo"
        currentAudioFrame = integer,
        currentOffset     = integer,
        totalSize         = integer, -- filesize in bytes
        currentTime       = number,  -- in seconds
        totalTime         = number   -- in seconds
}
```

movie:isPlaying()

Returns true if the movie is playing, false otherwise.

movie:setAlpha(alpha)

Changes the alpha value of the movie. *alpha* has to be between 0 (transparent) and 255 (opaque).

movie:getAlpha()

Returns the current alpha value of the movie. This value is between 0 (transparent) and 255 (opaque).

movie:loadFirstFrame()

Loads the first frame of the movie. **Note:** This function does not display anything on the screen. It just updates the internal image that is rendered during the next call to *movie:render()*

movie:render(x,y)

Renders the movie at the window coordinates *x* and *y*.

movie:render(table)

This is nearly the same as the last one. This time a *table* contains the arguments. This function is the same as image:render(table) but renders the current movie frame, not an image. The complete structure of the table follows.

```
table = {
        x, -- x-coordinate of the movie frame position -- mandatory
        y, -- y-coordinate of the movie frame position -- mandatory
        centerX = [x-coordinate of the center for positioning, scaling and rotation], -- o
        centerY = [y-coordinate of the center for positioning, scaling and rotation], -- o
        scaleX = [scale factor for the x-axis], -- optional
        scaleY = [scale factor for the y-axis], -- optional
        rotate = [angle in degrees to rotate the movie frame], -- optional
        rect = { x, y, width, height }, -- optional
        color = { red, green, blue, alpha } -- optional
}
```

The first two elements of the array part of the *table* have to be the x- and y-coordinate of the point that the movie frame should be rendered at. The *table* may have an optional *rect* entry. This has to be a table describing the rectangle the movie frame should be clipped to. It contains the x- and y-coordinate of the upper left corner and the width and height of the rectangle inside the movie frame. The entries *centerX* and *centerY* are optional and default to zero, the left-upper corner of the movie frame. *scaleX* and *scaleY* are optional as well and default to one. *rotate* has a default value of zero dregrees and is optional, too. The optional *color* entry is a table defining some kind of color filter. If you have a grayscale movie, it will be colorized with this color. As always the table looks like this: {red, green, blue, [alpha]} with each value between 0 and 255. The default is opaque white, so the color of the movie frame is not changed.

movie:remove()

If the movie is playing, this function has the same effects as a call to *movie:stop()*. In addition, this function removes the movie from memory. After that, calling any method of the movie variable is forbidden and will raise an error.

11.4 Example

```
-- create a dummy window, because you have to call
-- scrupp.init() before you can load movies
scrupp.init("Movie Test", 10, 10, 32, false)

-- load a mpeg file
local movie = scrupp.addMovie("path_to_mpeg_file")

-- get the width and height of the movie
local width, height = movie:getSize()

-- resize the Scrupp window to the movie size
scrupp.init("Movie Test", width, height, 32, false)

-- play the movie in a loop
movie:play(true)

main = {
        render = function(dt)
                movie:render(0,0)
        end,

        keypressed = function(k)
                if k == "ESCAPE" then scrupp.exit() end
        end,
}
```

12 Mouse handling

12.1 Functions

scrupp.getMouseX()
: Returns the current x-coordinate of the mouse pointer.

scrupp.getMouseY()
: Returns the current y-coordinate of the mouse pointer.

scrupp.getMousePos()
: Returns the current x- and y-coordinate of the mouse pointer

scrupp.mouseButtonIsDown(button)
: Returns the boolean state of the *button*. *button* is one of the strings "left", "middle" or "right".

12.2 Callbacks

mousepressed(x, y, button)
: Gets called when a mouse button is pressed. Arguments are the *x*- and the *y*-coordinate of the mouse pointer and the pressed *button*. *button* is one of the strings "left", "middle", "right", "wheelUp" or "wheelDown".

mousereleased(x, y, button)
: Gets called when a mouse button is released. Arguments are the *x*- and the *y*-coordinate of the mouse pointer and the released *button*. *button* is one of the strings "left", "middle", "right", "wheelUp" or "wheelDown".

12.3 Example

```
scrupp.init("Mouse Test", 600, 400, 32, false)

require "font"

local font = Font("fonts/Vera.ttf", 20)
local text = ""

local cursor = scrupp.addImage("path_to_cursor_image")

main = {
	render = function(dt)
		font:print(10,10, text)
		if scrupp.mouseButtonIsDown("left") then
			cursor:render(scrupp.getMouseX(), scrupp.getMouseY())
			--or: cursor:render(scrupp.getMousePos())
		end
	end,

	mousepressed = function(x, y, button)
		text = button .." button pressed at "..x..", "..y
	end,

	mousereleased = function(x, y, button)
		text = button .." button released at "..x..", "..y
	end
}
```

13 Keyboard handling

13.1 Functions

scrupp.keyIsDown(key)
Returns the boolean state of the *key*. *key* is one of the key strings.

13.2 Callbacks

keypressed(key)
Gets called when a key is pressed. Argument is the key string of the pressed key.
keyreleased(key)
Gets calles when a key is released. Argument is the key string of the released key.

13.3 Key strings

The first table shows virtual key strings. These keys do **not** exist. You should not try to use them with the callbacks. Instead of e.g. "SHIFT", test for "LSHIFT" or "RSHIFT"

They should be used with *scrupp.keyIsDown()* to test for some special keys which happen to be twice on a standard keyboard. *scrupp.keyIsDown("SHIFT")* will return *true* if either of the two shift keys is pressed.

"SHIFT"	either of the shift keys
"CTRL"	either of the ctrl keys
"ALT"	either of the alt keys
"META"	either of the meta keys
"SUPER"	either of the windows keys

The following table shows all key strings usable with Scrupp.

"UNKNOWN"	unknown key
"BACKSPACE"	backspace
"TAB"	tab
"CLEAR"	clear
"RETURN"	return
"PAUSE"	pause
"ESCAPE"	escape
"SPACE"	space
"!"	exclaim
"""	quotedbl
"#"	hash
"$"	dollar
"&"	ampersand
"'"	quote
"("	left parenthesis
")"	right parenthesis
"*"	asterisk
"+"	plus sign

","	comma
"-"	minus sign
"."	period
"/"	forward slash
"0"	0
"1"	1
"2"	2
"3"	3
"4"	4
"5"	5
"6"	6
"7"	7
"8"	8
"9"	9
":"	colon
";"	semicolon
"<"	less-than sign
"="	equals sign
">"	greater-than sign
"?"	question mark
"@"	at
"["	left bracket
"\"	backslash
"]"	right bracket
"^"	caret
"_"	underscore
"`"	grave
"a"	a
"b"	b
"c"	c
"d"	d
"e"	e
"f"	f
"g"	g
"h"	h
"i"	i
"j"	j
"k"	k
"l"	l
"m"	m
"n"	n
"o"	o
"p"	p
"q"	q
"r"	r

"s"	s
"t"	t
"u"	u
"v"	v
"w"	w
"x"	x
"y"	y
"z"	z
"DELETE"	delete
"KP0"	keypad 0
"KP1"	keypad 1
"KP2"	keypad 2
"KP3"	keypad 3
"KP4"	keypad 4
"KP5"	keypad 5
"KP6"	keypad 6
"KP7"	keypad 7
"KP8"	keypad 8
"KP9"	keypad 9
"KP_PERIOD"	keypad period
"KP_DIVIDE"	keypad divide
"KP_MULTIPLY"	keypad multiply
"KP_MINUS"	keypad minus
"KP_PLUS"	keypad plus
"KP_ENTER"	keypad enter
"KP_EQUALS"	keypad equals
"UP"	up arrow
"DOWN"	down arrow
"RIGHT"	right arrow
"LEFT"	left arrow
"INSERT"	insert
"HOME"	home
"END"	end
"PAGEUP"	page up
"PAGEDOWN"	page down
"F1"	F1
"F2"	F2
"F3"	F3
"F4"	F4
"F5"	F5
"F6"	F6
"F7"	F7
"F8"	F8
"F9"	F9
"F10"	F10

"F11"	F11
"F12"	F12
"F13"	F13
"F14"	F14
"F15"	F15
"NUMLOCK"	numlock
"CAPSLOCK"	capslock
"SCROLLOCK"	scrollock
"RSHIFT"	right shift
"LSHIFT"	left shift
"RCTRL"	right ctrl
"LCTRL"	left ctrl
"RALT"	right alt
"LALT"	left alt
"RMETA"	right meta
"LMETA"	left meta
"LSUPER"	left windows key
"RSUPER"	right windows key
"MODE"	mode shift
"COMPOSE"	compose
"HELP"	help
"PRINT"	print-screen
"SYSREQ"	SysRq
"BREAK"	break
"MENU"	menu
"POWER"	power
"EURO"	euro
"UNDO"	undo

The source of this table is the SDL man page for SDLKey.

Additionally, there exist strings for keys on international keyboards. They are named from "WORLD_0" to "WORLD_95".

13.4 Example

```
scrupp.init("Keyboard Test", 600, 400, 32, false)

require "font"

local font = Font("fonts/Vera.ttf", 20)

local text = "Press any key."

main = {
	render = function(dt)
		font:print(10, 10, text)
	end,

	keypressed = function(key)
		text = key .. " pressed."
```

```
                -- the escape key exits the demo
                if key == "ESCAPE" then
                        scrupp.exit()
                end
        end,

        keyreleased = function(key)
                text = key .. " released."
        end
}
```

14 Network support

Scrupp 0.4 contains version 2.0.2 of the Lua network library LuaSocket written by Diego Nehab.

The website of LuaSocket contains an introduction and a reference.

The doc directory of the Scrupp distribution contains a local snapshot of the LuaSocket documentation.

Everything described in this documentation works exactly the same way in Scrupp.

14.1 Example

```
-- size of the window
local width, height = 600, 400

scrupp.init("LuaSocket Test", width, height, 32, false)

-- loads the HTTP module and any libraries it requires
local http = require("socket.http")

-- download a screenshot from the scrupp website
local img = http.request("http://scrupp.sourceforge.net/screenshots/linux_default_thumb.png")

-- convert the string to a scrupp image
img = scrupp.addImageFromString(img)

-- get the size of the image
local w, h = img:getSize()

-- calculate the coordinates of the image (for center position)
local x, y = (width-w)/2, (height-h)/2

main = {
	render = function(dt)
		-- render the screenshot at the center of the window
		img:render(x,y)
	end,

	keypressed = function(key)
		if key == "ESCAPE" then
			scrupp.exit()
		end
	end
}
```

15 Simple class implementation

This is a simple class implementation. It was taken from the lua-users wiki. The original author is unknown.

15.1 Functions

class(ctor)
: Creates and returns an object that represents the new class with the constructor *ctor*.

class(base, ctor)
: Creates and returns an object that represents the new class with the constructor *ctor* which inherits from the class *base*.

The returned objects can be used to create new instances. Any changes to the returned object will influence all instances of this class. An example usage of this feature is the implementation of methods after the creation of the class.

15.2 Constructor

If a new instance of a class is created, the constructor will be called with the new instance as first parameter followed by any additional parameters. The constructor can be used to set any values of the new instance to either default values or values which depend on the additional parameters.

15.3 Creation of instances

In order to create a new instance of a class just call the object returned by the *class* function with any optional arguments to the constructor.

15.4 Methods

Every instance of a new class implements the following methods.

obj:init(...)
: Calls the constructor of the class of the instance *obj* with the same arguments.

obj:is_a(class)
: Returns true if the *class* is a base class of the instance *obj*, false otherwise.

15.5 Example

```
scrupp.init("Class Test", 600, 400, 32, false)

-- load required plugin
require "font"

-- load a font
local font = Font("fonts/Vera.ttf", 20)

-- define a class for animals
local Animal = class(function(a, name)
        a.name = name
end)
-- define a method that returns the name of the animal
function Animal:getName()
        return self.name
end
```

```
-- define a class for dogs
local Dog = class(Animal, function(a, name, lord)
        a.name = name
        a.lord = lord
end)
-- define a method that returns the name of the lord of the dog
-- remark: the method getName() is taken from the base class
function Dog:getNameOfLord()
        return self.lord
end

-- create an instance of the class Dog
local Hasso = Dog("Hasso", "Mr. Smith")

main = {
        render = function(dt)
                font:print(10,10, "Name: ", Hasso:getName())
                font:print(10,40, "Lord: ", Hasso:getNameOfLord())

                -- tostring() is nescessary in order to print a boolean value
                font:print(10,70, "Hasso is an animal: ", tostring(Hasso:is_a(Animal)))
                font:print(10,100, "Hasso is a dog: ", tostring(Hasso:is_a(Dog)))
        end
}
```

16 Animation class

This class provides an easy to use interface for animations.

16.1 Requirements

- Class Plugin

16.2 Creation

Animation()
: Creates and returns a new animation.

16.3 Methods

animation:addFrame(image, x, y, width, height, delay)
: Adds a new frame to the *animation*. The *image* is the source of the frame. It can be either an Image or a string pointing to the image file. The parameters *x*, *y*, *width* and *height* define the rectangular area of the image which will be used as the frame. The duration of the frame in milliseconds is defined by *delay*.

animation:addFrames(image, sizex, sizey, width, height, sep, delay)
: Adds multiple frames stored in an *image* at once. The *image* can be either an Image or a string pointing to the image file. *sizex* and *sizey* define the number of frames in the source *image* in x- and y-direction. *sep* is the width in pixels of any separator between the frames. The duration of the frames in milliseconds is defined by *delay*.

animation:getWidth()
: Returns the width of the active frame.

animation:getHeight()
: Returns the height of the active frame.

animation:getSize()
: Returns the width and the height of the active frame.

animation:isTransparent(x, y)
: Returns true if the pixel with the coordinates *x* and *y* in the active frame is transparent, false otherwise.

animation:copy()
: Creates and returns a copy of the *animation*. This is useful when the same animation is used multiple times in parallel. By using a copy every animation has its own timing.

animation:start()
: Starts the *animation*.

animation:stop()
: Stops the *animation*

animation:render(x, y, delta)
: Renders the *animation* at the point defined by *x* and *y*. *delta* is the time in milliseconds passed since the last rendering. Usually this is exactly the *delta* passed to the render callback by Scrupp. See the example for clarification.

16.4 Example

You can download an example image that contains the frames of an animation which can be used with the example below: animation.png.

```
scrupp.init("Animation Test", 600, 400, 32, false)
```

```
-- load required plugins
require "animation"

-- create a new animation
local animation = Animation()

-- loads the frames from animation.png which contains 3x2 = 6
-- single frames, each with a size of 48x48 pixels, with no
-- separating pixels in between
-- each frame is shown for one second (1000 milliseconds)
animation:addFrames("animation.png", 3, 2, 48, 48, 0, 1000)

main = {
	render = function(dt)
		animation:render(10,10, dt)
	end
}
```

17 Colors

This plugin provides a table containing some often used colors.

17.1 Provided colors

- white
- black
- gray
- silver
- maroon
- red
- green
- lime
- olive
- yellow
- navy
- blue
- purple
- fuchsia
- teal
- aqua

17.2 Usage

The execution of *require "color"* creates a table, returns it and stores it in the global variable named *color*. This table contains the tables of the provided colors. They can be accessed by the name of the color. Each color table has the format {red, green, blue}, with each component being an integer from 0 to 255. These colors can be used everytime scrupp expects a color.

17.3 Example

```
scrupp.init("Color Test", 600, 357, 32, false)

-- load the color plugin and store it in a local variable
-- for faster access
local color = require "color"
-- alternative:
-- require "color"

-- this example uses the font plugin
require "font"
local font = Font("fonts/Vera.ttf", 20)

-- generate a rectangle
local rect = {
        100,  0, -- placeholder for the coordinates
        0,    0,
        495,  0,
        495, 18,
        0,   18,
        relative = true,
        fill = true
}

-- y-position of the displayed color
local y
```

```
main = {
	render = function(dt)
		y = 5
		-- cycle through all available colors
		for name, color in pairs(color) do
			-- print the name of the color
			font:print(10, y, name)
			-- change the color of the rectangle
			rect.color = color
			-- change the y-coordinate of the rectangle
			rect[2] = y
			-- draw the colored rectangle
			scrupp.draw(rect)
			y = y + 22
		end
	end
}
```

18 Font class

This class provides a high level interface to TTF fonts. If you want to print some text to the screen by only using the functions provided by the core of Scrupp, you will have to create a new image for every text. This is very inefficient if the text changes in every frame. This class creates an image for every single letter and prints text to the screen by placing them in the right order.

18.1 UTF-8

This plugin supports UTF-8. If you want to use this feature, remember to edit your scripts using the UTF-8 mode of your editor. Save them with UTF-8 encoding (without BOM).

18.2 Kerning

In typography, kerning is the process of adjusting letter spacing in a proportional font. In a well-kerned font, the two-dimensional blank spaces between each pair of letters all have similar area (source: Wikipedia). This plugin supports autokerning. You don't have to do anything, your texts will just look right.

18.3 Requirements

- Class Plugin

18.4 Creation

Font(filename, size)
: Creates and returns a new font instance. The TrueType font is loaded using *filename* and the *size* in pixels. The default color of the font is white.

18.5 Methods

Font:getTextSize(text)
Font:getHeight()
Font:getLineSkip()
Font:generateImage(text)
Font:generateImage(table)
: These functions are wrappers of the Scrupp core functions for fonts with the same name.

Font:setColor(color)
: Changes the color of the font to the specified value. *color* has to be a table of the format {red, green, blue, [alpha]} with each component being a number between 0 and 255.

Font:cache([str])
: After the creation of a new instance the cache with the images of the letters is empty. Normally it gets filled by using Font:print because every letter that is not in the cache is stored there automatically. By calling Font:cache this can be done manually. Every letter of the string *str* is added to the cache. If *str* is omitted, it is set to its default value which contains all upper- and lowercase letters, numbers and "!?()[]{}.,;:-_"

Font:print(x, y, ...)
: Prints its arguments at the position defined by *x* and *y*. Any number of parameters can follow these two. For every letter that is not in the cache an image is created and added to it on the fly.

18.6 Example

```
scrupp.init("Font Test", 600, 400, 32, false)

-- load required plugin
require "font"

local font = Font("fonts/Vera.ttf", 20)

main = {
	render = function(dt)
		font:print(10,10, "Hello world!")
		font:print(10,40, "Hello ", "world!")
		font:print(10,70, "Strings with embedded newlines\nare supported as well.")
	end
}
```

19 Timer class

This class provides an implementation of timers.

19.1 Requirements

- Class Plugin

19.2 Creation

Timer(duration, callback)
: Creates and returns a new timer instance. The timer expires after the specified *duration* in milliseconds. It calls the function *callback* on this event. After the creation the timer is stopped and has to be started by calling Timer:start().

19.3 Methods

Timer:start()
: Starts the timer.

Timer:stop()
: Stops the timer. The elapsed time is saved.

Timer:reset()
: Resets the timer.

Timer:update()
: Updates the timer. If time is up it will call the *callback* function with the timer object as argument.

19.4 Example

```
scrupp.init("Timer Test", 600, 400, 32, false)

-- load required plugins
require "font"
require "timer"

local font = Font("fonts/Vera.ttf", 20)

-- this variable is used to distinguish between tick and tack :)
local tick = true

-- create a new timer with a duration of 1 second
-- (= 1000 milliseconds)
-- the callback negates the tick variable
local timer = Timer(1000, function(timer)
        tick = not tick
end)

-- start the timer
timer:start()

main = {
        render = function(dt)
                -- update the timer
                timer:update()

                -- print Tick or Tack depending on the state of the
                -- tick variable
                if tick then
```

```
				font:print(10,10, "Tick!")
			else
				font:print(10,10, "Tack!")
			end
		end
}
```