Scrupp 0.1 Manual

Table of Contents

1 Getting Started......1
- 1.1 Architecture......1
- 1.2 Command Line Interface......1
- 1.3 Script Arguments......2

2 Callbacks......3
- 2.1 Overview......3
- 2.2 Callbacks......3
- 2.3 Example......3

3 General Functions......5
- 3.1 General functions......5
- 3.2 Example......5

4 Image handling......7
- 4.1 Loading......7
- 4.2 Methods......7
- 4.3 Example......7

5 Font handling......9
- 5.1 Loading......9
- 5.2 Methods......9
- 5.3 Examples......9

6 Sound handling......11
- 6.1 Loading......11
- 6.2 Methods......11
- 6.3 Example......11

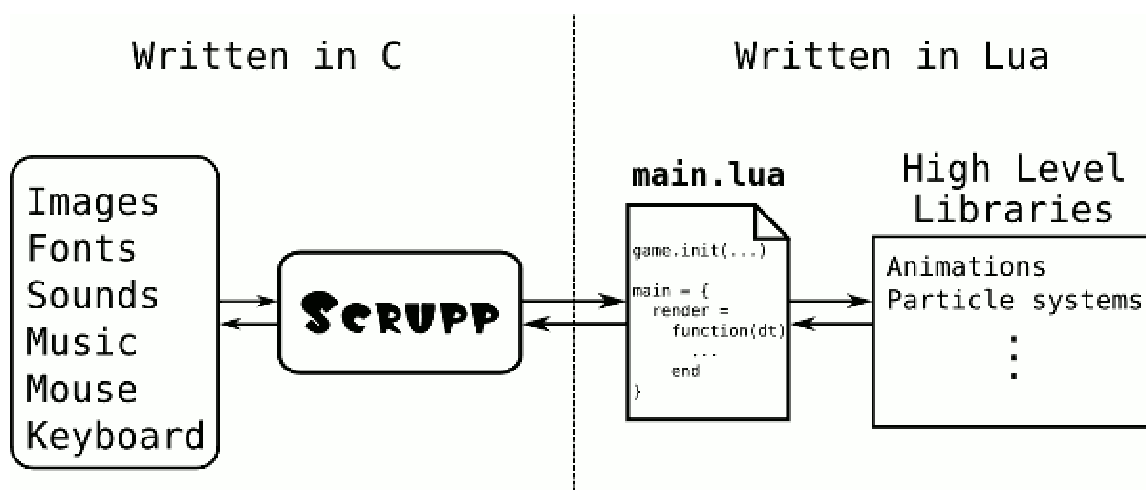
7 Music handling......12
- 7.1 Loading......12
- 7.2 Methods......12
- 7.3 Functions......12
- 7.4 Example......13

8 Mouse handling......14
- 8.1 Functions......14
- 8.2 Callbacks......14
- 8.3 Example......14

9 Keyboard handling......15
- 9.1 Functions......15
- 9.2 Callbacks......15
- 9.3 Key constants......15
- 9.4 Example......18

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 Command Line Interface

At the beginning, Scrupp tries to run a Lua file which is chosen depending on the command line arguments.

1. No command line arguments

   ```
   $ scrupp
   ```

   In this case Scrupp tries to open the file `main.lua` in the directory where the Scrupp executable is located. This directory will also be the working directory of the script. However, it is not possible to give extra arguments to the Lua file, yet.
2. A Lua file as argument

   ```
   $ scrupp -e <Lua file> arg1 arg2 ...
   ```

   Scrupp adds the directory that contains the Lua file to the search path. Every attempt to open a file (images, sounds, music, fonts, other Lua files) will result in a search for it in this directory. In case the file is not found there it will be searched in the directory of the executable. After that the Lua file is executed. Now it is possible to give extra arguments to the Lua file.
3. A directory as argument

   ```
   $ scrupp <directory> arg1 arg2 ...
   ```

   The directory is added to the search path and set as the working directory. Every file access results in a search in this directory at first. Only if it is not found there the file will be searched in the directory of the executable. Then Scrupp tries to execute `main.lua` which should be located inside this directory. Now it is possible to give extra arguments to the Lua file.
4. A zip archive as argument

   ```
   $ scrupp <zip archive> arg1 arg2 ...
   ```

The archive is added to the search path. The directory containing the archive becomes the working directory. Every file access results in a search in this archive at first. The directory of the executable is searched after that. Then Scrupp tries to execute `main.lua` which should be located inside this archive. It is possible to give extra arguments to the Lua file.

1.3 Script Arguments

Before running the 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. Any arguments before this name (name of the executable, the options) go to negative indices. 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
game.exit()
```

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

```
$ scrupp -e 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 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 20 ms long because Scrupp tries to make constant 50 frames per second. This information can be used to time animations. The rendering of images should happen in this function.

keypressed(key)
: This callback function is optional. If a key is pressed then 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 then 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 then 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 then this function will be executed. Read the section about Mouse usage for details and an example.

2.3 Example

```
game.init("Callback Test", 600, 400, 32, false)

image = game.addImage(<image file>)

main = {
        render = function(dt)
                image:render(mouse.getX(), mouse.getY())
        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 == key.ESCAPE then
                        game.exit()
                end
                print("key with keycode " .. k .. " pressed")
        end,

        keyreleased = function(k)
                print("key with keycode " .. k .. " released")
```

```
        end
}
```

3 General Functions

3.1 General functions

game.init(title, width, height, bit_depth, fullscreen)
: 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 windowed mode (*false*).

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

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

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

game.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*.

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

game.addFile(filename)
: Runs the file with the chosen *filename*.

game.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.

  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

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

-- prepare one table for big white pixels
pixels = { antialiasing = false, size = 20, pixellist = true }
-- prepare one table for red connected lines
```

```
lines = { color = {255,0,0}, antialiasing = true, connect = true }

main = {
	render = function(dt)
		game.draw(lines)
		game.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

game.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. 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.

4.3 Example

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

-- load an image file
image = game.addImage(<image file>)

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

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

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

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

5 Font handling

5.1 Loading

game.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.

5.2 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.

font:generateImage(table)

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

font:print(x, y, text)

Prints the given *text* at the coordinates *x* and *y*. The text color is white and the background is transparent. To use another color call *font:print()* with a table as sole argument.
Note: This function is REALLY slow! That is due to the creation of an image everytime it is called. In fact this function behaves like calling *font:generateImage(text)* and then rendering the returned image using *image:render(x,y)*. I don't recommended to use this function!

font:print(table)

This is nearly the same as the last one. This time a *table* contains the arguments. The first three elements of the array part of the *table* have to be the x- and y-coordinate and 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).
Note: This function is REALLY slow! That is due to the creation of an image everytime it is called. In fact this function behaves like calling *font:generateImage(table)* and then rendering the returned image using *image:render(x,y)*. I don't recommended to use this function!

5.3 Examples

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

-- load a font
font = game.addFont("fonts/Vera.ttf", 20)
-- get the recommended line height in pixel
lineSkip = font:getLineSkip()
-- get the text size of a sample text
w, h = font:getTextSize("Hello World")

-- define a color (opaque blue)
```

```
cBlue = {0, 0, 255, 255}
-- the transparency defaults to 255 (opaque), so this is the same:
cBlue = {0, 0, 255}

-- generate an image containing some text using the default color
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
image_2 = font:generateImage{"Hello World", color = cBlue}
-- this has the same result:
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)
		-- render some text using font:print():
		font:print(100, 50 + 2*lineSkip, "Hello World")
		font:print{100 + w, 50 + 3*lineSkip, "Hello World", color = cBlue}
	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

game.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 *game.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

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

-- load a sound file
sound = game.addSound(<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
}
```

7 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.

7.1 Loading

game.addMusic(filename)

Loads a music file with the given *filename*. This can load WAV, MOD, MIDI, 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 *game.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.

7.3 Functions

game.setMusicVolume(volume)

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

game.getMusicVolume()

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

game.pauseMusic()

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

game.resumeMusic()

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

game.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.

game.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, OGG, MP3, Native MIDI.

game.isMusicPlaying()

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*.

game.isMusicPaused()

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

7.4 Example

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

-- load some music
music = game.addMusic(<music file>)
-- start music
music:play()

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

8 Mouse handling

8.1 Functions

mouse.getX()
Returns the current x-coordinate of the mouse pointer.

mouse.getY()
Returns the current y-coordinate of the mouse pointer.

mouse.getPos()
Returns the current x- and y-coordinate of the mouse pointer

mouse.isDown(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".

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".

8.3 Example

```
game.init("Mouse Test", 600, 400, 32, false)
cursor = game.addImage(<path to cursor image>)

main = {
        render = function(dt)
                if mouse.isDown("left") then
                        cursor:render(mouse.getX(), mouse.getY())
                        --or: cursor:render(mouse:getPos())
                end
        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
}
```

9 Keyboard handling

9.1 Functions

key.isDown(key)

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

9.2 Callbacks

keypressed(key)

Gets called when a key is pressed. Argument is the key constant of the pressed *key*.

keyreleased(key)

Gets calles when a key is released. Argument is the key constants of the released *key*.

9.3 Key constants

The first table shows virtual key constants. These keys do **not** exist. You should not try to use them with the callbacks because they will never occur. Instead of e.g. *key.SHIFT*, test for *key.LSHIFT* or *key.RSHIFT*

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

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

The following table shows all key constants usable with Scrupp.

key.BACKSPACE	'\b'	backspace
key.TAB	'\t'	tab
key.CLEAR		clear
key.RETURN	'\r'	return
key.PAUSE		pause
key.ESCAPE	'^['	escape
key.SPACE	' '	space
key.EXCLAIM	'!'	exclaim
key.QUOTEDBL	'"'	quotedbl
key.HASH	'#'	hash
key.DOLLAR	'$'	dollar
key.AMPERSAND	'&'	ampersand
key.QUOTE	'''	quote
key.LEFTPAREN	'('	left parenthesis
key.RIGHTPAREN	')'	right parenthesis
key.ASTERISK	'*'	asterisk
key.PLUS	'+'	plus sign
key.COMMA	','	comma

key.MINUS	'-'	minus sign
key.PERIOD	'.'	period
key.SLASH	'/'	forward slash
key.0	'0'	0
key.1	'1'	1
key.2	'2'	2
key.3	'3'	3
key.4	'4'	4
key.5	'5'	5
key.6	'6'	6
key.7	'7'	7
key.8	'8'	8
key.9	'9'	9
key.COLON	':'	colon
key.SEMICOLON	';'	semicolon
key.LESS	'<'	less-than sign
key.EQUALS	'='	equals sign
key.GREATER	'>'	greater-than sign
key.QUESTION	'?'	question mark
key.AT	'@'	at
key.LEFTBRACKET	'['	left bracket
key.BACKSLASH	'\'	backslash
key.RIGHTBRACKET	']'	right bracket
key.CARET	'^'	caret
key.UNDERSCORE	'_'	underscore
key.BACKQUOTE	'`'	grave
key.a	'a'	a
key.b	'b'	b
key.c	'c'	c
key.d	'd'	d
key.e	'e'	e
key.f	'f'	f
key.g	'g'	g
key.h	'h'	h
key.i	'i'	i
key.j	'j'	j
key.k	'k'	k
key.l	'l'	l
key.m	'm'	m
key.n	'n'	n
key.o	'o'	o
key.p	'p'	p
key.q	'q'	q
key.r	'r'	r
key.s	's'	s

key.t	't'	t
key.u	'u'	u
key.v	'v'	v
key.w	'w'	w
key.x	'x'	x
key.y	'y'	y
key.z	'z'	z
key.DELETE	'^?'	delete
key.KP0		keypad 0
key.KP1		keypad 1
key.KP2		keypad 2
key.KP3		keypad 3
key.KP4		keypad 4
key.KP5		keypad 5
key.KP6		keypad 6
key.KP7		keypad 7
key.KP8		keypad 8
key.KP9		keypad 9
key.KP_PERIOD	'.'	keypad period
key.KP_DIVIDE	'/'	keypad divide
key.KP_MULTIPLY	'*'	keypad multiply
key.KP_MINUS	'-'	keypad minus
key.KP_PLUS	'+'	keypad plus
key.KP_ENTER	'\r'	keypad enter
key.KP_EQUALS	'='	keypad equals
key.UP		up arrow
key.DOWN		down arrow
key.RIGHT		right arrow
key.LEFT		left arrow
key.INSERT		insert
key.HOME		home
key.END		end
key.PAGEUP		page up
key.PAGEDOWN		page down
key.F1		F1
key.F2		F2
key.F3		F3
key.F4		F4
key.F5		F5
key.F6		F6
key.F7		F7
key.F8		F8
key.F9		F9
key.F10		F10
key.F11		F11

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

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

9.4 Example

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

main = {
	render = function(dt)
		-- we need at least an empty render function
	end,

	keypressed = function(k)
		-- check if a valid letter key is pressed;
		-- string.byte is a Lua function that returns
		-- the ASCII value of a char
		if k>=string.byte('a') and k<=string.byte('z') then
			-- if a letter key is pressed, then the argument k
			-- is a number (the ASCII code);
			-- string.char is a Lua function that returns
			-- the char of such a code
			local ch = string.char(k)
			-- check whether any of the two shift keys is pressed
			if key.isDown(key.SHIFT) then
				-- make the letter uppercase
				ch = string.upper(ch)
			end
			-- print the letter of the key;
			-- string.char is a Lua function that returns
```

```
                        -- the char of a ASCII value
                        print(ch .. " pressed")
                end

                -- the escape key exits the demo
                if k == key.ESCAPE then
                        game.exit()
                end
        end,

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