SLAYER

Multimedia programming platform for Guile Scheme
Version 1.0.0, updated 17 November 2013

Panicz Maciej Godek

Table of Contents

The Slayer Platform

1 Introduction

When I was 14, I felt a desperate urge to program computer games. I collected $25 to buy a book that would teach me programming. It was a huge brick titled "Teach yourself Visual Basic 6.0 in 21 days", and I soon figured out that I'd need $1000 more to obtain the tools that would allow me to use the knowledge contained in the book. That I could not afford.

Times have changed a lot since then. Windows 95 is no longer a dictator on desktop computers, there are many excellent free tools and libraries for developers (often much better than the proprietary ones), and knowledge on the Web is available to virtually everyone.

However, many tools are much more difficult to learn than they could be, or aren't available on as many systems as one could wish. Furthermore, existing soultions frequently impose many restrictions on the way the things can be done.

SLAYER was conceived to address those issues. It is meant to be an accessible, portable and extensible multimedia environment that would be suitable for learning as well as game development and unconventional or highly customizable GUI/multimedia applications. It uses SDL for portability and is built on top of Guile Scheme for accessibility and extensibility.

In a way SLAYER can resemble fluxus[1]. There are however some essential differences between those systems. Fluxus is intended to work mainly with 3D-graphics, while SLAYER works with 2D objects just as fine. Fluxus has a built-in editor, while SLAYER is kept minimalistic and only allows to add an editor widget to the stage, which perhaps makes it more flexible. (By the way, such widget is already provided and used in one of the demos.) Also, fluxus is focused on live coding and sound processing, but there are no good sound processing libraries for SLAYER yet.

Lastly, SLAYER works with Guile Scheme, and fluxus uses Racket. These are both Scheme-based languages, but their extensions are incompatible with each other. They're both cool, though.

1.1 Getting started

This section assumes that SLAYER is already installed and properly configured on your system, and that you are familiar with programming in general, and Guile Scheme in particular.

To get the grasp on how SLAYER works, we'll create a simple application for browsing images[2].

In order to do so, we need to create a text file; let's call it "image-browser.scm"[3]. Let's fill the file with the following content:

```
(use-modules (slayer) (slayer image)
     (srfi srfi-1) (srfi srfi-2))
(keydn 'esc quit)

(define (list-directory path)
  (let ((dir (opendir path)))
    (unfold eof-object? (lambda(f)(string-append path f))
    (lambda x (readdir dir)) (readdir dir))))

(define (file? f) (eq? 'regular (stat:type (stat f))))
```

[1] www.pawfal.org/fluxus/

[2] The idea for this program was inspired by Thien-Thi Nguyen, who placed a similar program in the manual for the Guile-SDL package

[3] SLAYER uses Scheme programming language, so it is convinient to name files with .scm extension, because various editors can detect that and load appropreate editing mode

```
(define *directory* (if (defined? '$1) $1 "/usr/share/pixmaps/"))
(define *image-names* (filter file? (list-directory *directory*)))
(define *current-image* #f)
(define *image-index* 0)

(if (<= (length *image-names*) 0)
    (begin (display "no images found\n") (quit)))

(set-display-procedure! (lambda () (draw-image! *current-image* 0 0)))

(define (show-image! i)
  (set! *image-index* (modulo i (length *image-names*)))
  (and-let* ((image-name (list-ref *image-names* *image-index*))
     (image (load-image image-name)))
    (set! *current-image* image)
    (apply set-screen-size! (image-size *current-image*))
    (set-window-title! image-name)))

(show-image! 0)

(keydn 'left (lambda () (show-image! (+ *image-index* 1))))
(keydn 'right (lambda () (show-image! (- *image-index* 1))))
```

In order to execute the program, one shall type:

```
$ slayer image-browser.scm
```

The argument to `set-display-procedure!` is executed after a series of events has been processed – so evey time an event occurs (like keyboard or mouse input, or window resize, or timer event), the whole screen is wiped and its content is displayed on the screen from scratch.

While this might seem a waste of resources, this is a desired behaviour for the class of applications that SLAYER aims to deliver, i.e. 3d games and real-time multimedia processing apps. Since the input can be gathered during the redisplay stage, the applications generally tend to be responsive.

1.2 Invoking SLAYER

Type `slayer --help` to get the list of possible options.

(TODO: this section definitely requires some elaboration)

1.3 Trying out the demos

In addition to the core SLAYER and some libraries, SLAYER comes with a set of demos that show how easily complex things can be achieved. They are located in the "demos" subdirectory. Note however, that the demos are available only if you obtained SLAYER source from the repository, and are not contained in the release tarball.

If you manage to build SLAYER (even without installing it), you can run the demos with the following commands (assuming you're in the "demos" dir):

```
$ ./slayer # equivalent to ./slayer slayer.scm
```

```
$ ./slayer -e3d
```

```
$ ./slayer pong.scm
```

If you have Open Dynamics Engine installed on your system, you can also try building the `(scum physics)` module by entering "src/scum" dir and typing

```
$ make
```

If you manage to build it successfully, then, back in the "demos" dir, you can try it out by typing:

```
$ ./slayer -e3d ode.scm
```

Note however, that the `(scum physics)` module isn't bundled with the tarball and needs to be obtained from the mercurial repository:

```
$ hg clone http://hg.gnu.org.ua/hgweb/slayer
```

More importantly than just playing with the demos (which, honestly, can become boring quite quickly), you are encouraged to read their source code (which, in turn, is a fascinating and endless endeavour). In particular, the "guile-modules/widgets/3d-view.scm" module defines a file format for representing 3d mesh (an example can be found in the "demos/art/3d/cube.3d" file), and the scum/physics.scm defines a "define-rigs-for" procedure, which is used to implement a file format for representing rigs (see "demos/art/rigs/car.rig" to see a more advanced example, or "demos/arg/rigs/ground.rig" for hopelessly simple one)

2 Direct Layer

In order to make SLAYER useful, you need to attach event handlers to certain events. If you are willing to create a real-time simulation, you may also wish to run a timer that generates its own events. In this chapter, you will find a description of all procedures available directly from SLAYER, divided into three groups:

2.1 Input handling and screen management

Once SLAYER is up and running, it has one window where everything is displayed, and which gathers all the input. If you want to use procedures described here, you need to `(use-modules (slayer))`.

In order to get something displayed on the screen, you need to set a display procedure of your liking.

`set-display-procedure!` *thunk* [Procedure]
Sets *thunk* to be called after each series of events is handled and the screen is wiped.

`screen-size` [Procedure]
Return a list containing width and height of the window, in pixels.

`set-screen-size!` *width height* [Procedure]
Set the screen size to *width* times *height* pixels. It causes a screen resize event, which can be handled using `set-resize-procedure!`.

`set-resize-procedure!` *proc* [Procedure]
Sets the binary procedure *proc* to be invoked on window resize event. *proc* takes two arguments, namely – the new width and height of the screen.

`set-exit-procedure!` *proc* [Procedure]
Sets the *proc* procedure to be called on exit. Due to historical reasons, *proc* currently takes one argument, which should be ignored.

`set-window-title!` *string* [Procedure]
Sets the window caption to *string*. This can only be visible if SLAYER is run in a window manager, in non-fullscreen mode.

`keydn` *key* [*thunk*] [Procedure]
Set *thunk* to be invoked whenever *key* is pressed (including mouse buttons). *key* can either be a string or a symbol. If *thunk* is not given, or is given but is not a procedure, it returns the current procedure set for a given *key*. Section 2.1.1 [Key names], page 6 table to get the names of specific keys.

`keyup` *key thunk* [Procedure]
This function works exactly as `keydn`, except that the *thunk* is invoked on *key* release.

`mousemove` *proc* [Procedure]
Set *proc* to be invoked whenever mouse is moved. *proc* takes four arguments: `(x y dx dy)`, where `(x y)` is the current mouse position and `(dx dy)` is the difference between the previous and the current position. If *proc* not given, returns the current procedure.

`modifier-pressed?` *modifier* [Procedure]
Checks whether *modifier* is pressed. *modifier* is a symbol that can be either `shift`, `rshift`, `lshift`, `alt`, `lalt`, `ralt`, `ctrl`, `lctrl`, `rctrl`, `meta`, `lmeta`, `rmeta`.

`input-mode` [Procedure]
Return the symbol describing current input mode, which can either be `'typing` or `'direct`. Direct mode is the default for SLAYER.

`set-direct-input-mode!` [Procedure]
Sets the input mode to "direct", which means that events are generated only when keys are actually pressed, which is the desired behaviour for most games, and the default behaviour of SLAYER.

`set-typing-input-mode!` [Procedure]
Sets the input mode to "typing", which means that the keyboard input behaves like when typing in an editor – once a key is pressed, after a certain time it gets repeated at a certain frequency. In typing mode, pressing printable characters do not cause the procedures specified with `keydn` and `keyup` to be called; instead, it causes the typed character to be written to current output port, or it invokes a procedure specified with `set-typing-special-procedure!` to be called with the pressed special key name as argument.

`set-typing-special-procedure!` *proc* [Procedure]
When in typing mode, *proc* will be called whenever a special (i.e. non-printable, e.g. return, escape of F1) key is pressed or repeated. *proc* will receive one argument: the name of the special key that was pressed (as a string).

`grab-input!` *state* [Procedure]
If *state* is given and not `#f`, the procedure causes the SLAYER window to grab all the keyboard and mouse input. This can be deactivated and brought back to normal by passing `#f` as *state*. When *state* is not given, the procedure only returns the current state.

`register-userevent!` *proc* [Procedure]
Register a new user event. The *proc* procedure will be called with zero, one or two arguments, depending on the way the corresponding `generate-userevent!` procedure is called. `register-userevent!` returns an identifier of user event (which happens to be an integer). Up to 255 user events can be registered. Note also, that the procedure `add-timer!` can be used to generate an user event periodically.

`generate-userevent!` *identifier* [*data1*] [*data2*] [Procedure]
Generate user event identified with `identifier`, obtained with a previous call to `register-userevent`. The *data1* and *data2* arguments are passed to event handler if provided.

`add-timer!` *ms proc* [Procedure]
Generate a new user-event to be called periodically at interval of *ms* miliseconds, and install *proc* as a handler for the newly-created event. Returns the id of this new event.

`remove-timer!` *id* [Procedure]
Remove a timer with the id *id* obtained by prior call to `add-timer!`. Returns `#t` on success and `#f` on failute.

2.1.1 Key names

The names of the keys are usually rather intuitive: 'a names the key A, `"1"` names the key 1, and so on. Note that, although usually the names can be given as symbols, in some cases the string representation is required, like in the case of digits, punctuation marks or brackets. Note also, that not all these characters will be accessible through the event subsystem in direct mode (it should depend on the keyboard layout): for instance, if exclamation mark is obtained by pressing shift and '1' (in typing mode), only the events related with pressing shift and '1' will be generated (in direct mode).

SLAYER NAME	*ASCII VALUE*	*COMMON NAME (SDL)*
"mouse-left"		left mouse button
"mouse-right"		right mouse button
"mouse-middle"		middle mouse button
"mwheelup"		mouse wheel up
"mwheeldown"		mouse wheel down
"backspace"	'\b'	backspace
"tab"	'\t'	tab
"clear"		clear
"return"	'\r'	return
"pause"		pause
"esc"	'^['	escape
"space"	' '	space
"!"	'!'	exclaim
"\""	'"'	quotedbl
"#"	'#'	hash
"$"	'$'	dollar
"&"	'&'	ampersand
"'"	"'"	quote
"("	'('	left parenthesis
")"	')'	right parenthesis
"*"	'*'	asterisk
"+"	'+'	plus sign
"-"	'-'	minus sign
","	','	comma
"."	'.'	period
"/"	'/'	forward slash
"0"	'0'	0
"1"	'1'	1
"2"	'2'	2
"3"	'3'	3
"4"	'4'	4
"5"	'5'	5
"6"	'6'	6
"7"	'7'	7
"8"	'8'	8
"9"	'9'	9
":"	':'	colon
";"	';'	semicolon
"<"	'<'	less-than sign
"="	'='	equals sign
">"	'>'	greater-than sign
"?"	'?'	question mark
"@"	'@'	at
"["	'['	left bracket
"\\"	'\'	slash
"]"	']'	right bracket
"^"	'^'	caret
"_"	'_'	underscore
"`"	'`'	grave

"a"	'a'	a
"b"	'b'	b
"c"	'c'	c
"d"	'd'	d
"e"	'e'	e
"f"	'f'	f
"g"	'g'	g
"h"	'h'	h
"i"	'i'	i
"j"	'j'	j
"k"	'k'	k
"l"	'l'	l
"m"	'm'	m
"n"	'n'	n
"o"	'o'	o
"p"	'p'	p
"q"	'q'	q
"r"	'r'	r
"s"	's'	s
"t"	't'	t
"u"	'u'	u
"v"	'v'	v
"w"	'w'	w
"x"	'x'	x
"y"	'y'	y
"z"	'z'	z
"delete"	'^?'	delete
"num0"		keypad 0
"num1"		keypad 1
"num2"		keypad 2
"num3"		keypad 3
"num4"		keypad 4
"num5"		keypad 5
"num6"		keypad 6
"num7"		keypad 7
"num8"		keypad 8
"num9"		keypad 9
"num."	'.'	keypad period
"num/"	'/'	keypad divide
"num*"	'*'	keypad multiply
"num-"	'-'	keypad minus
"num+"	'+'	keypad plus
"numret"	'\r'	keypad enter
"num="	'='	keypad equals
"up"		up arrow
"down"		down arrow
"left"		left arrow
"right"		right arrow
"ins"		insert
"home"		home
"end"		end
"pgup"		page up

"pgdown"	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
"caps"	capslock
"scroll"	scrollock
"lshift"	left shift
"rshift"	right shift
"lctrl"	left ctrl
"rctrl"	right ctrl
"lalt"	left alt
"ralt"	right alt
"lmeta"	left meta
"rmeta"	right meta
"lsuper"	left windows key
"rsuper"	right windows key
"mode"	mode shift
"help"	help
"print"	print screen
"sysrq"	SysRq
"break"	break
"menu"	menu
"power"	power
"euro"	euro

2.1.2 Implementing persistent input

The `keydn` and `keyup` procedures register event handlers that are called whenever a certain key is pressed or released. It can sometimes be desired, however, to call a certain procedure "persistently", i.e. as long as a given key is pressed.

SLAYER itself doesn't provide such procedures, because they frequently depend on a specific application. However, this document provides a general solution that can be taylored to one's needs.

```
(use-modules (slayer))

;; we need to keep track on which keys are pressed, in the *modes* hash table.
;; its keys are the names of pressed keys, and values -- the procedures that
;; are meant to be called "persistently"
```

```
(define *modes* (make-hash-table))

;; we also need to be able to define a procedure to be called as long as the
;; key is pressed

(define (key name proc)
  (keydn name (lambda()(hash-set! *modes* name proc)))
  (keyup name (lambda()(hash-remove! *modes* name))))

;; lastly, we shall invoke procedures corresponding to the pressed keys with
;; a certain period, say, 30 miliseconds:

(add-timer! 30 (hash-for-each (lambda(key thunk)(thunk)) *modes*))

;; now we can bind keys to persistent procedures:
(key 'return
     (lambda ()
        (display (string-append "this message will be printed"
                                " for as long as 'return is pressed\n"))))
```

2.2 Loading and processing fonts, images and sounds

When creating a multimedia application, it is inevitable to work with external data, such as sounds and images. It can also be strongly desired to be able to render text to images and display it on the screen.

The following sections describe how to perform those activities in SLAYER.

2.2.1 Loading and displaying images

In order to be able to use the procedures documented here, one shall `(use-modules (slayer image))`. The module provides a symbol `slayer-image` to use with `cond-expand` (srfi-0).

`load-image` *path* [Procedure]
Loads an image from an external file, indicated by *path*. The supported file formats depend on the way the SDL_image library was compiled. The procedure returns an object that represents the image internally, and can be used with `draw-image!` and other procedures. FIXME: if something goes wrong (i.e. the file doesn't exist), the application will probably crash.

`draw-image!` *image x y* [Procedure]
Display *image* on the screen. Its top left corner will be located at (x, y).

`rectangle` *w h color* [Procedure]
Generates and returns solid rectangle whose width is w, height h and color is *color*. *color* is a 32-bit integer value, which can be written as `#xAARRGGBB`, where `AA` means alpha channel, `RR` is red component, `GG` – green component, and `BB` – blue component.

`image-size` *image* [Procedure]
Returns `(width height)` of the *image*.

`image->array` *image* [Procedure]
Converts *image* to two-dimensional uniform array of integers.

`array->image` *array* [Procedure]
Converts two-dimensional uniform *array* of integers to image.

`decompose-color-to-rgba` *color* [Procedure]
Return a list of `(red green blue alpha)` components of 32-bit integer *color*. The returned values range between 0 and 255, the higher being more intensive.

`compose-color-from-rgba` *red green blue alpha* [Procedure]
Return a 32-bit integer value (as elements of the array returned by `image->array` are) that represents color with *red*, *green*, *blue* and *alpha* components.

2.2.2 Rendering and displaying text

In order to be able to use the procedures documented here, one shall `(use-modules (slayer font))`.

`load-font` *path pt-size* [Procedure]
Load TrueType font from a file indicated by *path*. The size of the font will be *pt-size*. Returns the object representing font, which can later be used for rendering text using `render-text` procedure.

`render-text` *text font* [*color*] [*bgcolor*] [Procedure]
Returns a new image containing *text* rendered using *font* with *color* (which defaults to white). If *bgcolor* is given, it will become the background color of the image; otherwise, the background is transparent. The obtained image can be displayed on the screen using `draw-image!`.

`font-line-skip` *font* [Procedure]
Returns the line skip (integer) defined by the *font*.

2.2.3 Loading and playing sounds

In order to be able to use the procedures documented here, one needs to `(use-modules (slayer audio))`. The module provides a symbol `slayer-audio` for `cond-expand` (srfi-0).

`load-sound` *path* [Procedure]
Loads a sound file indicated by *path*. It ought to be a wave file.

`play-sound!` *sound* [Procedure]
Plays a specified *sound*, which has previously been loaded using the `load-sound` procedure. Many sound files can be played at once.

`load-music` *path* [Procedure]
Loads a music file indicated by *path*. It can be mp3 or ogg file.

`play-music!` *music* [Procedure]
Plays a specified *music*, which has previously been loaded using `load-music` procedure. Only one music track can be played at once, and it can be paused using `pause-music!` procedure.

`pause-music!` [Procedure]
Pauses currently played music track (if any).

`resume-music!` [Procedure]
Resumes previously paused music track (if any).

2.3 Diving into 3D graphics

Apart from the elementary support for images, SLAYER also allows to use OpenGL display context and exports some of the OpenGL/GLU procedures.

In order to use it, SLAYER needs to operate in 3d mode. This can be achieved by running it with `-e3d` or `--extension 3d` command line argument, provided that SLAYER has been built with OpenGL support. It is also possible to compile SLAYER so that it runs in 3d mode by default.

To recognise whether or not the 3d mode is available, one can check for the presence of `slayer-3d-available` symbol within `cond-expand` form (srfi-0). Furthermore, if 3d mode is enabled, the `slayer-3d` is provided.

If you're not familiar with the OpenGL library, I recommend you to read at least the first three chapters of the OpenGL Programming Guide, also known as The Red Book[1].

Note, that programming in SLAYER differs in a few ways from the raw OpenGL API. Firstly, many OpenGL tutorials use "glBegin", "glEnd", "glVertex*" and similar procedures. SLAYER does not support them. Instead, it forces you to use its wrappers for "glVertexPointer", "glColorPointer", "glDrawElements" and so on, which makes it a little bit more difficult for beginners, but results in a more concise code. (There are also higher level interface functions available in (extra 3d) and (widgets 3d-view) modules)

Secondly, because of Guile's more informative data types, it is unnecessary to maintain so many variants of the same procedure (like "glColor3f", "glColor4f", "glColor3u", "glColor4i") – SLAYER chooses the appropreate variant depending on the type of the argument.

Thirdly, OpenGL doesn't allow to choose to perform operations on the projection matrix stack: only modelview matrix is available to the procedures like "push-matrix!", "pop-matrix!" or "multiply-matrix!"; the "glMatrixMode" procedure is unavailable to the programmer, because I didn't find that useful at all.

`multiply-matrix!` *M* [Procedure]
Multiply current modelview matrix by 4x4 uniform array *M* containing floats or doubles.

`push-matrix!` [Procedure]
Push current modelview matrix on the matrix stack.

`pop-matrix!` [Procedure]
Pop modelview matrix from the stack.

`load-identity!` [Procedure]
Set current modelview matrix to identity.

`translate-view!` *vector* [Procedure]
Add a translation *vector* of 3 numbers to current modelview matrix.

`rotate-view!` *quaternion* [Procedure]
Rotate current matrix by *quaternion*. Quaternions are represented by pairs, whose first element is the real scalar, and second – the imaginary vector, but this representation may change in the future[2].

`set-viewport!` *x y w h* [Procedure]
Set left upper corner of viewport to (*x*, *y*) and its dimensions to (*w*, *h*). Note that this differs from OpenGL's viewport, because the origin is the upper left corner, and not in the lower left.

[1] http://www.glprogramming.com/red/

[2] To find out more about how quaternions can be used to represent rotations in the 3d space, see e.g. http://www.genesis3d.com/~kdtop/Quaternions-UsingToRepresentRotation.htm

`current-viewport` [Procedure]
Return a list `(x y w h)` describing the current viewport. Again, origin is located in the upper left corner, contrary to OpenGL's convention.

`set-perspective-projection!` *fovy* [*aspect*] [*near*] [*far*] [Procedure]
Set the projection matrix to calculate perspective projection, where horizontal field of view is specified with *fovy*. If *aspect* isn't given, it is calculated according to current viewport, to preserve natural aspect ratio; the *near* clipping plane defaults to 0.1, and *far* – to 1000.0.

`set-orthographic-projection!` *left right bottom top* [*near*] [*far*] [Procedure]
Sets the projection matrix to calculate ortographic projection. If *near* and *far* not given, they default to `-1.0` and `1.0`, respectively.

`set-vertex-array!` *array* [Procedure]
Sets current vertex pointer to the two dimensional uniform *array* of vertices to use it with the `draw-faces!` procedure. The first dimension of the *array* specifies the number of vertices, and the second – the number of coordinates (from 2 to 4), so for instance if *array* is `#2f32((0 0 0)(1 1 1))`, it will be interpreted as an array containing two three-dimensional vertices: `(0, 0, 0)` and `(1, 1, 1)`. The uniform array can be of any real type, so it can be either f32, f64, s32, u32, s16, u16, s8 or u8. This procedure enables GL_VERTEX_ARRAY client state.

`set-color-array!` *array* [Procedure]
Sets current color pointer to the two-dimensional uniform *array* of color parameters, that can either have 3 or 4 parameters (the fourth value will be interpreted as alpha channel). The remaining notes from `set-vertex-array!` apply here as well (mutatis mutandis).

`set-normal-array!` *array* [Procedure]
Sets current normal pointer to the two-dimensional uniform *array* of normal vectors. The vectors must be three-dimensional. The remaining notes from `set-vertex-array!` apply here as well (mutatis mutandis).

`set-texture-coord-array!` *array* [Procedure]
I implemented this procedure, because it was similar to the ones above, but I really don't know what it does. It has never been used nor tested, so if you would like to use it, I wish you all best!

`draw-faces!` *type indices* [Procedure]
Draws vertices from the array set using `set-vertices-array!`, interpreted depending on the value of the *type* parameter, in the order specified in the uniform array of integer *indices* (which can be either u8, u16 or u32, and the dimension of which is irrelevant). The *type* is a symbol that can have one of the following values: `points`, `lines`, `line-strip`, `line-loop`, `triangles`, `triangle-strip`, `triangle-fan`, `quads`, `quad-strip`, `polygon` (consult OpenGL reference for details).

`forget-array!` *type* [Procedure]
Forget a pointer to the array specified in *type*, so that `draw-faces!` won't take it into consideration. Possible values of *type* are the following symbols: `vertex-array`, `color-array`, `texture-coord-array`, `normal-array`.

`set-color!` *color* [Procedure]
Sets the current display color to *color*, which can either be a uniform vector containing three or four elements, or an integer, interpreted as in the `rectangle` function from `(slayer image)`.

`make-light-` [Procedure]
Allocate new light. Once the light is allocated, various properties can be set using `set-light-property!` procedure. When the light is no longer needed, it is desirable to remove it using `remove-light!`.

`remove-light!` *light* [Procedure]
Remove *light* allocated by `make-light-` procedure (or its derivatives). It is desirable to remove lights in the opposite order to their allocation order.

`set-light-property!` *light property value* [Procedure]
Set the *light*'s *property* to *value*. The supported properties (symbols) are similar to parameters of `glLightf*`, but the `'position` and `'direction` parameters differ slightly.

property	*value type*	*description*
'ambient	`#f32(r g b a)`	ambient intensity
'diffuse	`#f32(r g b a)`	diffuse intensity
'specular	`#f32(r g b a)`	specular intensity
'position	`#f32(x y z)` or `#f`	light position; if `#f`, then the light is directional
'direction	`#f32(x y z)`	direction of light
'cutoff	`real`	spotlight cutoff angle in degrees (for positional lights). The value of 180.0 causes spherical, non-directiona light.
'exponent	`real`	
'constant-attenuation	`real`	
'linear-attenuation	`real`	
'quadratic-attenuation	`real`	

I realize that the above list of procedures might be insufficient for certain purposes. If you notice that an essential procedure is lacking here, don't hesitate to let me know about it, and we can try to work out a solution.

3 Widgets

When developing a graphics user interface based application, it is convinient to assemble it from simple and elastic components, such as buttons, text areas, draggable icons, canvases, menus, windows and so on. The common name for these components is "widget". SLAYER is shipped with its own framework for creating and using widgets.

In order to use the widget framework, one shall `(use-modules (widgets base))`. It creates and exports the main widget under the symbol `*stage*`, and sets the appropriate display and resize procedures, as well as handlers for mouse1 and mouse2 press and release events and mousemove event.

It also contains a definition of a `<widget>` class, which is the base class for all widgets. Here's a simplified definition of `<widget>` class (in fact, all event handler slots are initialized to noop, and their init keywords correspond strictly to their names):

```
(define-class <widget> ()
  (parent #:init-value #f #:init-keyword #:parent)
  (children #:init-value '() #:init-keyword #:children)

  ;; event handlers (initialized to noop)
  left-mouse-down
  left-mouse-up
  right-mouse-down
  right-mouse-up
  mouse-over
  mouse-out
  drag
  update!
  activate
  deactivate
  resize

  (x #:init-value 0 #:init-keyword #:x)
  (y #:init-value 0 #:init-keyword #:y)
  (w #:init-value 0 #:init-keyword #:w)
  (h #:init-value 0 #:init-keyword #:h))
```

As you can see, the widget structure is hierarchical: all widgets (except `*stage*`) have their parent, and some can have their children. The biggest part of the definition consists of some event handlers, which are called in certain situations (like clicking, dragging and so on). Furthermore, all widgets have rectangular shape, and the rectangle should be big enough to fit all of the widget's children (otherwise they can become unreachable).

In order to use the widget framework, it is enough to load the desired widget modules and add their instances to `*stage*` using the `(add-child! *stage* /instance-of-a-widget/)`. The simplest program that uses widgets – a draggable rectangle – could look like this:

```
(use-modules (slayer) (slayer image)
     (widgets base) (widgets bitmap)
     (oop goops))
(keydn 'esc quit)

(define-method (dragger (w <widget>))
  (lambda (x y xrel yrel)
    (slot-set! w 'x (+ (slot-ref w 'x) xrel))
```

```
      (slot-set! w 'y (+ (slot-ref w 'y) yrel))))

(let* ((rect (rectangle #;w 50 #;h 50 #;color #xcc33dd))
       (img (make-image rect #;x 50 #;y 50)))
  (slot-set! img 'drag (dragger img))
  (add-child! *stage* img))
```

There are a few widgets already bundled with slayer, among which there is <text-area> widget (which still requires some work) and <3d-view> widget.

Check the slayer.scm demo to see how to use them.

Note also, that although the direct layer described in the previous chapter is rather stable and will only be extended rather than modified, the widget layer is more likely to change.

4 Plug-ins (SCUM)

Currently SLAYER is shipped with one additional module – a wrapper for the prominent Open Dynamics Engine, which is required for the module to build. The module is located outside the build tree, so it doesn't get installed by default, but requires additional effort. (This should change in the future)

In order to build the module, one needs to enter the src/scum directory from the installation package, and type: `$ make`

If everything goes well, a file named "physics.so" will be created. In order to run, it should be moved to a path specified in the LTDL_LIBRARY_PATH environment variable, and the accompanyign file, "physics.scm" should be placed in a "scum" directory within the range of GUILE_LOAD_PATH environment variable.