

Scrapp 0.2 Manual

Table of Contents

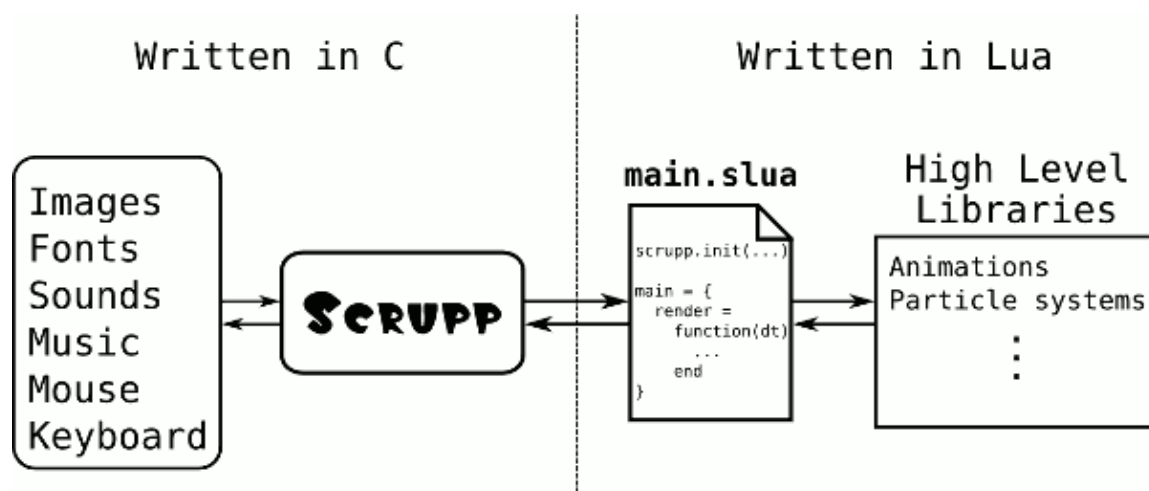
<u>1 Getting Started</u>	1
<u>1.1 Architecture</u>	1
<u>1.2 Startup</u>	1
<u>1.3 Script Arguments</u>	3
<u>2 Callbacks</u>	4
<u>2.1 Overview</u>	4
<u>2.2 Callbacks</u>	4
<u>2.3 Example</u>	4
<u>3 General Functions</u>	6
<u>3.1 General functions</u>	6
<u>3.2 Example</u>	7
<u>4 Image handling</u>	8
<u>4.1 Loading</u>	8
<u>4.2 Methods</u>	8
<u>4.3 Example</u>	9
<u>5 Font handling</u>	10
<u>5.1 Note</u>	10
<u>5.2 Loading</u>	10
<u>5.3 Methods</u>	10
<u>5.4 Example</u>	10
<u>6 Sound handling</u>	12
<u>6.1 Loading</u>	12
<u>6.2 Methods</u>	12
<u>6.3 Example</u>	12
<u>7 Music handling</u>	13
<u>7.1 Loading</u>	13
<u>7.2 Methods</u>	13
<u>7.3 Functions</u>	13
<u>7.4 Example</u>	14
<u>8 Mouse handling</u>	15
<u>8.1 Functions</u>	15
<u>8.2 Callbacks</u>	15
<u>8.3 Example</u>	15
<u>9 Keyboard handling</u>	16
<u>9.1 Functions</u>	16
<u>9.2 Callbacks</u>	16
<u>9.3 Key strings</u>	16
<u>9.4 Example</u>	19
<u>10 Simple class implementation</u>	21
<u>10.1 Functions</u>	21
<u>10.2 Constructor</u>	21
<u>10.3 Creation of instances</u>	21
<u>10.4 Methods</u>	21

Table of Contents

<u>10 Simple class implementation</u>	
<u>10.5 Example</u>	21
<u>11 Animation class</u>	23
<u>11.1 Requirements</u>	23
<u>11.2 Creation</u>	23
<u>11.3 Methods</u>	23
<u>11.4 Example</u>	23
<u>12 Font class</u>	25
<u>12.1 Requirements</u>	25
<u>12.2 Creation</u>	25
<u>12.3 Methods</u>	25
<u>12.4 Example</u>	25
<u>13 Timer class</u>	27
<u>13.1 Requirements</u>	27
<u>13.2 Creation</u>	27
<u>13.3 Methods</u>	27
<u>13.4 Example</u>	27

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 Startup

The behaviour of Scrupp depends on the command line arguments and the operating system. If you drop a file or directory on the Scrupp executable, it's like this file or directory is the first command line argument.

On startup, Scrupp constructs a search path. This search path is a list of paths and zip archives. If Scrupp tries to open a file, it looks for it in these paths/archives. If the file exists in multiple paths, the first one is chosen.

In order to understand the way Scrupp constructs the search path and sets the working directory the following definitions are needed:

base directory

The directory that contains the executable.

working directory

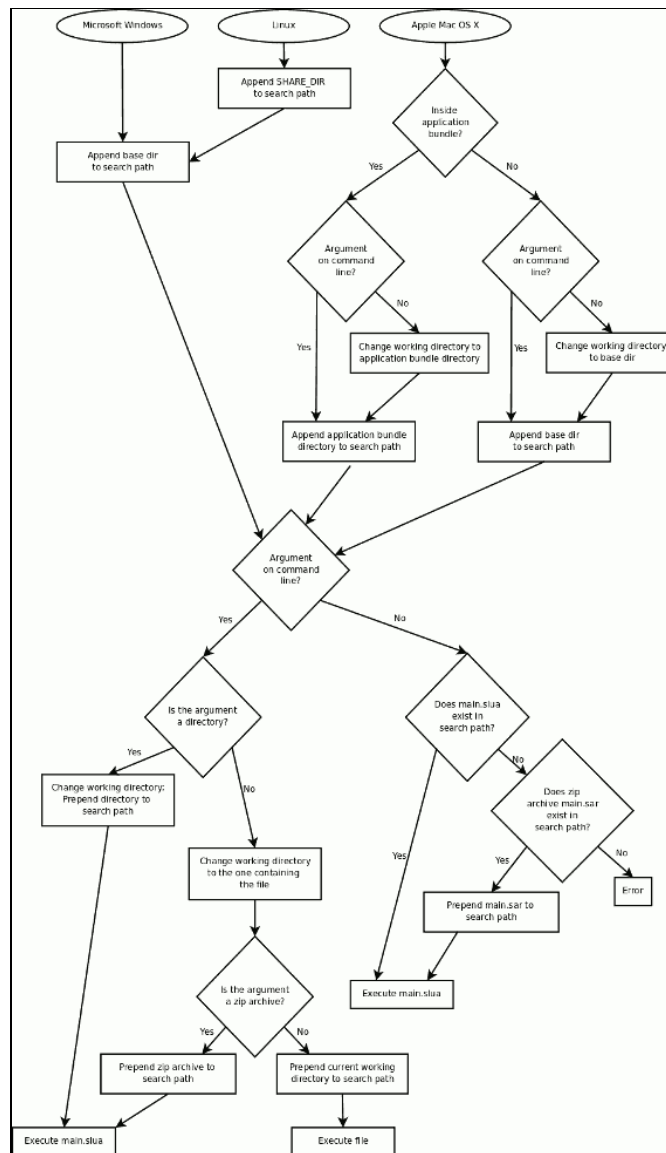
This directory is important if you want to modify files using the input and output facilities of plain Lua.

SHARE_DIR

On Linux, the SHARE_DIR defaults to `/usr/share/scrupp` (defined in `Main.h`). If you copy the support files ("main.lua" and the directories "fonts" and "scripts") to `/usr/share/scrupp` they can be found by Scrupp.

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.

Scrapp 0.2 Manual



The following list shows the possibilities for running Scrapp.

1. No command line arguments

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

Scrapp tries to open the file `main.lua`. The working directory of the script will be the one containing the executable. If this file is not found, Scrapp will try to open the zip archive `main.sar` which has to contain the file `main.lua`. 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

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

Scrapp 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

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

The archive is prepended to the search path. The directory that contains the archive becomes the

Scrapp 0.2 Manual

working directory. After this Scrapp tries to execute `main.lua` which should be located inside this archive. It is possible to give extra arguments to the Lua file.

4. A directory as argument

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

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

1.3 Script Arguments

Before running a Lua file, Scrapp 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 Scrapp
scrapp.exit()
```

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

```
$ scrapp 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 Scrapp.

2 Callbacks

2.1 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.

2.2 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.

mouserreleased(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.

2.3 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,

    mouserreleased = function(x, y, button)
        print(button .. " button released at "..x..", "..y)
    end
}
```

Scrapp 0.2 Manual

```
end,  
  
keypressed = function(k)  
    if k == "ESCAPE" then  
        scrapp.exit()  
    end  
    print("key '" .. k .. "' pressed")  
end,  
  
keyreleased = function(k)  
    print("key '" .. k .. "' released")  
end  
}
```


3 General Functions

3.1 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.

scrupp.addFile(filename)

Runs the Lua file with the chosen *filename*. Returns all values returned by the script.

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}]

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, pixels with a *size* greater one will become circles and 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 points will be interpreted as one or more line(s) or as a polygon depending on the other options. The default value is false.

3.2 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
}
```

4 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](#)

4.1 Loading

scrupp.addImage(filename)

Loads the image file with the given *filename*. Supported formats: BMP, PNM (PPM/PGM/PBM), XBM, LBM, PCX, GIF, JPEG, PNG and TIFF. Returns an image object.

4.2 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.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],
    centerY = [y-coordinate of the center for positioning, scaling and rotation],
    scaleX = [scale factor for the x-axis],
    scaleY = [scale factor for the y-axis],
    rotate = [angle in degrees to rotate the image],
    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 upper 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 left-upper corner of the image. *scaleX* and *scaleY* are optional as well and default to one. *rotate* has a default value of zero degrees 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.

4.3 Example

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

-- load an image file
local image = scrapp.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 > scrapp.getWindowWidth() or height > scrapp.getWindowHeight() then
    print("Please choose an image with smaller dimensions.")
    scrapp.exit()
end

-- calculate the x- and y-coordinate of the image
local x = (scrapp.getWindowWidth()-width)/2
local y = (scrapp.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/scrapp.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
end
}
```

5 Font handling

5.1 Note

Have a look at the [font plugin](#). This make the usage of truetype fonts easier.

5.2 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.

5.3 Methods

font:getTextSize(text)

Returns the width and height of the resulting surface of the LATIN1 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](#).

5.4 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:
```

Scrapp 0.2 Manual

```
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 parentheses 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
}
```

6 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!

6.1 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.

6.2 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()*.

6.3 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
end
}
```

7 Music handling

The music support is implemented using [SDL_mixer](#).

Scrapp 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.

7.1 Loading

scrapp.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>
```

7.2 Methods

music:play([loops=0], [fade_in_time=0])

Plays a music file loaded with *scrapp.addMusic()*. The optional parameter *loops* defines the number of times the music will be played. 0 means infinite 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.

7.3 Functions

scrapp.setMusicVolume(volume)

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

scrapp.getMusicVolume()

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

scrapp.pauseMusic()

Pauses the music playback. Paused music may be stopped with *scrapp.stopMusic()*.

scrapp.resumeMusic()

Unpauses the music. This is safe to use on stopped, paused and playing music.

scrapp.stopMusic([fade_out_time=0])

Stop the playback of music. The optional parameter *fade_out_time* gives the number of milliseconds the music will fade out. It defaults to 0 ms.

scrapp.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.

scrapp.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*.

scrapp.musicIsPaused()

Tells you if music is paused, or not.

Note: This state is only influenced by calls to *scrapp.pauseMusic()* and *scrapp.resumeMusic()*.

7.4 Example

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

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

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

8 Mouse handling

8.1 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".

8.2 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".

moureleased(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".

8.3 Example

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

scrupp.addFile("scripts/class.lua")
scrupp.addFile("scripts/font.lua")

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,

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

9 Keyboard handling

9.1 Functions

`scrupp.keyIsDown(key)`

Returns the boolean state of the *key*. *key* is one of the key strings.

9.2 Callbacks

`keypressed(key)`

Gets called when a key is pressed. Argument is the key string of the pressed key.

`keyreleased(key)`

Gets called when a key is released. Argument is the key string of the released key.

9.3 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
"SCROLLLOCK"	scrolllock
"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".

9.4 Example

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

scrupp.addFile("scripts/class.lua")
scrupp.addFile("scripts/font.lua")

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."
    end
}
```

Scrapp 0.2 Manual

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

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

10 Simple class implementation

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

10.1 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.

10.2 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.

10.3 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.

10.4 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.

10.5 Example

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

-- load required plugins
scrupp.addFile("scripts/class.lua")
scrupp.addFile("scripts/font.lua")

-- 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
```


Scrapp 0.2 Manual

```
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 necessary 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
}
```

11 Animation class

This class provides an easy to use interface for animations.

11.1 Requirements

- Class Plugin

11.2 Creation

Animation()

Creates and returns a new animation.

11.3 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.

11.4 Example

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

-- load required plugins
scrupp.addFile("scripts/class.lua")
scrupp.addFile("scripts/animation.lua")
```

Scrapp 0.2 Manual

```
local animation = Animation()

-- loads the frames from animation.png which contains 5x5 = 25
-- single frames, each with a size of 20x20 pixels, with no
-- separating pixels in between
-- each frame is shown for 150 ms
animation:addFrames("animation.png", 5, 5, 20, 20, 0, 150)

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

12 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.

12.1 Requirements

- Class Plugin

12.2 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.

12.3 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.

12.4 Example

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

-- load required plugins
scrupp.addFile("scripts/class.lua")
scrupp.addFile("scripts/font.lua")

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
end
```

}

13 Timer class

This class provides an implementation of timers.

13.1 Requirements

- Class Plugin

13.2 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()`.

13.3 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.

13.4 Example

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

-- load required plugins
scrupp.addFile("scripts/class.lua")
scrupp.addFile("scripts/font.lua")
scrupp.addFile("scripts/timer.lua")

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
```

Scrapp 0.2 Manual

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