SDL::Manual

Writing Games in Perl

Kartik Thakore

With contributions by the community

Latex based on the Perl6 book: HTTPS://GitHub.Com/perl6/book

Contents

1

Preface

Simple DirectMedia Layer (or *libsdl*) is a cross-platform C library that provides access to several input and output devices. Its most popular usage is to provide access to the video framebuffer and input devices for games. SDL also has several extension libraries to provide features such as text display, sound mixing, image handling, and graphics effects.

SDL Perl binds several of these libraries together in the `SDL::*` namespace. Moreover, SDL Perl provides several high-level libraries in the `SDLx::*` namespace that encapsulate valuable game-writing abstractions.

1.1 SDL and SDLx

The main purpose of the `SDLx::*` layer is to smooth out the drudgery of using the `SDL::*` layer directly.

> Don't worry about understanding the details of this code right now. Compare the complexity and size of the code listings.

Using the `SDL::*` layer to draw a blue rectangle looks something like:

```
use SDL;
use SDL::Video;
use SDL::Surface;
use SDL::Rect;

# the size of the window box or the screen resolution if fullscreen
my $screen_width   = 800;
my $screen_height  = 600;

SDL::init(SDL_INIT_VIDEO);

# setting video mode
my $screen_surface = SDL::Video::set_video_mode($screen_width,
                                                $screen_height,
                                                32,
                                                SDL_ANYFORMAT);

# drawing a rectangle with the blue color
my $mapped_color   = SDL::Video::map_RGB($screen_surface->format(),
                                         0, 0, 255);
SDL::Video::fill_rect($screen_surface,
                      SDL::Rect->new($screen_width / 4, $screen_height / 4,
                                     $screen_width / 2, $screen_height / 2),
                      $mapped_color);
```

```
25
26 # update an area on the screen so it's visible
27 SDL::Video::update_rect($screen_surface, 0, 0,
28                         $screen_width, $screen_height);
29
30 # just to have time to see it
31 sleep(5);
```

... while drawing a blue rectangle in the `SDLx::*` layer is as simple as:

```
1  use strict;
2  use warnings;
3
4  use SDL;
5  use SDLx::App;
6
7  my $app = SDLx::App->new( width=> 800, height => 600 );
8
9  $app->draw_rect([ $app->width / 4, $app->height / 4,
10                   $app->width / 2, $app->height / 2, ],
11                   [ 0, 0, 255, 255] );
12
13 $app->update();
14
15 sleep(5);
```

The `SDLx::*` modules also provide and manage higher-level concerns for users, such as layers and game loops.

1.2 About the Book

This book has a two-fold purpose: first, to introduce game development to Perl programmers, and second, to introduce Modern Perl concepts through game development. While the examples assume some experience with Perl, no experience with SDL in Perl or as `libsdl` itself is necessary.

The book presents a progression from simple to intermediate examples and provides suggestions for more advanced endeavors. The chapters of this book increase progressively in complexity, but each chapter has a singular goal (such as chapter five's *Making Pong*) which stands alone as an individual tutorial. Sources and data files are all available from `http://sdl.perl.org/`.

1.3 Installing SDL Perl

We assume the presence of a recent version of the Perl language (at least Perl 5.10) and supporting packages. We also assume that you can install packages from the CPAN, including SDL Perl itself.

1.3.1 Windows

`Alien::SDL` will install binaries for 32bit and 64bit so there is no need to compile anything.

1.3.2 Mac OS X

Fink has packages for SDL Perl available. However, they do not support Pango, a library which provides internalization support for text handling.

Installing `Alien::SDL` from the CPAN will compile SDL and its dependencies, provided you have installed severan necessary dependencies. We recommend that you install `libfreetype6`, `libX11`, `libvorbis`, `libogg`, `libpng`, and their headers.

1.3.3 GNU/Linux

Most current GNU/Linux distributions include all the parts needed for this tutorial in the default install and in their package management system. It is also always possible to install on GNU/Linux using the available open source code from the proper repositories. The `Alien::SDL` perl module automates much of downloading, compiling, and installing the needed libraries.

You can probably use your distribution's packages. On Ubuntu and Debian try:

```
$ sudo apt-get install libsdl-net1.2-dev libsdl-mixer1.2-dev \
libsdl1.2-dev libsdl-image1.2-dev libsdl-ttf2.0-dev \
libsdl-gfx1.2-dev libsdl-pango-dev
```

To compile from scratch, you must install a compiler, system header packages, and some libraries are required.

```
$ sudo apt-get install build-essential xorg-dev libx11-dev libxv-dev \
libpango1.0-dev libfreetype6-dev libvorbis-dev libpng12-dev \
libogg-dev
```

1.3.4 CPAN install

Before installing SDL Perl, ensure that you have the most recent versions of the modules necessary to build SDL:

```
$ sudo cpan CPAN
$ sudo cpan YAML Module::Build
```

After these two steps CPAN will be able to install SDL:

```
$ sudo cpan SDL
```

For most platforms a CPAN install will suffice. Supported and tested platforms are listed at `http://pass.cpantesters.org/distro/S/SDL.html`.

1.4 Contact

Hopefully this book answers most of your questions. For additional assistance, contact the project via:

- *the web*, by visiting the SDL Perl homepage at `http://sdl.perl.org/`.
- *IRC*, in the `#sdl` channel on `irc.perl.org`. This is a very active and helpful resource.
- *email*, through the `sdl-devel@perl.org` mailing list.

1.5 Examples

The code examples in this book are available from `https://github.com/PerlGameDev/SDL_Manual/tree/master/code_listings`.

1.6 Acknowledgements

Thanks to contributors and reviewers from the `#sdl` channel, including:

Alias

bobross

Blaizer

cfedde

chromatic

FROGGS

garu

jamesw

perlpilot

PerlJam

Pip

waxhead

and many more

(Apologies if I have missed you; let me know and I will add you.)

2

The Screen

SDL's primary purpose is to display graphics. It does so by providing an abstraction called a *screen*, which represents a *video device*. This video device is an interface provided by your operating system, such as X11 or DirectX. Before you can display anything, you must create a screen. The `SDLx::App` class does so for you:

```
1  use strict;
2  use warnings;
3  use SDL;
4  use SDLx::App;
5
6  my $app = SDLx::App->new();
7
8  sleep( 2 );
```

This example causes an empty window to appear on the desktop. Most systems will fill that window with the color black. Other systems might display a transparent window. SDL's

default behavior is to fill the screen with black. To enforce this behavior on all systems, you must `update()` the app to draw to the window:

```
$app->update();
```

2.1 `SDLx::App` Options

`SDLx::App` allows you to specify several options for the screen and your application. First are the physical dimensions of the screen itself. To make the screen of the `SDLx::App` window a 400×400 pixel square, change the initialization line to:

```
my $app = SDLx::App->new( width => 400, height => 400 );
```

Another important option is the window's title. Some systems display the path to the running program. Others leave the title blank. You can change the displayed title with another argument to the `SDLx::App` constructor:

```
my $app = SDLx::App->new( width  => 400,
                          height => 400,
                          title  => 'Pong - A clone' );
```

At this point your screen will be:

2.1.1 Shortcuts

Abbreviations for these parameters are available. Instead of `width`, `height`, and `title`, you may use `w`, `h`, and `t` respectively. The previous example could also be written:

```
my $app = SDLx::App->new( w => 400,
                          h => 400,
                          t => 'Pong - A clone' );
```

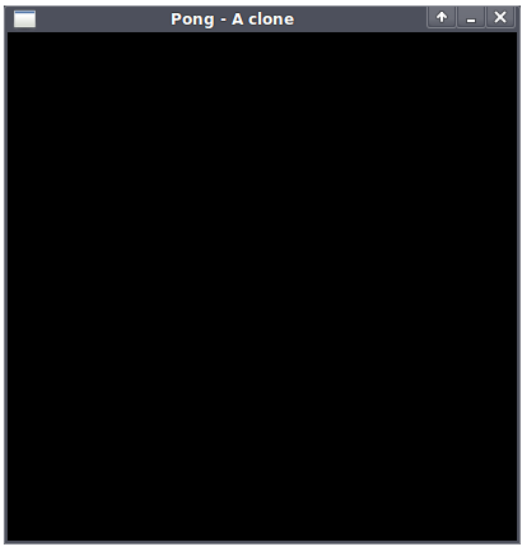

Figure 2.1: Your first SDL screen!

3

Drawing

SDL provides several ways to draw graphical elements on the screen in three general categories: primitives, images, and text. All drawing occurs on a surface, represented by the `SDLx::Surface` class. Even the `SDLx::App` is an `SDLx::Surface`. Though this means it's possible to draw directly to the app's surface, there are several advantages to drawing on multiple surfaces.

3.1 Coordinates

SDL's surface coordinate system has its origin (where both the x and y coordinates have the value of zero) in the upper left corner. As the value of x increases, the position moves to the right of the origin. As the value of y increases, the position moves downward from the origin. The API always lists coordinates in x, y order.

The SDL library documentation has an extended discussion on coordinates: `http://sdltutorials.com/sdl-coordinates-and-blitting`.

3.2 Drawing with SDL

You can produce original pictures knowing little more than how to draw to a surface with SDL:

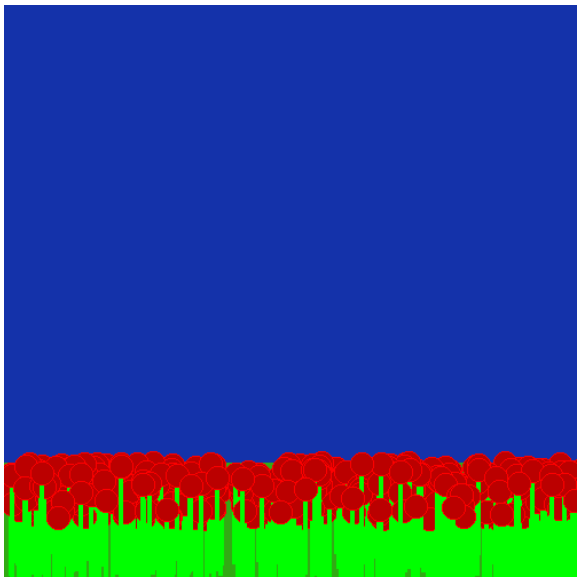

Figure 3.1: A field of flowers

3.2.1 Surface Drawing Methods

As mentioned earlier, all drawing in SDL requires a surface. The `SDLx::Surface` object provides access to methods in the form of:

```
$surface->draw_{something}( .... );
```

Parameters to these methods are generally coordinates and colors, provided as array references.

Rectangular Parameters

Some parameters are sets of coordinate positions and dimensions. For example, parameters to describe a rectangle of `40x40` pixels placed at `(20, 20)` pixel units on the screen make a four-element array reference of x, y, width, height:

```
my $rect = [20, 20, 40, 40];
```

Color

SDL color parameters require four-element array references. The first three numbers define the Red, Green, and Blue intensity of the color. The final number defines the transparency of the color.

```
my $color = [255, 255, 255, 255];
```

The magnitude of each color value determines how much of that color component will be mixed into the resulting color. A 0 value specifies that none of the color channel should be used while 255 specifies a maximum intensity for a particular channel. The first value corresponds with the Red channel, so a higher number there means more red will be mixed into the resulting color. It is a common practice to achieve a grayscale of varying intensity by

specifying the same value for each of the Red, Green, and Blue color channels. The fourth and final value designates the transparency (or Alpha channel) where a 0 value makes the resulting color fully transparent and 255 makes it entirely opaque. A transparency value somewhere in between will allow underlying (pixel data of surfaces below the current one) colors to be blended with the specified RGB values into the final color output.

You may also represent a color as hexadecimal values, where the values of the numbers range from 0-255 for 32 bit depth in RGBA format:

```
my $color = 0xFFFFFFFF;
my $white = 0xFFFFFFFF;
my $black = 0x000000FF;
my $red   = 0xFF0000FF;
my $green = 0x00FF00FF;
my $blue  = 0x0000FFFF;
```

... or as four-byte hexadecimal values, where each two-digit byte encodes the same RGBA values:

```
my $goldenrod = 0xDAA520FF;
```

NOTE: Depth of Surface

The color depth of the surface–how many bits are available to describe colors–is a property of the relevant `SDLx::Surface` or `SDLx::App`. Set it in its constructor:

```
my $app = SDLx::App->new( depth => 32 );
```

The default bit depth is 32, such that each color component has 256 possible values. Other options are 24, 16, and 8.

3.2.2 Pixels

All `SDLx::Surfaces` are collections of pixels. You can read from and write to these pixels by treating the surface as an array reference:

```
$app->[$x][$y] = $color;
```

... where `$color` is an unsigned integer value using the hexadecimal format (`0xRRGGBBAA`) *or* an anonymous array of the form `[$red, $green, $blue, $alpha]`.

3.2.3 Primitives

Drawing primitives are simple shapes that SDL supports natively.

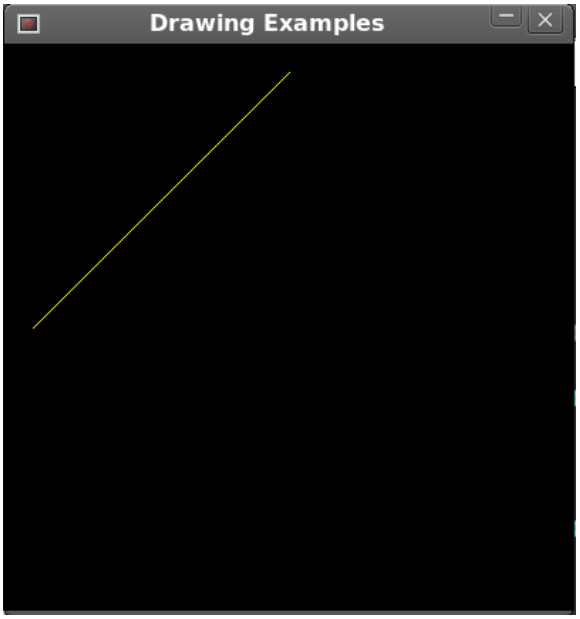

Figure 3.2: Drawing a line

Lines

A line is a series of contiguous pixels between two points. The `draw_line` method causes SDL to draw a line to a surface:

```
$app->draw_line( [200, 20], [20, 200], [255, 255, 0, 255] );
```

This will draw a yellow line from positions `(200, 20)` to `(20, 200)`.

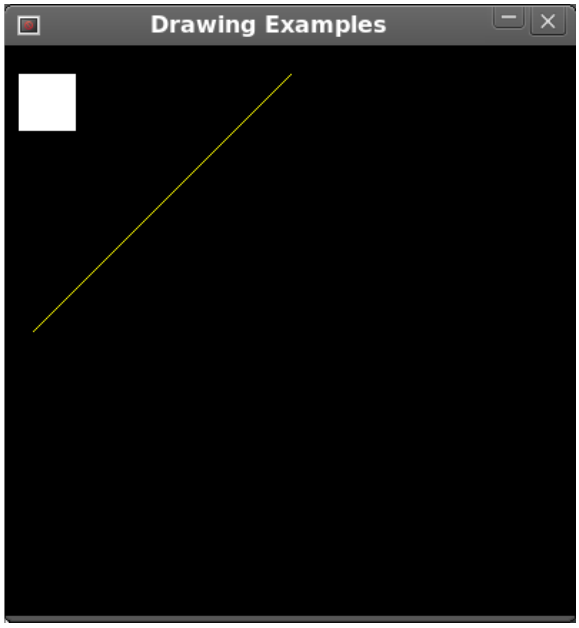

Figure 3.3: Drawing a Rectangle

Rectangles

A rectangle is a four-sided, filled polygon. Rectangles are a common building block for games. In SDL, rectangles are the most cost effective of the primitives to draw. The `draw_rect` method draws a rectangle on a surface:

```
$app->draw_rect( [10, 20, 40, 40 ], [255, 255, 255,255] );
```

This draws a white square of size `40x40` onto the screen at the position `(10,20)`.

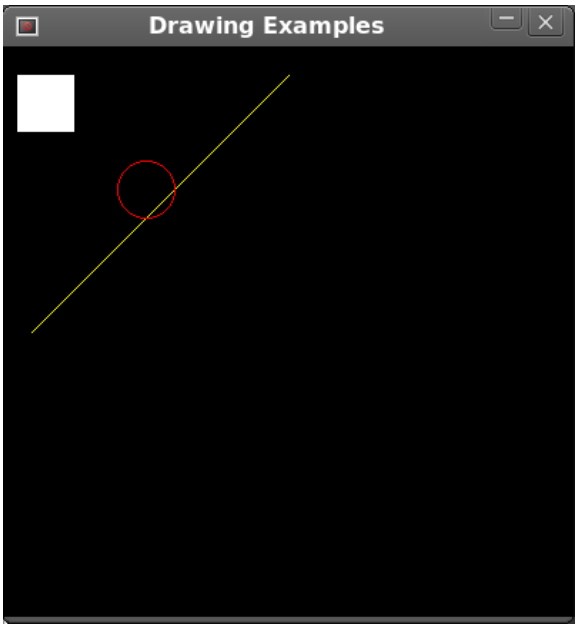

Figure 3.4: Drawing a Circle

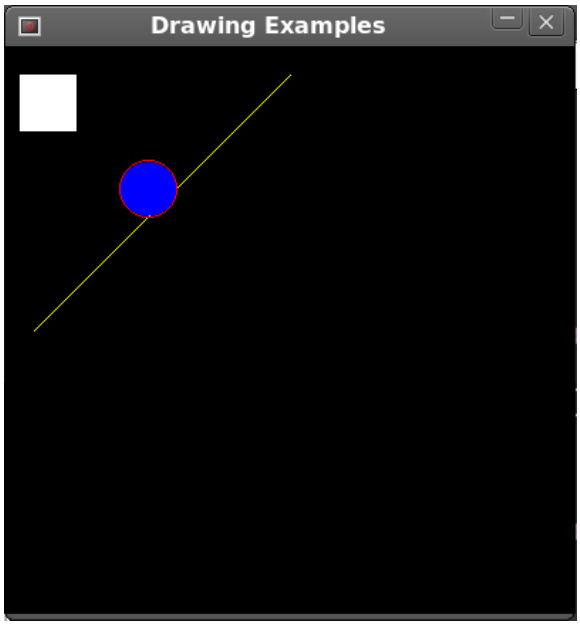

Figure 3.5: Drawing a filled Circle

Circles

A circle is a primitive a fixed radius around a given point. Circles may be filled or unfilled. The `draw_circle` and `draw_circle_filled` methods draw these to a surface:

```
$app->draw_circle(        [100, 100], 20, [255, 0,   0, 255] );
$app->draw_circle_filled( [100, 100], 19, [0,   0, 255, 255] );
```

These draw an unfilled red circle and a filled blue circle.

SDL provides more complex primitives in `SDL::GFX::Primitives`.

3.2.4 Drawing with Primitives

It’s easy to combine several primitives to draw an interesting images.

```
use strict;
use warnings;
use SDL;
use SDLx::App;

my $app = SDLx::App->new(
    w     => 500,
    h     => 500,
    d     => 32,
    title => 'Pretty Flowers'
);

# Add the blue skies
$app->draw_rect( [ 0,   0, 500, 500 ], [ 20,  50, 170, 255 ] );

# Draw a green field
$app->draw_rect( [ 0, 400, 500, 100 ], [ 50, 170,  20, 100 ] );

# Make a surface for the flower
my $flower = SDLx::Surface->new( width => 50, height => 100 );

# With a black background
$flower->draw_rect( [  0,  0, 50, 100 ], [ 0,   0, 0,   0 ] );

# Draw a pretty green stem
$flower->draw_rect( [ 23, 30,  4, 100 ], [ 0, 255, 0, 255 ] );

# And a simple flower bud
$flower->draw_circle_filled( [ 25, 25 ], 10, [ 150, 0, 0, 255 ] );
$flower->draw_circle(        [ 25, 25 ], 10, [ 255, 0, 0, 255 ] );

# Draw flower on $app
$flower->blit( $app, [ 0, 0, 50, 100 ] );
```

```
34
35  $app->update();
36
37  sleep(1);
```

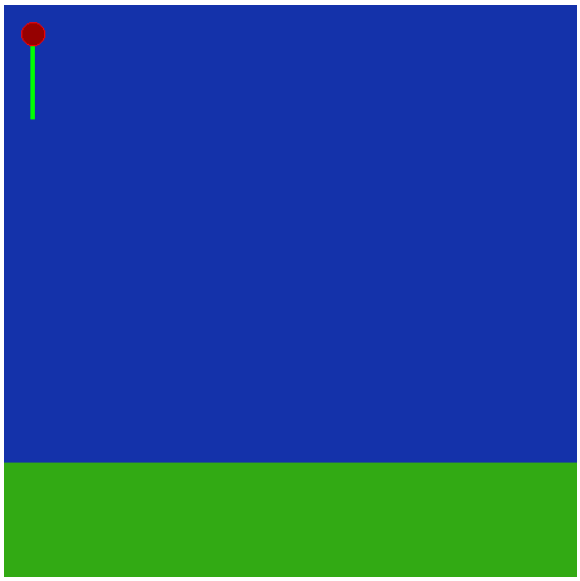

Figure 3.6: Looks so lonely there all alone

3.3 Drawing on Multiple Surfaces

The examples so far have drawn on only a single surface, the display. SDL makes it possible to write on multiple surfaces. These other surfaces exist only in memory until you draw them to the display.

3.3.1 Creating Surfaces

There are several ways to create an `SDLx::Surface` for use. The most common is to create one manually with a constructor call:

```
$surface = SDLx::Surface->new( width => $width, height => $height );
```

`SDL::Image` and `SDL::Video` can load images as surfaces too. `SDL::Image` provides support for all types of images, provided that the underlying `SDL_image` library supports the image type you want to load. For example, `SDL_image` must support PNG images to use:

```
$surface = SDL::Image::load( 'picture.png' );
```

In the event that the desired `SDL_image` library is unavailable, you can fallback to the built-in support for the `.bmp` format.

```
$surface = SDL::Video::load_BMP( 'picture.bmp' );
```

The `SDLx::Sprite` module provides another option to manipulate surfaces.

3.4 Lots of Flowers but One Seed

The flower example used a method called `blit` to draw a surface to the display. This method copies data from one surface to another. It's a fundamental operation, but it's a low level operation. `SDLx::Sprite` provides higher level options. Besides making drawing simpler, `SDLx::Sprite` adds several other features useful for moving images. Here's a revised example using `SDLx::Sprite` for flowers:

```
1  use strict;
2  use warnings;
3  use SDL;
4  use SDLx::App;
```

```
5  use SDLx::Sprite;
6
7  my $app = SDLx::App->new(
8      w     => 500,
9      h     => 500,
10     d     => 32,
11     title => 'Pretty Flowers'
12 );
13
14 # Adding blue skies
15 $app->draw_rect( [ 0,   0, 500, 500 ], [ 20, 50, 170, 255 ] );
16
17 # Draw a green field
18 $app->draw_rect( [ 0, 400, 500, 100 ], [ 50, 170, 20, 100 ] );
19
20 my $flower = SDLx::Sprite->new( width => 50, height => 100 );
21
22 # Use ->surface() to access a sprite's SDLx::Surface
23
24 # Make the background black
25 $flower->surface->draw_rect( [  0,  0, 50, 100 ], [ 0,   0, 0,   0 ] );
26
27 # Now for a pretty green stem
28 $flower->surface->draw_rect( [ 23, 30,  4, 100 ], [ 0, 255, 0, 255 ] );
29
30 # Add the simple flower bud
31 $flower->surface->draw_circle_filled( [ 25, 25 ], 10, [ 150, 0, 0, 255 ] );
32 $flower->surface->draw_circle(        [ 25, 25 ], 10, [ 255, 0, 0, 255 ] );
33
34 $flower->draw_xy( $app, 0, 0 );
35
36 $app->update();
37
38 sleep(1);
```

Flowers usually don’t grow in the sky. Flowers make more sense on the ground. It’s easy to insert plenty of identical flowers from a single sprite. Replace the line:

```
    $flower->draw_xy( $app, 0, 0 );
```

... with:

```
1  for (0 .. 500) {
2      my $y =           425 - rand( 50);
3      $flower->draw_xy( $app, rand(500) - 20, $y );
4  }
```

... to make an entire field of flowers.

4

Handling Events

The cornerstone of an SDL application is event handling. The user presses a key or moves the mouse. The operating system switches the focus of the active window. The user selects the quit option from the menu or the operating system. These are all events. How do you handle them?

SDL provides an event queue which holds all events that occur until they are removed. Every time an event occurs, SDL places it into the queue. The `SDL::Event` object represents this queue in Perl, allowing you to add and remove events constantly:

```
1       use strict;
2       use warnings;
3       use SDL;
4       use SDL::Event;
5       use SDL::Events;
6       use SDLx::App;
7
```

```
8       my $app   = SDLx::App->new( w => 200, h => 200 );
9       my $event = SDL::Event->new();
10
11      my $quit  = 0;
12
13      while (!$quit) {
14      # Updates the queue to recent events
15      SDL::Events::pump_events();
16
17              # process all available events
18              while ( SDL::Events::poll_event($event) ) {
19
20                      # check by Event type
21                      do_key() if $event->type == SDL_KEYDOWN;
22              }
23      }
24
25      sub do_key { $quit = 1 }
```

Every event has an associated type which represents the category of the event. The previous example looks for a keypress event [1]. The SDL library defines several types of events, and SDL_perl makes them available as constants with names such as `SDL_KEYDOWN` and `SDL_QUIT`. See `perldoc SDL::Events` for a list of all event types.

Checking for every possible event type within that event loop can be tedious. The `SDLx::Controller` available from the `SDLx::App` offers the use of event callbacks with which to handle events. Processing events is a matter of setting up the appropriate callbacks and letting SDL do the heavy work.

SDL Events Types

Additional Event types that can be captured by SDL are:

Keyboard

[1] SDL separates the event of pressing a key from the event of releasing a key, which allows you to identify combinations of keypresses, such as Ctrl + P to print.

`SDL_KEYDOWN` `SDL_KEYUP` - Keyboard button pressed

Mouse

`SDL_MOUSEMOTION` - Mouse motion occured

`SDL_MOUSEBUTTONDOWN` `SDL_MOUSEBUTTONUP` - Mouse button pressed

Joystick

`SDL_JOYAXISMOTION` - Joystick axis motion

`SDL_JOYBALLMOTION` - Joystick trackball motion

`SDL_JOYHATMOTION` - Joystick hat position change

`SDL_JOYBUTTONDOWN` `SDL_JOYBUTTONUP` - Joystick button pressed

Window & System

`SDL_ACTIVEEVENT` - Application visibility

`SDL_VIDEORESIZE` - Window resized

`SDL_VIDEOEXPOSE` - Window exposed

`SDL_QUIT` - Quit requested

`SDL_USEREVENT` - A user-defined event type

`SDL_SYSWMEVENT` - Platform-dependent window manager event

For more information look at:

```
perldoc SDL::Event
```

4.1 Quitting with Grace

The example applications so far have not exited cleanly. Handling quit events is much better:

```
use strict;
use warnings;
use SDL;
use SDL::Event;
use SDLx::App;

my $app = SDLx::App->new(
    w     => 200,
    h     => 200,
    d     =>  32,
    title => "Quit Events"
);

$app->add_event_handler( \&quit_event );
$app->run();

sub quit_event
{
    # the callback receives the appropriate SDL::Event
        my $event = shift;

        # ... as well as the calling SDLx::Controller
        my $controller = shift;

    # stopping the controller will exit $app->run() for us
```

```
26          $controller->stop if $event->type == SDL_QUIT;
27  }
```

SDLx::App calls the event_handlers, from an internal SDLx::Controller. When this event handler receives a quit event, it calls SDLx::Controller::stop() which causes SDLx::App to exit gracefully.

4.1.1 Exit on Quit

Exiting on receiving the SDL_QUIT event is such a common operation that SDLx::App provides it as a constructor option:

```
1   use strict;
2   use warnings;
3   use SDL;
4   use SDLx::App;
5
6   my $app = SDLx::App->new(
7       w            => 200,
8       h            => 200,
9       d            =>  32,
10      title        => "Quit Events",
11      exit_on_quit => 1
12  );
13
14  $app->run();
```

4.2 Small Paint: Input Devices

SDL events also allow input handling. Consider a simple paint program. It will provide a small black window. Moving the mouse draws on this window. Pressing a number key chooses a paint color. Pressing `q` or `Q` exits the program. Pressing `c` or `C` clears the screen. Pressing `Ctrl-S` saves the image to a file named *painted.bmp*.

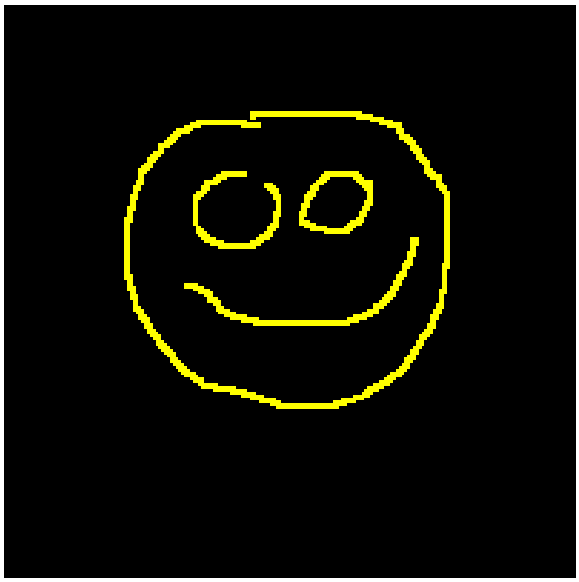

Figure 4.1: Simple Paint: Smile

4.2.1 Saving the image

Start by defining the saving function:

```
1  sub save_image {
2      if (SDL::Video::save_BMP( $app, 'painted.bmp' ) == 0
3      && -e 'painted.bmp')
```

```
    {
         warn 'Saved painted.bmp to ' . cwd();
    }
    else
    {
        warn 'Could not save painted.bmp: ' . SDL::get_errors();
    }
}
```

4.2.2 Keyboard

Keyboard handling requires some color data as well as a keypress callback:

```
my $brush_color = 0;

sub keyboard_event
{
    my $event = shift;

    if ( $event->type == SDL_KEYDOWN )
    {
        # convert the key_symbol (integer) to a keyname
        my $key_name = SDL::Events::get_key_name( $event->key_sym );

        # if $key_name is a digit, use it as a color
        $brush_color = $key_name if $key_name =~ /^\d$/;

        # get the keyboard modifier (see perldoc SDL::Events)
        my $mod_state = SDL::Events::get_mod_state();

        # we are using any CTRL so KMOD_CTRL is fine
        save_image() if $key_name =~ /^s$/ && ($mod_state & KMOD_CTRL);

        # clear the screen
        $app->draw_rect( [ 0, 0, $app->w, $app->h ], 0 )
            if $key_name =~ /^c$/;
```

```
24
25          # exit
26          $app->stop() if $key_name =~ /^q$/;
27      }
28
29      $app->update();
30  }
31
32  $app->add_event_handler(\&quit_event);
33  $app->add_event_handler(\&keyboard_event);
```

NOTE: When adding a callback to `SDLx::App` which uses variables declared outside of the function (`$brush_color` and `@colors` in this case), be sure to define them before declaring the subroutine. Normal Perl scoping and initialization rules apply.

4.2.3 Mouse

Handling mouse events is almost as straightforward as keyboard events: =begin programlisting

```
# track the drawing status
my $drawing = 0;

sub mouse_event {
    my $event = shift;

    # detect Mouse Button events and check if user is currently drawing
    if ($event->type == SDL_MOUSEBUTTONDOWN || $drawing)
    {
        # set drawing to 1
        $drawing = 1;

        # get the X and Y values of the mouse
```

```
        my $x = $event->button_x;
        my $y = $event->button_y;

        # draw a rectangle at the specified position
        $app->draw_rect( [ $x, $y, 2, 2 ], $colors[$brush_color] );

        $app->update();
    }

    # disable drawing when user releases mouse button
    $drawing = 0 if ($event->type == SDL_MOUSEBUTTONUP );
}

$app->add_event_handler( \&mouse_event );
```

This is all of the code necessary to make a simple drawing application.

Take note of two things. First, SDL_perl invokes the event handlers in the order of attachment. If the user presses `Q` and then moves the mouse, the application will quit before processing the mouse movement.

Second, the application makes no distinction between right, middle, or left mouse clicks. SDL provides this information. See the `button_button()` method in `SDL::Event`.

4.3 POD ERRORS

Hey! **The above document had some coding errors, which are explained below:**

Around line 317:

=end programlisting without matching =begin. (Stack: [empty])

5

The Game Loop

Just as an interactive SDL app builds around an event loop, a game builds around a game loop. The simplest game loop is something like:

```
while (!$quit)
{
   get_events();
   calculate_next_positions();
   render();
}
```

The names of the functions called in this loop hint at their purposes, but the subtleties of even this simple code are important. `get_events()` obviously processes events from the relevant input devices (keyboard, mouse, joystick). Processing events at the start of every game loop iteration helps to prevent lag.

`calculate_next_positions` updates the game state according to user input as well as any active animations (a player walking, an explosion, a cut scene). `render()` finally updates and displays the screen.

5.1 A Practical Game Loop

Consider a game with a moving laser bolt:

```
use strict;
use warnings;
use SDL;
use SDL::Event;
use SDL::Events;
use SDLx::App;

my $app = SDLx::App->new(
    width  => 200,
    height => 200,
    title  => 'Pew Pew'
);

my $quit = 0;

# start laser on the left
my $laser = 0;

sub get_events {
    my $event = SDL::Event->new();

    SDL::Events::pump_events;

    while( SDL::Events::poll_event($event) )
    {
        $quit = 1 if $event->type == SDL_QUIT
    }
```

```
}

sub calculate_next_positions {
        # move the laser
    $laser++;

        # if the laser goes off the screen, bring it back
    $laser = 0 if $laser > $app->w();
}

sub render {
    # draw the background first
    $app->draw_rect( [ 0, 0, $app->w, $app->h ], 0 );

    # draw the laser halfway up the screen
    $app->draw_rect( [ $laser, $app->h / 2, 10, 2 ], [ 255, 0, 0, 255 ]);

    $app->update();
}

while (!$quit)
{
     get_events();
     calculate_next_positions();
     render();
}
```

This game loop works very well for consoles and other devices where you know exactly how much CPU time the game will get for every loop iteration. That hardware stability is easy to predict: each animation and calculation will happen at the same time for each machine. Unfortunately, this is *not* true for modern operating systems and general purpose computing hardware. CPU speeds and workloads vary, so for this game to play consistently across multiple machines and myriad configurations, the game loop itself needs to regulate its updates.

5.1.1 Fixed FPS

One way to solve this problem is to regulate the number of frames per second the game will produce. A *frame* is a complete redraw of the screen representing the updated game state. If each iteration of the game loop draws one frame, the more frames per second, the faster the game is running. If the game loop limits the number of frames per second, the game will perform consistently on all machines fast enough to draw that many frames per second.

You can see this with the example program *game_fixed.pl*. When run with no arguments:

```
$ perl game_fixed.pl
```

.... the FPS rate will be erratic. The laser seems to change its speed randomly. When run with a single argument, the game sets an upper bound on the number of frames per second:

```
$ perl game_fixed.pl 1
```

This will prevent the laser from going faster than 60 frames per second. When run with a second argument, the game will set a lower bound of frames per second:

```
$ perl game_fixed.pl 1 1
```

At this point the FPS should hold steady at 60 frames per second.

```
1   use strict;
2   use warnings;
3   use SDL;
4   use SDL::Event;
5   use SDL::Events;
6   use SDLx::App;
7
8   my $app = SDLx::App->new(
9                   width  => 200,
```

```
                    height => 200,
                    title  => 'Pew Pew'
);

my ( $start, $end, $delta_time, $FPS, $frames ) = ( 0, 0, 0, 0, 0 );

# aim for a rate of 60 frames per second
my $fixed_rate = 60;

# compensate for times stored in microseconds
my $fps_check = (1000 / $fixed_rate );

my $quit = 0;

# start laser on the left
my $laser = 0;

sub get_events {
        my $event = SDL::Event->new();

        SDL::Events::pump_events;

        while ( SDL::Events::poll_event($event) ) {
                $quit = 1 if $event->type == SDL_QUIT;
        }
}

sub calculate_next_positions {
        $laser++;

        $laser = 0 if $laser > $app->w;
}

sub render {
        # draw the background first
        $app->draw_rect( [ 0, 0, $app->w, $app->h ], 0 );

        # draw the laser
```

```
48              $app->draw_rect( [ $laser, $app->h / 2, 10, 2 ], [ 255, 0, 0, 255 ] );
49
50              # draw the FPS
51              $app->draw_gfx_text( [ 10, 10 ], [ 255, 0, 255, 255 ], "FPS: $FPS" );
52
53              $app->update();
54      }
55
56      # Called at the end of each frame, whether we draw or not
57      sub calculate_fps_at_frame_end
58      {
59              # Ticks are microseconds since load time
60              $end = SDL::get_ticks();
61
62      # smooth the frame rate by averaging over 10 frames
63              if ( $frames < 10 ) {
64                      $frames++;
65                      $delta_time += $end - $start;
66              }
67              else {
68                      # frame rate is Frames * 100 / Time Elapsed in us
69          $FPS        = int( ( $frames * 100 ) / $delta_time )
70                      if $delta_time != 0;
71
72                      # reset metrics
73                      $frames     = 0;
74                      $delta_time = 0;
75              }
76      }
77
78      while ( !$quit ) {
79              # Get the time for the starting of the frame
80              $start = SDL::get_ticks();
81
82              get_events();
83
84              # if fixing the lower bounds of the frame rate
85              if( $ARGV[1] )
```

```
86              {
87                      # if delta time is going too slow for frame check
88                      if ( $delta_time > $fps_check ) {
89
90                              calculate_fps_at_frame_end();
91
92              # skip rendering and collision detections
93              # (heavy functions in the game loop)
94                              next;
95                      }
96              }
97
98              calculate_next_positions();
99              render();
100
101             # a normal frame with rendering actually performed
102             calculate_fps_at_frame_end();
103
104             # if fixing the upper bounds of the frame rate
105             if ( $ARGV[0] ) {
106
107                     # if delta time is going too fast compared to the frame check
108                     if ( $delta_time < $fps_check ) {
109
110                             # delay for the difference
111                             SDL::delay( $fps_check - $delta_time );
112                     }
113             }
114     }
```

This method is generally sufficient for most computers. The animations will be smooth enough to provide the same gameplay even on machines with different hardware.

However, this method still has some serious problems. First, if a computer is too slow to sustain a rate of 60 FPS, the game will skip rendering some frames, leading to sparse and jittery animation.it will skip a lot of rendering, and the animation will look sparse and

jittery. It might be better to set a lower bounds of 30 FPS, though it's difficult to predict the best frame rate for a user.

The worst problem is that this technique still ties rendering speed to the CPU speed: a very fast computer will waste CPU cycles delaying.

5.1.2 Variable FPS

To fix the problem of a computer being consistently too fast or too slow for the hard-coded FPS rate is to adjust the FPS rate accordingly. A slow CPU may limit itself to 30 FPS, while a fast CPU might run at 300 FPS. Although you may achieve a consistent rate this way (consistent for any one particular computer), this technique still presents the problem of differing animation speeds between different computers.

Better solutions are available.

5.2 Integrating Physics

The problem caused by coupling rendering to the CPU speed has a convenient solution. Instead of updating object positions based on how fast the computer can get through the game loop, derive their positions from a physical model based on the passage of time. Objects moving according to real world time will have consistent behavior at all CPU speeds and smooth interpolation between frames. `SDLx::App` provides this behavior through movement and show handlers.

Consider a simple physics model for the laser has a consistent horizontal velocity in pixels per time step at the window's mid-point:

```
X = Velocity * time step,
Y = 100
```

Assuming a velocity of 10, the laser will pass through the coordinates:

```
  0, 100
 10, 100
 20, 100
 30, 100
...
200, 100
```

Note that the speed of processing the game loop no longer matters. The position of the laser depends instead on the passage of real time.

The biggest problem with this approach is the required bookkeeping for the many objects and callbacks. The implementation of such complex models is non-trivial; see the lengthy discussion in the documentation of the `SDLx::Controller` module.

`SDLx::App` using the `SDLx::Controller` module provide callbacks to handle both aspects of this type of game loop. One is the the movement handler, which is a callback where calculations of the next step for each relevant data point is calculated. In the above example the movement handler would calculate the `X` and `Y` values, for each time step between the frames of animations.

When we are ready to render the frame it is handled by the show handler. In the above example that would mean the show handler would print or render the `X, Y` values.

5.2.1 Laser in Real Time

This version of the laser example demonstrates the use of movement, show handlers, and a simple physics model. This example also shows how `SDLx::App` can do more of the work, even providing the entire game loop:

```
1       use strict;
2       use warnings;
3       use SDL;
```

```
4       use SDL::Event;
5       use SDLx::App;
6
7       my $app = SDLx::App->new(
8           width  => 200,
9           height => 200,
10          title  => 'Pew Pew'
11      );
12
13      my $laser    = 0;
14      my $velocity = 10;
15
16      $app->add_event_handler( \&quit_event );
17
18      # tell app to handle the appropriate times to
19      # call both rendering and physics calculation
20
21      $app->add_move_handler( \&calculate_laser );
22      $app->add_show_handler( \&render_laser );
23
24      $app->run();
25
26      sub quit_event {
27          my $event      = shift;
28          my $controller = shift;
29
30          $controller->stop if $event->type == SDL_QUIT;
31      }
32
33      sub calculate_laser {
34
35          # The step is the difference in Time calculated for the next jump
36          my ( $step, $app, $t ) = @_;
37          $laser += $velocity * $step;
38          $laser = 0 if $laser > $app->w;
39      }
40
41      sub render_laser {
```

```
42          my ( $delta, $app ) = @_;
43
44          # The delta can be used to render blurred frames
45
46          # draw the background first
47          $app->draw_rect( [ 0, 0, $app->w, $app->h ], 0 );
48
49          # draw the laser
50          $app->draw_rect( [ $laser, $app->h / 2, 10, 2 ], [ 255, 0, 0, 255 ] );
51          $app->update();
52      }
```

To learn more about this topic please, see an excellent blog post by **GafferOnGames.com**: `HTTP://GafferOnGames.Com/game-physics/fix-your-timestep`.

6

Pong!

6.1 The Game

Pong is one of the first popular video games in the world. It was created by Allan Alcorn for Atari Inc. and released in 1972, being Atari's first game ever, and sparkling the beginning of the video game industry.

Pong simulates a table tennis match ("ping pong"), where you try to defeat your opponent by earning a higher score. Each player controls a paddle moving it vertically on the screen, and use it to hit a bouncing ball back and forth. You earn a point if your opponent is unable to return the ball to your side of the screen.

And now we're gonna learn how to create one ourselves in Perl and SDL.

6.1.1 Getting our feet wet

Let's start by making a simple screen for our Pong clone. Open a file in your favourite text editor and type:

```
+ #!/usr/bin/perl
+ use strict;
+ use warnings;
+
+ use SDL;
+ use SDLx::App;
+
+ # create our main screen
+ my $app = SDLx::App->new(
+     width        => 500,
+     height       => 500,
+     title        => 'My Pong Clone!',
+     dt           => 0.02,
+     exit_on_quit => 1,
+ );
+
+ # let's roll!
+ $app->run;
```

Save this file as `"pong.pl"` and run it by typing on the command line:

```
rl pong.pl
```

You should see a 500x500 black window entitled "*My Pong Clone!*". In our SDLx::App construction we also set a time interval (dt) of 0.02 for the game loop, and let it handle SDL_QUIT events for us. If any of the arguments above came as a surprise to you, please refer to previous chapters for an in-depth explanation.

6.1.2 Game Objects

There are three main game objects in Pong: the player's paddle, the enemy's paddle, and a bouncing ball.

Paddles are rectangles moving vertically on the screen, and can be easily represented with SDLx::Rect objects. First, put `SDLx::Rect` in your module's declarations:

```
  use SDL;
  use SDLx::App;
+ use SDLx::Rect;
```

Now let's add a simple hash reference in our code to store our player's paddle, between the call to `SDLx::App->new()` and `$app->run`.

We'll use a hash reference instead of just assigning a `SDLx::Rect` to a variable because it will allow us to store more information later on. If you were building a more complex game, you should consider using actual objects. For now, a simple hash reference will suffice:

```
+ my $player1 = {
+      paddle => SDLx::Rect->new( 10, $app->h / 2, 10, 40),
+ };
```

As we know, `SDLx::Rect` objects receive four arguments: x, y, width and height, in this order. So in the code above we're creating a 10x40 paddle rect for player 1, on the left side of the screen (`x = 10`) and somewhat in the center (`y = $app->h / 2`).

Let's do the same for player 2, adding the following code right after the one above:

```
+ my $player2 = {
+      paddle => SDLx::Rect->new( $app->w - 20, $app->h / 2, 10, 40),
+ };
```

Player 2's paddle, also 10x40, needs to go to the right end of the screen. So we make its `x` position as our screen's width minus 20. Since the paddle has a width of 10 itself and the `x` position refers to the rect's top-left corner, it will leave a space of 10 pixels between its rightmost side and the end of the screen, just like we did for player 1.

Finally, the bouncing ball, a 10x10 rect in the middle of the screen:

```
+ my $ball = {
+     rect => SDLx::Rect->new( $app->w / 2, $app->h / 2, 10, 10 ),
+ };
```

Yes, it's a "square ball", just like the original :)

Show me what you got!

Now that we created our game objects, let's add a 'show' handler to render them on the screen:

```
+ $app->add_show_handler(
+     sub {
+         # first, we clear the screen
+         $app->draw_rect( [0, 0, $app->w, $app->h], 0x000000FF );
+
+         # then we render the ball
+         $app->draw_rect( $ball->{rect}, 0xFF0000FF );
+
+         # ... and each paddle
+         $app->draw_rect( $player1->{paddle}, 0xFF0000FF );
+         $app->draw_rect( $player2->{paddle}, 0xFF0000FF );
+
+         # finally, we update the screen
+         $app->update;
+     }
+ );
```

Our approach is rather simple here, "clearing" the screen by painting a black rectangle the size of the screen, then using `draw_rect()` calls to paint opaque red (`0xFF0000FF`) rectangles in each object's position.

The result can be seen on the screenshot below:

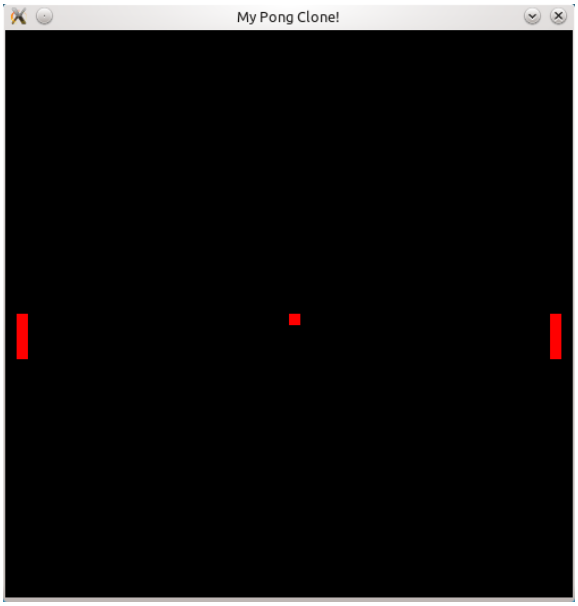

Figure 6.1: First view of our Pong clone

6.1.3 Moving the Player's Paddle

It's time to let the player move the left paddle! Take a few moments to recap what motion is all about: changing your object's position with respect to time. If it's some sort of magical teleportation repositioning, just change the (x,y) coordinates and be done with it. If however, we're talking about real motion, we need to move at a certain speed. Our paddle will have constant speed, so we don't need to worry about acceleration. Also, since it will only move vertically, we just need to add the vertical (y) velocity. Let's call it `v_y` and add it to our paddle structure:

```
  my $player1 = {
       paddle => SDLx::Rect->new( 10, $app->h / 2, 10, 40),
+      v_y    => 0,
  };
```

Ok, now we have an attribute for vertical velocity (`v_y`) in our paddle, so what? How will this update the `y` position of the paddle? Well, velocity is how much displacement happens in a unit of time, like 20 km/h or 4 m/s. In our case, the unit of time is the app's `dt`, so all we have to do is move the paddle `v_y` pixels per `dt`. Here is where the motion handlers come in handy:

```
+ # handles the player's paddle movement
+ $app->add_move_handler( sub {
+     my ( $step, $app ) = @_;
+     my $paddle = $player1->{paddle};
+     my $v_y = $player1->{v_y};
+
+     $paddle->y( $paddle->y + ( $v_y * $step ) );
+ });
```

If you recall previous chapters, the code above should be pretty straightforward. When `v_y` is 0 at any given run cycle, the paddle won't change its `y` position. If, however, there is a vertical velocity, we update the `y` position based on how much of the expected cycle time (our app's "dt") has passed. A value of 1 in `$step` indicates a full cycle went through, and makes `$v_y * $step` the same as `$v_y * 1`, thus, plain `$v_y` - which is the desired speed for our cycle. Should the handler be called in a shorter cycle, we'll move only the relative factor of that.

Player 2? Rinse and repeat

We're not going to worry at this point about moving your nemesis' paddle, but since it uses the same motion mechanics of our player's, it won't hurt to prepare it:

```
  my $player2 = {
       paddle => SDLx::Rect->new( $app->w - 20, $app->h / 2, 10, 40),
```

```
+      v_y    => 0,
  };
```

And add a simple motion handler, just like our player's:

```
+ # handles AI's paddle movement
+ $app->add_move_handler( sub {
+     my ( $step, $app ) = @_;
+     my $paddle = $player2->{paddle};
+     my $v_y = $player2->{v_y};
+
+     $paddle->y( $paddle->y + ( $v_y * $step ) );
+ });
```

Back to our Player: Move that Paddle!

We have preset `v_y` to zero as the paddle's initial velocity, so our player's paddle won't go haywire when the game starts. But we still need to know when the user wants to move it up or down the screen. In order to do that, we can bind the up and down arrow keys of the keyboard to positive and negative velocities for our paddle, through an event hook. Since we're going to use some event constants like `SDLK_DOWN`, we need to load the SDL::Events module:

```
use SDL;
use SDL::Events;
use SDLx::App;
use SDLx::Rect;
```

Then we can proceed to create our event hook:

```
# handles keyboard events
$app->add_event_handler(
    sub {
        my ( $event, $app ) = @_;
```

```
            # user pressing a key
            if ( $event->type == SDL_KEYDOWN ) {

                # up arrow key means going up (negative vel)
                if ( $event->key_sym == SDLK_UP ) {
                    $player1->{v_y} = -2;
                }
                # down arrow key means going down (positive vel)
                elsif ( $event->key_sym == SDLK_DOWN ) {
                    $player1->{v_y} = 2;
                }
            }
            # user releasing a key
            elsif ( $event->type == SDL_KEYUP ) {

                # up or down arrow keys released, stop the paddle
                if (
                        $event->key_sym == SDLK_UP
                     or $event->key_sym == SDLK_DOWN
                ) {
                    $player1->{v_y} = 0;
                }
            }
        }
    );
```

Again, nothing new here. Whenever the user presses the up arrow key, we want the paddle to go up. Keep in mind our origin point (0,0) in SDL is the top-left corner, so a negative `v_y` will decrease the paddle's `y` and send it **up** the screen. Alternatively, we add a positive value to `v_y` whenever the user presses the down arrow key, so the paddle will move **down**, away from the top of the screen. When the user releases either the up or down arrow keys, we stop the paddle by setting `v_y` to 0.

6.1.4 A Bouncing Ball

How about we animate the game ball? The movement itself is pretty similar to our paddle's, except the ball will also have a horizontal velocity ("v_x") component, letting it move all over the screen.

First, we add the velocity components to our ball structure:

```
my $ball = {
    rect => SDLx::Rect->new( $app->w / 2, $app->h / 2, 10, 10 ),
    v_x  => -2.7,
    v_y  => 1.8,
};
```

The ball will have an initial velocity of -2.7 horizontally (just as a negative vertical velocity moves the object up, a negative horizontal velocity will move it towards the left side of the screen), and 1.8 vertically. Next, we create a motion handler for the ball, updating the ball's `x` and `y` position according to its speed:

```
# handles the ball movement
$app->add_move_handler( sub {
    my ( $step, $app ) = @_;
    my $ball_rect = $ball->{rect};

    $ball_rect->x( $ball_rect->x + ($ball->{v_x} * $step) );
    $ball_rect->y( $ball_rect->y + ($ball->{v_y} * $step) );
});
```

This is just like our paddle's motion handler: we update the ball's `x` and `y` position on the screen according to the current velocity. If you are paying attention, however, you probably realized the code above is missing a very important piece of logic. Need a clue? Try running the game as it is. You'll see the ball going, going, and... gone!

We need to make sure the ball is bound to the screen. That is, it needs to collide and bounce back whenever it reaches the top and bottom edges of the screen. So let's change our ball's motion handler a bit, adding this functionality:

```
# handles the ball movement
$app->add_move_handler( sub {
    my ( $step, $app ) = @_;
    my $ball_rect = $ball->{rect};

    $ball_rect->x( $ball_rect->x + ($ball->{v_x} * $step) );
    $ball_rect->y( $ball_rect->y + ($ball->{v_y} * $step) );

    # collision to the bottom of the screen
    if ( $ball_rect->bottom >= $app->h ) {
        $ball_rect->bottom( $app->h );
        $ball->{v_y} *= -1;
    }

    # collision to the top of the screen
    elsif ( $ball_rect->top <= 0 ) {
        $ball_rect->top( 0 );
        $ball->{v_y} *= -1;
    }
});
```

If the new y (`"bottom"` or `"top"`) value would take the ball totally or partially off the screen, we replace it with the farthest position possible (making it "touch" that edge of the screen) and reverse `v_y`, so it will go the opposite way on the next cycle, bouncing back into the screen.

He shoots... and scores!!

So far, so good. But what should happen when the ball hits the left or right edges of the screen? Well, according to the rules of Pong, this means the player on the opposite side scored a point, and the ball should go back to the center of the screen. Let's begin by adding a 'score' attribute for each player:

```
my $player1 = {
     paddle => SDLx::Rect->new( 10, $app->h / 2, 10, 40),
```

```
       v_y    => 0,
+      score  => 0,
  };

  my $player2 = {
       paddle => SDLx::Rect->new( $app->w - 20, $app->h / 2, 10, 40),
       v_y    => 0,
+      score  => 0,
  };
```

Now we should teach the ball's motion handler what to do when it reaches the left and right corners:

```
# handles the ball movement
$app->add_move_handler( sub {
    my ( $step, $app ) = @_;
    my $ball_rect = $ball->{rect};

    $ball_rect->x( $ball_rect->x + ($ball->{v_x} * $step) );
    $ball_rect->y( $ball_rect->y + ($ball->{v_y} * $step) );

    # collision to the bottom of the screen
    if ( $ball_rect->bottom >= $app->h ) {
        $ball_rect->bottom( $app->h );
        $ball->{v_y} *= -1;
    }

    # collision to the top of the screen
    elsif ( $ball_rect->top <= 0 ) {
        $ball_rect->top( 0 );
        $ball->{v_y} *= -1;
    }

    # collision to the right: player 1 score!
    elsif ( $ball_rect->right >= $app->w ) {
        $player1->{score}++;
        reset_game();
```

```
            return;
        }

        # collision to the left: player 2 score!
        elsif ( $ball_rect->left <= 0 ) {
            $player2->{score}++;
            reset_game();
            return;
        }
    });
```

If the ball's right hits the right end of the screen (the app's width), we increase player 1's score, call `reset_game()`, and return without updating the ball's position. If the ball's left hits the left end of the screen, we do the same for player 2.

We want the `reset_game()` function called above to set the ball back on the center of the screen, so let's make it happen:

```
sub reset_game {
    $ball->{rect}->x( $app->w / 2 );
    $ball->{rect}->y( $app->h / 2 );
}
```

6.1.5 Collision Detection: The Ball and The Paddle

We already learned how to do some simple collision detection, namely between the ball and the edges of the screen. Now it's time to take it one step further and figure out how to check whether the ball and the paddles are overlapping one another (colliding, or rather, intersecting). This is done via the Separating Axis Theorem, which roughly states that two convex shapes in a 2D plane are **not** intersecting if and only if we can place a line separating them. Since our rect objects (the ball and paddles) are both axis-aligned, we can simply pick one, and there will be only 4 possible lines to test: its left, right, top and bottom. If the other object is completely on one side of any of those lines, then there is **no** collision. But if all four conditions are false, they are intersecting.

To put it in more general terms, if we have 2 rects, A and B, we can establish the following conditions, illustrated by the figure below:

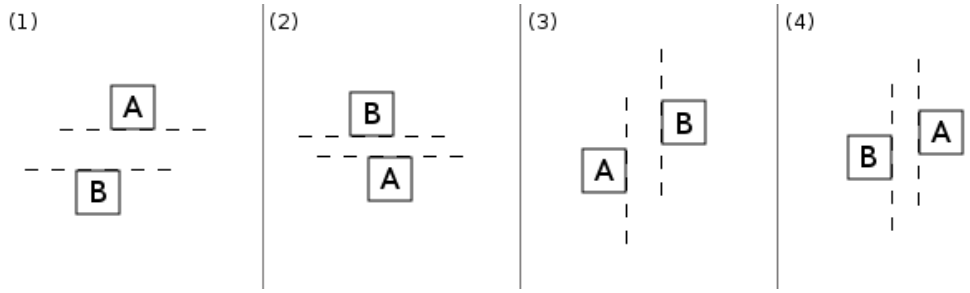

Figure 6.2: if B is completely to the left, right, top or bottom of A, they do NOT intersect

- if A's bottom side is above B's top side, then A is completely above B (fig. 6.2.1).
- if A's top side is below B's bottom side, then A is completely below B (fig. 6.2.2).
- if A's right side is to the left of B's left side, then A is completely to the left of B (fig. 6.2.3).
- if A's left side is to the right of B's right side, then A is completely to the right of B (fig 6.2.4).

Keeping in mind that our origin point (0,0) in SDL is the top-left corner, we can translate the rules above to the following generic `check_collision()` function, receiving two rect objects and returning true if they collide:

```
sub check_collision {
    my ($A, $B) = @_;

    return if $A->bottom < $B->top;
    return if $A->top    > $B->bottom;
    return if $A->right  < $B->left;
    return if $A->left   > $B->right;
```

```
    # if we got here, we have a collision!
    return 1;
}
```

We can now use it in the ball's motion handler to see if it hits any of the paddles:

```
# handles the ball movement
$app->add_move_handler( sub {
    my ( $step, $app ) = @_;
    my $ball_rect = $ball->{rect};

    $ball_rect->x( $ball_rect->x + ($ball->{v_x} * $step) );
    $ball_rect->y( $ball_rect->y + ($ball->{v_y} * $step) );

    # collision to the bottom of the screen
    if ( $ball_rect->bottom >= $app->h ) {
        $ball_rect->bottom( $app->h );
        $ball->{v_y} *= -1;
    }

    # collision to the top of the screen
    elsif ( $ball_rect->top <= 0 ) {
        $ball_rect->top( 0 );
        $ball->{v_y} *= -1;
    }

    # collision to the right: player 1 score!
    elsif ( $ball_rect->right >= $app->w ) {
        $player1->{score}++;
        reset_game();
        return;
    }

    # collision to the left: player 2 score!
    elsif ( $ball_rect->left <= 0 ) {
        $player2->{score}++;
        reset_game();
```

```
            return;
        }

        # collision with player1's paddle
        elsif ( check_collision( $ball_rect, $player1->{paddle} )) {
            $ball_rect->left( $player1->{paddle}->right );
            $ball->{v_x} *= -1;
        }

        # collision with player2's paddle
        elsif ( check_collision( $ball_rect, $player2->{paddle} )) {
            $ball->{v_x} *= -1;
            $ball_rect->right( $player2->{paddle}->left );
        }
    });
```

That's it! If the ball hits player1's paddle, we reverse its horizontal velocity (v_x) to make it bounce back, and set its left edge to the paddle's right so they don't overlap. Then we do the exact same thing for the other player's paddle, except this time we set the ball's right to the paddle's left - since the ball is coming from the other side.

6.1.6 Artificial Stupidity

Our Pong game is almost done now. We record the score, the ball bounces around, we keep track of each player's score, and we can move the left paddle with the up and down arrow keys. But this will be a very dull game unless our nemesis moves too!

There are several complex algorithms to model artificial intelligence, but we don't have to go that far for a simple game like this. What we're going to do is make player2's paddle follow the ball wherever it goes, by adding the following to its motion handler:

```
# handles AI's paddle movement
$app->add_move_handler( sub {
    my ( $step, $app ) = @_;
    my $paddle = $player2->{paddle};
```

```
        my $v_y = $player2->{v_y};

+       if ( $ball->{rect}->y > $paddle->y ) {
+           $player2->{v_y} = 1.5;
+       }
+       elsif ( $ball->{rect}->y < $paddle->y ) {
+           $player2->{v_y} = -1.5;
+       }
+       else {
+           $player2->{v_y} = 0;
+       }

        $paddle->y( $paddle->y + ( $v_y * $step ) );
    });
```

If the ball's "y" value (its top) is greater than the nemesis' paddle, it means the ball is below it, so we give the paddle a positive velocity, making it go downwards. On the other hand, if the ball has a lower "y" value, we set the nemesis' `v_y` to a negative value, making it go up. Finally, if the ball is somewhere in between those two values, we keep the paddle still.

6.1.7 Cosmetics: Displaying the Score

How about we display the score so the player can see who's winning? To render a text string in SDL, we're going to use the SDLx::Text module, so let's add it to the beginning of our code:

```
use SDL;
use SDL::Events;
use SDLx::App;
use SDLx::Rect;
use SDLx::Text;
```

Now we need to create the score object:

```
my $score = SDLx::Text->new( font => 'font.ttf', h_align => 'center' );
```

The optional `font` parameter specifies the path to a TrueType Font. Here we are loading the '*font.ttf*' file, so feel free to change this to whatever font you have in your system. Otherwise, you can leave it out and use the bundled default font. The `h_align` parameter lets us choose a horizontal alignment for the text we put in the object. It defaults to '*left*', so we make it '*center*' instead.

All that's left is using this object to write the score on the screen, so we update our 'show' handler:

```
    $app->add_show_handler(
        sub {
            # first, we clear the screen
            $app->draw_rect( [0, 0, $app->w, $app->h], 0x000000FF );

            # then we render the ball
            $app->draw_rect( $ball->{rect}, 0xFF0000FF );

            # ... and each paddle
            $app->draw_rect( $player1->{paddle}, 0xFF0000FF );
            $app->draw_rect( $player2->{paddle}, 0xFF0000FF );

+           # ... and each player's score!
+           $score->write_to(
+               $app,
+               $player1->{score} . ' x ' . $player2->{score}
+           );

            # finally, we update the screen
            $app->update;
        }
    );
```

The `write_to()` call will write to any surface passed as the first argument - in our case, the app itself. The second argument, as you probably figured, is the string to be rendered. Note that the string's position is relative to the surface it writes to, and defaults to (0,0). Since we told it to center horizontally, it will write our text to the top/center, instead of top/left.

The result, and our finished game, can be seen on the figure below:

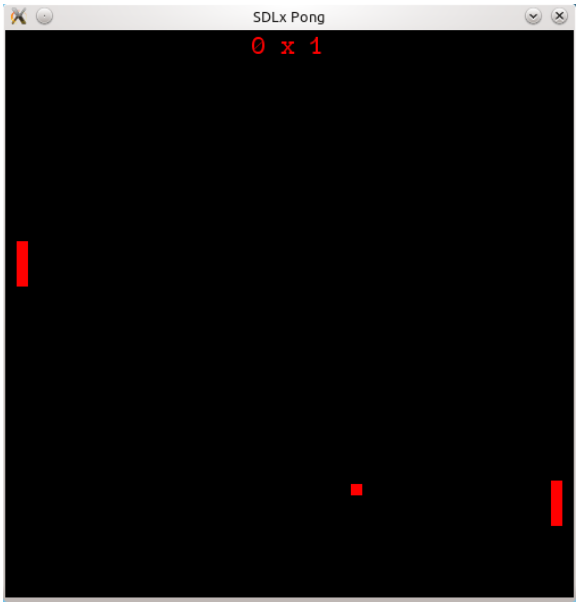

Figure 6.3: our finished Pong clone, in all its glory

6.1.8 Exercises

1. Every time a player scores, the ball goes back to the middle but has the same sense and direction as before. See if you can make it restart at a random direction instead.

2. Red is boring, you want to make a completely psychedelic Pong! Pick 3 different colours and make each paddle oscillate between them every time the ball hits it.

See if you can solve the exercises above by yourself, to make sure you understand what is what and how to do things in SDL Perl. Once you're done, check out the answers below. Of course, there's always more than one way to do things, so the ones below are not the only possible answers.

Answers

1. To make the ball restart at a random direction, we can improve our `reset_game()` function to set the ball's `v_x` and `v_y` to a random value between, say, 1.5 and 2.5, or -1.5 and -2.5:

```
sub reset_game {
    $ball->{rect}->x( $app->w / 2 );
    $ball->{rect}->y( $app->h / 2 );

    $ball->{v_x} = (1.5 + int rand 1) * (rand 2 > 1 ? 1 : -1);
    $ball->{v_y} = (1.5 + int rand 1) * (rand 2 > 1 ? 1 : -1);
}
```

2. We can either choose one colour set for both paddles or one for each. Let's go with just one set, as an array of hex values representing our colours. We'll also hold the index for the current colour for each player:

```
+ my @colours = qw( 0xFF0000FF 0x00FF00FF 0x0000FFFF 0xFFFF00FF );

  my $player1 = {
       paddle => SDLx::Rect->new( 10, $app->h / 2, 10, 40),
       v_y    => 0,
       score  => 0,
+      colour => 0,
  };

  my $player2 = {
       paddle => SDLx::Rect->new( $app->w - 20, $app->h / 2, 10, 40),
       v_y    => 0,
       score  => 0,
+      colour => 0,
  };
```

Next we make it update the `colour` every time the ball hits the paddle:

```
# handles the ball movement
$app->add_move_handler( sub {
    my ( $step, $app ) = @_;
    my $ball_rect = $ball->{rect};

    $ball_rect->x( $ball_rect->x + ($ball->{v_x} * $step) );
    $ball_rect->y( $ball_rect->y + ($ball->{v_y} * $step) );

    # collision to the bottom of the screen
    if ( $ball_rect->bottom >= $app->h ) {
        $ball_rect->bottom( $app->h );
        $ball->{v_y} *= -1;
    }

    # collision to the top of the screen
    elsif ( $ball_rect->top <= 0 ) {
        $ball_rect->top( 0 );
        $ball->{v_y} *= -1;
    }

    # collision to the right: player 1 score!
    elsif ( $ball_rect->right >= $app->w ) {
        $player1->{score}++;
        reset_game();
        return;
    }

    # collision to the left: player 2 score!
    elsif ( $ball_rect->left <= 0 ) {
        $player2->{score}++;
        reset_game();
        return;
    }

    # collision with player1's paddle
    elsif ( check_collision( $ball_rect, $player1->{paddle} )) {
        $ball_rect->left( $player1->{paddle}->right );
        $ball->{v_x} *= -1;
```

```
            $player1->{colour} = ($player1->{colour} + 1) % @colours;
        }

        # collision with player2's paddle
        elsif ( check_collision( $ball_rect, $player2->{paddle} )) {
            $ball->{v_x} *= -1;
            $ball_rect->right( $player2->{paddle}->left );
            $player2->{colour} = ($player2->{colour} + 1) % @colours;
        }
    });
```

Finally, we change our 'show' handler to use the current colour referenced by `colour`, instead of the previously hardcoded red (0xFF0000FF):

```
    $app->add_show_handler(
        sub {
            # first, we clear the screen
            $app->draw_rect( [0, 0, $app->w, $app->h], 0x000000FF );

            # then we render the ball
            $app->draw_rect( $ball->{rect}, 0xFF0000FF );

            # ... and each paddle
-           $app->draw_rect( $player1->{paddle}, 0xFF0000FF );
+           $app->draw_rect( $player1->{paddle}, $colours[ $player1->{colour} ] );
-           $app->draw_rect( $player2->{paddle}, 0xFF0000FF );
+           $app->draw_rect( $player2->{paddle}, $colours[ $player2->{colour} ] );

            # ... and each player's score!
            $score->write_to(
                $app,
                $player1->{score} . ' x ' . $player2->{score}
            );

            # finally, we update the screen
            $app->update;
```

```
        }
    );
```

6.2 Author

This chapter’s content graciously provided by Breno G. de Oliveira (`garu`).

7

Tetris

7.1 Eye Candy and Code

In this chapter we work on creating the classic Tetris game using what we have learned so far. Get the tetris code from `HTTPS://GitHub.Com/PerlGameDev/SDL_Manual/raw/master/games/tetris.zip`. To run the game invoke in the extracted folder.

```
perl tetris.pl
```

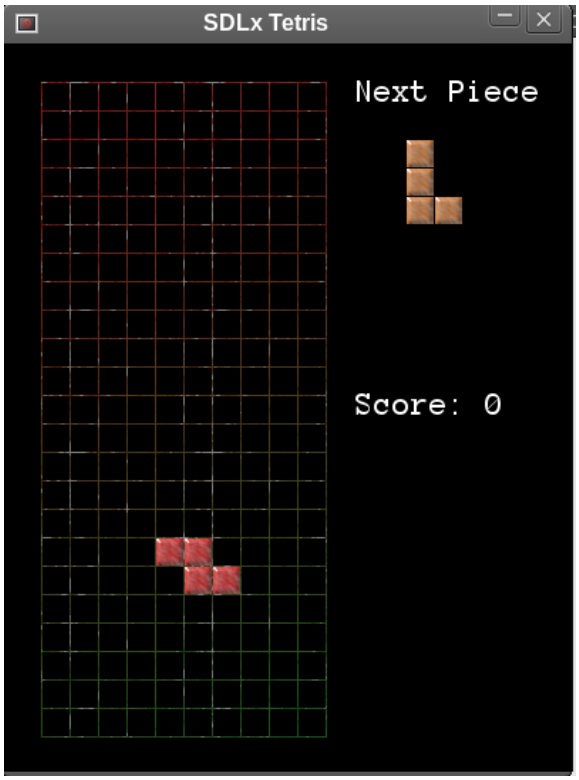

Figure 7.1: Tetris using SDLx Perl

7.2 The Game Window

First we will make our window with a fixed size so we can place our art work in a fixed format.

```
use strict;
use warnings;

use SDL;
use SDL::Event;
```

```
use SDL::Events;
use SDLx::App;

# create our main screen
my $app = SDLx::App->new(
   w            => 400,
   h            => 512,
   exit_on_quit => 1,
   dt           => 0.2,
   title        => 'SDLx Tetris'
);
```

7.3 Loading Artwork

We can load our artwork simply by storing an array of `SDLx::Surfaces`.

```
 use SDL;
+use SDLx::Surface;
```

Next we load up the artwork into an array.

```
+my $back  = SDLx::Surface->load( 'data/tetris_back.png' );
+my @piece = (undef);
+push(@piece, SDLx::Surface->load( "data/tetris_$_.png" )) for(1..7);
```

The background is held in the `$back` surface, and the pieces are held in the `@piece` array. Later on we will blit these onto our main screen as we need.

7.4 Data Structures

In Tetris the blocks are critical pieces of data that must be represented in code such that it is easy to access, and quick to perform calculations on. A hash will allow us to quickly access our pieces, based on their keys.

```
my %pieces = (
  I => [0,5,0,0,
        0,5,0,0,
        0,5,0,0,
        0,5,0,0],
  J => [0,0,0,0,
        0,0,6,0,
        0,0,6,0,
        0,6,6,0],
  L => [0,0,0,0,
        0,2,0,0,
        0,2,0,0,
        0,2,2,0],
  O => [0,0,0,0,
        0,3,3,0,
        0,3,3,0,
        0,0,0,0],
  S => [0,0,0,0,
        0,4,4,0,
        4,4,0,0,
        0,0,0,0],
  T => [0,0,0,0,
        0,7,0,0,
        7,7,7,0,
        0,0,0,0],
  Z => [0,0,0,0,
        1,1,0,0,
        0,1,1,0,
        0,0,0,0],
);
```

Further more we have a 1-dimensional array for each piece that represents a grid of the piece.

The grid of each piece is filled with empty spaces and a number from 1 to 7. When this grid is imposed on the game grid, we can use the non zero number to draw the right piece block on to it.

The non zero number corresponds to the images file that we loaded ealier.

```
push(@piece, SDLx::Surface->load( "data/tetris_$_.png" )) for(1..7);
```

7.5 Selecting Pieces

```
use strict;
use warnings;

+use List::Util qw(shuffle min max);
```

We will use the List::Util module to provide us with some neeeded functions.

```
    Z => [0,0,0,0,
          1,1,0,0,
          0,1,1,0,
          0,0,0,0],
);

+my $next_tile         = shuffle(keys %pieces);
+my $curr_tile         = [undef, 4, 0];
+   @{$curr_tile->[0]} = @{$pieces{$next_tile}};
+   $next_tile         = shuffle(keys %pieces);
```

We will randomly pick a `$next_tile` and then set the piece data for our first piece in `$curr_tile`. Then we will pick another tile for our `$next_tile`.

7.6 Moving Pieces

```
 push(@piece, SDLx::Surface->load( "data/tetris_$_.png" )) for(1..7);

+# to check for collisions we compare the position of the moving piece with the non-movin piece
+my $grid  = []; # moving piece
+my $store = []; # non-moving pieces
 my %pieces = (
      I => [0,5,0,0,
```

In our conceptual model of Tetris we have two grids that overlap each other. First we have the `$grid` where the piece that is moving is stored. Once a piece has collided with sometime we move it to `$store` grid and hold it there until a line is cleared.

```
 $next_tile         = shuffle(keys %pieces);
```

To rotate a piece we apply a transformation on each element of the piece.

```
+ sub rotate_piece {
+     my $_piece   = shift;
+     my $_rotated = [];
+     my $_i = 0;
+     for(@{$_piece}) {
+         $_rotated->[$_i + (($_i%4+1)*3) - (5*int($_i/4))] = $_;
+         $_i++;
+     }
+     return $_rotated;
+ }
```

Additionally we do a simple collision checking between the non zero elements in the pieces with the direction the user wants to move.

```
+ sub can_move_piece {
+     my $direction = shift;
+     my $amount    = shift || 1;
```

```
+     for my $y (0..3) {
+         for my $x (0..3) {
+             if($curr_tile->[0]->[$x + 4 * $y]) {
+                 return if $direction eq 'left'
+                         && $x - $amount + $curr_tile->[1]  < 0;
+                 return if $direction eq 'right'
+                         && $x + $amount + $curr_tile->[1]  > 9;
+                 return if $direction eq 'down'
+                         && int($y + $amount + $curr_tile->[2]) > 22;
+
+                 return if $direction eq 'right'
+                         && $store->[ $x + $amount +
+                                      $curr_tile->[1] +
+                                      10 * int($y + $curr_tile->[2]) ];
+                 return if $direction eq 'left'
+                         && $store->[ $x - $amount +
+                                      $curr_tile->[1] +
+                                      10 * int($y + $curr_tile->[2]) ];
+                 return if $direction eq 'down'
+                         && $store->[ $x +
+                                      $curr_tile->[1]
+                                      + 10 * int($y + $amount + $curr_tile->[2]) ];
+             }
+         }
+     }
+     return 1;
+ }
```

Finally we move the move piece by using the collision check and overlaying the piece array into the `@grid` for each next position.

```
+ sub move_piece {
+     my $direction = shift;
+     my $amount    = shift || 1;
+     if($direction eq 'right') {
+         $curr_tile->[1] += $amount;
+     }
```

```
+     elsif($direction eq 'left') {
+         $curr_tile->[1] -= $amount;
+     }
+     elsif($direction eq 'down') {
+         $curr_tile->[2] += $amount;
+     }
+
+     @{$grid} = ();
+     for my $y (0..3) {
+         for my $x (0..3) {
+             if($curr_tile->[0]->[$x + 4 * $y]) {
+                 $grid->[ $x + $curr_tile->[1] + 10 * ($y + int($curr_tile->[2])) ]
+                          = $curr_tile->[0]->[$x + 4 * $y];
+             }
+         }
+     }
+ }

+ sub store_piece {
+     for my $y (0..3) {
+         for my $x (0..3) {
+             if($curr_tile->[0]->[$x + 4 * $y]) {
+                 $store->[ $x + $curr_tile->[1] + 10 * ($y + int($curr_tile->[2])) ]
+                          = $curr_tile->[0]->[$x + 4 * $y];
+             }
+         }
+     }
+ }
```

Finally we hook it into the event handler where we use the events to move the pieces in the right direction.

```
+ sub trigger_move_event_handler {
+     my ( $event, $app ) = @_;
+     if( $event->type == SDL_KEYDOWN ) {
+         my $key = $event->key_sym;
+         if( $event->key_sym & (SDLK_LEFT|SDLK_RIGHT|SDLK_UP|SDLK_DOWN) ) {
```

```
+             if($key == SDLK_LEFT && can_move_piece('left')) {
+                 move_piece('left');
+             }
+             elsif($key == SDLK_RIGHT && can_move_piece('right')) {
+                 move_piece('right');
+             }
+             elsif($key == SDLK_DOWN && can_move_piece('down')) {
+                 move_piece('down')
+             }
+             elsif($key == SDLK_UP) {
+                 $curr_tile->[0] = rotate_piece($curr_tile->[0]);
+             }
+         }
+     }
+ }

+ $app->add_event_handler( \&trigger_move_event_handler );
```

7.6.1 Score and Game State

Next we add the move handler to update the game state. In tetris the game state can be summarized as the grid, current piece and the score. In this move handler we update all these things .

```
+ $app->add_move_handler( sub {
+     my ( $step, $app ) = @_;
```

We update the current piece’s state as movable or fixed.

```
+     if(can_move_piece('down', $step / 2)) {
+         move_piece('down', $step / 2);
+     }
+     else {
+         store_piece($curr_tile); # placing the tile
+
```

We update the status of the grid and see if there are lines to remove. + # checking for lines to delete + my $y; + my @to_delete = (); + for($y = 22; $y >= 0; $y–) { + # there is no space if min of this row is true (greater than zero) + if(min(@{$store}[($y*10)..(((($y+1)*10)-1)])) { + push(@to_delete, $y); + } + }

When we delete lines increment the score of the user.

```
+         # deleting lines
+         foreach(@to_delete) {
+             splice(@{$store}, $_*10, 10);
+             $score++;
+         }
+
 for each deleted line we clear the grid.
+         # adding blank rows to the top
+         foreach(@to_delete) {
+             splice(@{$store}, 0, 0, (0,0,0,0,0,0,0,0,0,0));
+         }
+
lly we lauch a new current tile if needed.
+         # launching new tile
+         @{$curr_tile->[0]}  = @{$pieces{$next_tile}};
+         $curr_tile->[1]     = 4;
+         $curr_tile->[2]     = 0;
+         $next_tile          = shuffle(keys %pieces);
+     }
+ });
```

7.6.2 Showing the Game

In the show handler we iterate through each element in the store and grid array and place the right colored tile where needed (using the numbers).

```
+ # renders game objects on the screen
+ $app->add_show_handler(
```

```
+     sub {
+         # first, we clear the screen
+         $app->draw_rect( [ 0, 0, $app->w, $app->h ], 0x000000 );
+         # and draw the background image
+         $back->blit( $app );
+         my $x = 0;
+         my $y = 0;
+         # draw the not moving tiles
+         foreach(@{$store}) {
+             $piece[$_]->blit( $app,
+                               undef,
+                              [ 28 + $x%10 * 20, 28 + $y * 20 ]
+                             ) if $_;
+             $x++;
+             $y++ unless $x % 10;
+         }
+         $x = 0;
+         $y = 0;
+         # draw the moving tile
+         foreach(@{$grid}) {
+             $piece[$_]->blit( $app, undef, [ 28 + $x%10 * 20, 28 + $y * 20 ] ) if $_;
+             $x++;
+             $y++ unless $x % 10;
+         }
+         # the next tile will be...
+         my $next_tile_index = max(@{$pieces{$next_tile}});
+         for $y (0..3) {
+             for $x (0..3) {
+                 if($pieces{$next_tile}->[$x + 4 * $y]) {
+                     $piece[$next_tile_index]->blit( $app, undef,
+                                                     [ 264 + $x * 20, 48 + $y * 20 ]
+                                                    );
+                 }
+             }
+         }
```

Lastly we draw texts needed.

```
+         $score_text->write_xy( $app, 248,  20, "Next Piece" );
+         $score_text->write_xy( $app, 248, 240, "Score: $score" );
+         # finally, we update the screen
+         $app->update;
+     }
+ );

+ # all is set, run the app!
+ $app->run();
```

7.7 Author

Code for this chapter was provided by Tobias Leich “FROGGS”.

8

Puzz! A puzzle game

8.1 Abstract

We are now ready to write another complete game. Instead of listing the code and then explaining it, I will go through the process of how I might write it.

Puzz is a simple rearrangment puzzle. A random image from the folder Puzz is in is chosen and broken into a 4x4 grid. The top left corner piece is then taken away, and every other piece is then moved to a random position, scrambling the image up. The goal is then to move pieces which are in the 4 squares adjacent to the empty square on to the empty square, and eventually restore the image.

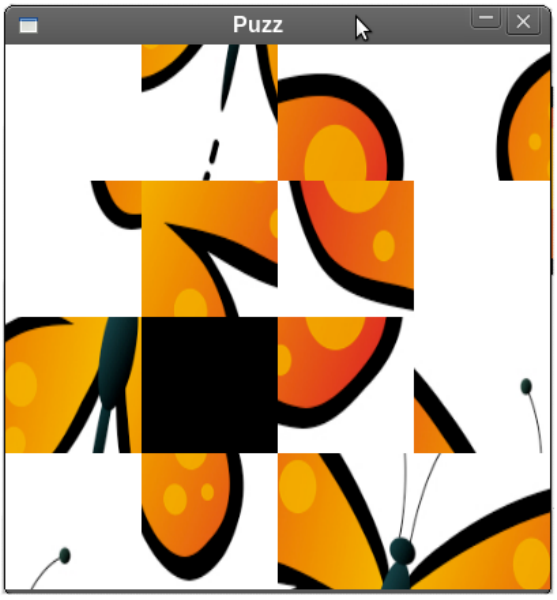

Figure 8.1: Credits to Sebastian Riedel (kraih.com) for the Perl6 logo used with permission in the application.

8.2 The Window

So, first thing we do is create the window. I've decided I want each piece to be 100x100, so the window needs to be 400x400.

```
use strict;
use warnings;

use SDL;
use SDLx::App;

my $App = SDLx::App->new(w => 400, h => 400, t => 'Puzz');
```

Next thing we usually do is figure out what global vars we will be needing. As with $App, I like to name my globals with title case, so they are easily distinguishable from lexical vars. The globals we need are the grid (the positions of the pieces), the images we have to use, the current image, and a construct that will give us piece movement, along with an animation.

```
my @Grid;
my @Img;
my $CurrentImg;
my %Move;
```

For now, lets fill in @Grid with what it's going to look like:

```
@Grid = (
   [0,  1,  2,  3],
   [4,  5,  6,  7],
   [8,  9,  10, 11],
   [12, 13, 14, 15],
);
```

`0` will be our blank piece, but we could have chosen it to be any other number. When the grid looks like this, it's solved, so eventually we will need a way to scramble it. It's good enough for now, though.

8.3 Loading the images

To load the images, we would normally use `SDLx::Surface`, but we're going to do it the libsdl way with `SDL::Image` because we need to do our own error handling.

```
use SDL::Image;
use SDL::GFX::Rotozoom 'SMOOTHING_ON';

while(<./*>) {
```

```
        if(-f and my $i = SDL::Image::load($_)) {
            $i = SDL::GFX::Rotozoom::surface_xy($i, 0, 400 / $i->w, 400 / $i->h, SMOOTHING_ON);
            push @Img, $i;
        }
        else
        {
            warn "Cannot Load $_: " . SDL::get_error() if $_ =~ /jpg|png|bmp/;
        }
    }
    $CurrentImg = $Img[rand @Img];

    die "Please place images in the Current Folder" if $#Img < 0;
```

We just go through every file in the current directory, and try to load it as an image. `SDL::Image::load` will return false if there was an error, so we want to discard it when that happens. If we used `SDLx::Surface` to load the images, we would get a warning every time a file fails to load as an image, which we don't want. The `my $i = SDL::Image::load($_)` is just an idiom for setting a var and checking it for truth at the same time.

We want the image to be 400x400, and `SDL::GFX::Rotozoom` makes this possible. The two Rotozoom functions that are the most useful are `surface` and `surface_xy`. They work like this:

```
$zoomed_src = SDL::GFX::Rotozoom::surface($src, $angle, $zoom, $smoothing)
$zoomed_src = SDL::GFX::Rotozoom::surface_xy($src, $angle, $x_zoom, $y_zoom, $smoothing)
```

The zoom values are the multiplier for that component, or for both components at once as with `$zoom`. `$angle` is an angle of rotation in degrees. `$smoothing` should be `SMOOTHING_ON` or `SMOOTHING_OFF` (which can be exported by `SDL::GFX::Rotozoom`) or just 1 or 0.

Once the image is zoomed, it is added to the image array. The current image is then set to a random value of the array.

8.4 Handling Events

The next part I like to write is the events. We're going to make Escape quit, and left click will move the pieces around. We `use SDL::Events` for the constants.

```
use SDL::Events;

sub on_event {
    my ($e) = @_;
    if($e->type == SDL_QUIT or $e->type == SDL_KEYDOWN and $e->key_sym == SDLK_ESCAPE) {
        $App->stop;
    }
    elsif($e->type == SDL_MOUSEBUTTONDOWN and $e->button_button == SDL_BUTTON_LEFT) {
        ...
    }
}

$App->add_event_handler(\&on_event);
# $App->add_move_handler(\&on_move);
# $App->add_show_handler(\&on_show);
$App->run;
```

8.5 Filling the Grid

Once we have something like this, it's a good time to put some `warn` messages in to make sure the inputs are working correctly. Once they are, it's time to fill it in.

```
my $x = int($e->button_x / 100);
my $y = int($e->button_y / 100);
if(!%Move and $Grid[$y][$x]) {`
    ...
}
```

From the pixel coordinates of the click (0 to 399), we want to find out the grid coordinates (0 to 3), so we divide both components by 100 and round them down. Then, we only want to continue on to see if that piece can move if no other piece is moving (`%Move` is false), and the piece clicked isn't the blank piece (0).

```
for([-1, 0], [0, -1], [1, 0], [0, 1]) {
    my $nx = $x + $_->[0];
    my $ny = $y + $_->[1];
    if($nx >= 0 and $nx < 4 and $ny >= 0 and $ny < 4 and !$Grid[$ny][$nx]) {
        ...
    }
}
```

8.6 Moving the Pieces

We check that the blank piece is in the 4 surrounding places by constructing 4 vectors. These will take us to those squares. The `x` component is first and the second is `y`. We iterate through them, setting `$nx` and `$ny` to the new position. Then if both `$nx` and `$ny` are within the grid (0 to 3), and that position in the grid is 0, we can move the piece to the blank square.

```
%Move = (
    x      => $x,
    y      => $y,
    x_dir  => $_->[0],
    y_dir  => $_->[1],
    offset => 0,
);
```

To make a piece move, we construct the move hash with all the information it needs to move the piece. The `x` and `y` positions of the piece, the `x` and `y` directions it will be moving (the vector), and it's current pixel offset from it's position (for the moving animation), which starts at 0.

8.6.1 The Move Handler Callback

Next we will write the move handler. All it needs to do is move any moving piece along by updating the offset, and click it in to where it's being moved to when it has moved the whole way (offset is 100 or more).

```
sub on_move {
    if(%Move) {
        $Move{offset} += 30 * $_[0];
        if($Move{offset} >= 100) {
            $Grid[$Move{y} + $Move{y_dir}][$Move{x} + $Move{x_dir}] = $Grid[$Move{y}][$Move{x}];
            $Grid[$Move{y}][$Move{x}] = 0;
            undef %Move;
        }
    }
}
```

30 has been arbitrarily chosen as the speed of the move, as it felt the best after a little playing and tweaking. Always remember to multiply things like this by the step value in `$_[0]` so that the animation moves in correct time with the updating.

Once the offset is 100 or more, the grid place that the piece is moving to is set to the value of the piece, and the piece is set to the blank value. The move is then finished, so `%Move` is deleted.

8.7 Rendering the Game

Now that we have all the functionality we need it's finally time to see the game.

```
sub on_show {
    $App->draw_rect( [0,0,$App->w,$App->h], 0 );
    for my $y (0..3) {
```

```
            for my $x (0..3) {
                ...
            }
        }
        $App->flip;
    }
```

We start the show handler by drawing a black rect over the entire app. Entire surface and black are the defaults of `draw_rect`, so letting it use the defaults is good. Next we iterate through a `y` and `x` of 0 to 3 so that we can go through each piece of the grid. At the end of the handler we update the app with a call to `flip`.

```
next unless my $val = $Grid[$y][$x];
my $xval = $val % 4;
my $yval = int($val / 4);
my $move = %Move && $Move{x} == $x && $Move{y} == $y;
...
```

Inside the two loops we put this. First we set `$val` to the grid value at the current position, and we skip to the next piece if it's the blank piece. We have the `x` and `y` coordinates of where that piece is on the board, but we need to figure out where it is on the image. If you refer back to the initialisation of the grid, the two operations to find the values should make sense. `$move` is set with a bool of whether it is this piece that is moving, if there is a piece moving at all.

```
$App->blit_by(
    $CurrentImg,
    [$xval * 100, $yval * 100, 100, 100],
    [$x * 100 + ($move ? $Move{offset} * $Move{x_dir} : 0),
     $y * 100 + ($move ? $Move{offset} * $Move{y_dir} : 0)]
);
```

Now that we have all of this, we can blit the portion of the current image we need to the app. We use `blit_by` because the image we're blitting isn't an SDLx::Surface (because we didn't load it as one), but the app is. Here's how `blit_by` works as opposed to `blit`:

```
$src->blit($dest, $src_rect, $dest_rect)
$dest->blit_by($src, $src_rect, $dest_rect)
```

The portion we need is from the `$xval` and `$yval`, and where it needs to go to is from `$x` and `$y`. All are multiplied by 100 because we're dealing with 0 to 300, not 0 to 3. If the piece is moving, the offset multiplied by the diretion is added to the position.

When the code is run with all 3 handlers, we have a fully working game. The pieces move around nicely when clicked. The only things it still needs are a shuffled grid and a way to check if the player has won. To imlement these two things, we will make two more functions.

```
use List::Util 'shuffle';

sub new_grid {
    my @new = shuffle(0..15);
    @Grid = map { [@new[ $_*4..$_*4+3 ]] } 0..3;
    $CurrentImg = $Img[rand @Img];
}
```

We will replace the grid initialising we did with this sub. First it shffles the numbers 0 through 15 with `List::Util::shuffle`. This array is then arranged into a 2D grid with a `map` and put in to @Grid. Setting the current image is also put into this sub.

```
sub won {
    my $correct = 0;
    for(@Grid) {
        for(@$_) {
            return 0 if $correct != $_;
            $correct++;
        }
    }
    return 1;
}
```

This sub returns whether the grid is in the winning configuration, that is, all piece values are in order from 0 to 15.

Now we put a call to `new_grid` to replace the grid initialisation we had before. We put `won` into the event handler to make click call `new_grid` if you have won. Finally, `won` is put into the show handler to show the blank piece if you have won.

8.8 Complete Code

Here is the finished code:

```
use strict;
use warnings;

use SDL;
use SDLx::App;
use SDL::Events;
use SDL::Image;
use SDL::GFX::Rotozoom 'SMOOTHING_ON';
use List::Util 'shuffle';

my $App = SDLx::App->new(w => 400, h => 400, t => 'Puzz');

my @Grid;
my @Img;
my $CurrentImg;
my %Move;

while(<./*>) {
    if(-f and my $i = SDL::Image::load($_)) {
        $i = SDL::GFX::Rotozoom::surface_xy($i, 0, 400 / $i->w, 400 / $i->h, SMOOTHING_ON);
        push @Img, $i;
    }
    else
```

```
    {
        warn "Cannot Load $_: " . SDL::get_error() if $_ =~ /jpg|png|bmp/;
    }

}

die "Please place images in the Current Folder" if $#Img < 0;

new_grid();

sub on_event {
    my ($e) = @_;
    if($e->type == SDL_QUIT or $e->type == SDL_KEYDOWN and $e->key_sym == SDLK_ESCAPE) {
        $App->stop;
    }
    elsif($e->type == SDL_MOUSEBUTTONDOWN and $e->button_button == SDL_BUTTON_LEFT) {
        my($x, $y) = map { int($_ / 100) } $e->button_x, $e->button_y;
        if(won()) {
            new_grid();
        }
        elsif(!%Move and $Grid[$y][$x]) {
            for([-1, 0], [0, -1], [1, 0], [0, 1]) {
                my($nx, $ny) = ($x + $_->[0], $y + $_->[1]);
                if($nx >= 0 and $nx < 4 and $ny >= 0 and $ny < 4 and !$Grid[$ny][$nx]) {
                    %Move = (
                        x      => $x,
                        y      => $y,
                        x_dir  => $_->[0],
                        y_dir  => $_->[1],
                        offset => 0,
                    );
                }
            }
        }
    }
}

sub on_move {
```

```
62      if(%Move) {
63          $Move{offset} += 30 * $_[0];
64          if($Move{offset} >= 100) {
65              $Grid[$Move{y} + $Move{y_dir}][$Move{x} + $Move{x_dir}] = $Grid[$Move{y}][$Move{x}]
66              $Grid[$Move{y}][$Move{x}] = 0;
67              undef %Move;
68          }
69      }
70  }
71
72  sub on_show {
73      $App->draw_rect( [0,0,$App->w,$App->h], 0 );
74      for my $y (0..3) {
75          for my $x (0..3) {
76              next if not my $val = $Grid[$y][$x] and !won();
77              my $xval = $val % 4;
78              my $yval = int($val / 4);
79              my $move = %Move && $Move{x} == $x && $Move{y} == $y;
80              $App->blit_by(
81                  $CurrentImg,
82                  [$xval * 100, $yval * 100, 100, 100],
83                  [$x * 100 + ($move ? $Move{offset} * $Move{x_dir} : 0),
84                   $y * 100 + ($move ? $Move{offset} * $Move{y_dir} : 0)]
85              );
86          }
87      }
88      $App->flip;
89  }
90
91  sub new_grid {
92      my @new = shuffle(0..15);
93      @Grid = map { [@new[ $_*4..$_*4+3 ]] } 0..3;
94      $CurrentImg = $Img[rand @Img];
95  }
96
97  sub won {
98      my $correct = 0;
99      for(@Grid) {
```

```
100             for(@$_) {
101                 return 0 if $correct != $_;
102                 $correct++;
103             }
104         }
105         return 1;
106     }
107
108     $App->add_event_handler(\&on_event);
109     $App->add_move_handler(\&on_move);
110     $App->add_show_handler(\&on_show);
111     $App->run;
```

You now hopefully know more of the process that goes in to creating a simple game. The process of creating a complex game is similar, it just requires more careful planning. You should have also picked up a few other tricks, like with `SDL::GFX::Rotozoom`, `SDL::Image::load` and `blit_by`.

8.9 Activities

1. Make the blank piece the bottom right piece instead of the top left piece.

2. Make the grid dimensions variable by getting the value from `$ARGV[0]`. The grid will then be 5x5 if `$ARGV[0]` is 5 and so on.

8.10 Author

This chapter's content graciously provided by Blaizer.

9

Sound and Music

Sound and Music in SDL are handled by the `Audio` and `SDL_Mixer` components. Enabling `Audio` devices is provided with the Core SDL Library and only supports wav files. `SDL_Mixer` supports more audio file formats and has additional features that we need for sound in Game Development.

Similarly to video in SDL, there are several way for perl developers to access the Sound components of SDL. For the plain `Audio` component the `SDL::Audio` and related modules are available. `SDL_Mixer` is supported with th `SDL::Mixer` module. There is currently a `SDLx::Sound` module in the work, but not completed at the time of writing this manual. For that reason this chapter will use `SDL::Audio` and `SDL::Mixer`.

9.1 Simple Sound Script

To begin using sound we must enable and open an audiospec:

```
se strict;
se warnings;
se SDL;
se Carp;
se SDL::Audio;
se SDL::Mixer;

DL::init(SDL_INIT_AUDIO);

nless( SDL::Mixer::open_audio( 44100, AUDIO_S16SYS, 2, 4096 ) == 0 )

  Carp::croak "Cannot open audio: ".SDL::get_error();
```

`open_audio` will open an audio device with frequency at 44100 Mhz, audio format AUDIO_S16SYS (Note: This is currently the most portable format, however there are others), 2 channels and a chunk size of 4096. Fiddle with these values if you are comfortable with sound terminology and techniques.

9.1.1 Loading Samples

Next we will load sound samples that generally used for sound effects and the like. Currently `SDL_Mixer` reserves samples for `.WAV`, `.AIFF`, `.RIFF .OGG`, and `.VOC` formats.

Samples run on one of the 2 channels that we opened up, while the other channel will be reserved for multiple plays of the sample. To load samples we will be doing the following:

```
use SDL::Mixer::Samples;

#Brillant Lazer Sound from HTTP://FreeSound.Org/samplesViewSingle.php?id=30935
my $sample = SDL::Mixer::Samples::load_WAV('data/sample.wav');

unless($sample)
{
   Carp::croak "Cannot load file data/sample.wav: ".SDL::get_error();
}
```

9.1.2 Playing the sample and closing audio

Now we can play that sample on any open channel looping forever:

```
use SDL::Mixer::Samples;
use SDL::Mixer::Channels;

y $sample =  SDL::Mixer::Samples::load_WAV('data/sample.wav');
nless( $sample)

  Carp::croak "Cannot load file data/sample.wav: ".SDL::get_error();


my $playing_channel = SDL::Mixer::Channels::play_channel( -1, $sample, 0 );
```

`play_channel` allows us to assign a sample to the channel `-1` which indicates any open channel. `0` indicates we want to play the sample only once.

Note that since the sound will be playing in an external process we will need to keep the perl script running. In a game this is no problem but for a single script like this we can just use a simple `sleep` function. Once we are done we can go ahead and close the audio device.

```
sleep(1);
SDL::Mixer::close_audio();
```

9.1.3 Streaming Music

Next we will use `SDL::Mixer::Music` to add a background music to our script here.

```
use SDL::Mixer::Channels;
+use SDL::Mixer::Music;

+#Load our awesome music from HTTP://8BitCollective.Com
+my $background_music =
    +         SDL::Mixer::Music::load_MUS('data/music/01-PC-Speaker-Sorrow.ogg');


+unless( $background_music )
+{
+   Carp::croak "Cannot load music file data/music/01-PC-Speaker-Sorrow.ogg: ".SDL::get_error()
+}
```

Music types in `SDL::Mixer` run in a separate channel from our samples which allows us to have sound effects (like jump, or lasers etc) to play at the same time.

```
SDL::Mixer::Music::play_music($background_music,0);
```

`play_music` also takes a parameter for how many loops you would like to play the song for, where 0 is 1.

To stop the music we can call `halt_music`.

```
sleep(2);
SDL::Mixer::Music::halt_music();
SDL::Mixer::close_audio();
```

Controlling Volume can be as simple as:

```
    All channels indicated by the -1
    DL::Mixer::Channels::volume(-1,10);

    Specifically for the Music
    DL::Mixer::Music::volume_music( 10 );
```

Volumes can be set at anytime and range from `1-100`.

9.1.4 Code so far

```
use strict;
use warnings;
use SDL;
use Carp;
use SDL::Audio;
use SDL::Mixer;
use SDL::Mixer::Samples;
use SDL::Mixer::Channels;
use SDL::Mixer::Music;
SDL::init(SDL_INIT_AUDIO);

unless( SDL::Mixer::open_audio( 44100, AUDIO_S16SYS, 2, 4096 ) == 0 )
{
    Carp::croak "Cannot open audio: ".SDL::get_error();
}


my $sample = SDL::Mixer::Samples::load_WAV('data/sample.wav');

unless( $sample)
{
    Carp::croak "Cannot load file data/sample.wav: ".SDL::get_error();
}

```

```
25 my $playing_channel = SDL::Mixer::Channels::play_channel( -1, $sample, 0 );
26
27 #Load our awesome music from HTTP://8BitCollective.Com
28 my $background_music = SDL::Mixer::Music::load_MUS('data/music/01-PC-Speaker-Sorrow.ogg');
29
30 unless( $background_music )
31 {
32     Carp::croak "Cannot load music file data/music/01-PC-Speaker-Sorrow.ogg: "
33             .SDL::get_error();
34 }
35
36 SDL::Mixer::Music::play_music( $background_music,0 );
37
38 sleep(2);
39
40 SDL::Mixer::Music::halt_music();
41 SDL::Mixer::close_audio;
```

9.2 Sound Applications

Now that we know how to prepare and play simple sounds we will apply it to an `SDLx::App`.

9.2.1 SDLx::App Audio Initialization

`SDLx::App` will initialize everything normally for us. However for a stream line application it is recommend to initialize only the things we need. In this case that is `SDL_INIT_VIDEO` and `SDL_INIT_AUDIO`.

```
use strict;
use warnings;
use SDL;
```

```
use Carp;
use SDLx::App;
use SDL::Audio;
use SDL::Mixer;
use SDL::Event;
use SDL::Events;
use SDL::Mixer::Music;
use SDL::Mixer::Samples;
use SDL::Mixer::Channels;

my $app = SDLx::App->new(
    init  => SDL_INIT_AUDIO | SDL_INIT_VIDEO,
    width => 250,
    height => 75,
    title => "Sound Event Demo",
    eoq   => 1
```

9.2.2 Loading Resources

It is highly recommended to perform all resource allocations before a `SDLx::App::run()` method is called.

```
# Initialize the Audio
unless ( SDL::Mixer::open_audio( 44100, AUDIO_S16SYS, 2, 4096 ) == 0 ) {
    Carp::croak "Cannot open audio: " . SDL::get_error();
}

#Something to show while we play music and sounds
my $channel_volume = 100;
my $music_volume   = 100;
my $laser_status   = 'none';
my $music_status   = 'not playing';

# Load our sound resources
```

```
my $laser = SDL::Mixer::Samples::load_WAV('data/sample.wav');
unless ($laser) {
    Carp::croak "Cannot load sound: " . SDL::get_error();
}

my $background_music =
SDL::Mixer::Music::load_MUS('data/music/01-PC-Speaker-Sorrow.ogg');
unless ($background_music) {
    Carp::croak "Cannot load music: " . SDL::get_error();
}
```

9.2.3 The Show Handler

For the purposes of describing the current state of the music lets draw text to the screen in a `show_handler`.

```
$app->add_show_handler(
sub {

    $app->draw_rect([0,0,$app->w,$app->h], 0 );

    $app->draw_gfx_text( [10,10], [255,0,0,255], "Channel Volume : $channel_volume" );
    $app->draw_gfx_text( [10,25], [255,0,0,255], "Music Volume   : $music_volume" );
    $app->draw_gfx_text( [10,40], [255,0,0,255], "Laser Status   : $laser_status" );
    $app->draw_gfx_text( [10,55], [255,0,0,255], "Music Status   : $music_status" );

    $app->update();

}
);
```

This will draw the channel volume of our samples, and the volume of the music. It will also print the status of our two sounds in the application.

9.2.4 The Event Handler

Finally our event handler will do the actual leg work and trigger the music and sound as we need it.

```
$app->add_event_handler(
    sub {
        my $event = shift;

        if ( $event->type == SDL_KEYDOWN ) {
            my $keysym  = $event->key_sym;
            my $keyname = SDL::Events::get_key_name($keysym);

            if ( $keyname eq 'space' ) {

                $laser_status = 'PEW!';
                #fire lasers!
                SDL::Mixer::Channels::play_channel( -1, $laser, 0 );

            }
            elsif ( $keyname eq 'up' ) {
                $channel_volume += 5 unless $channel_volume == 100;
            }
            elsif ( $keyname eq 'down' ) {
                $channel_volume -= 5 unless $channel_volume == 0;
            }
            elsif ( $keyname eq 'right' ) {
                $music_volume += 5 unless $music_volume == 100;
            }
            elsif ( $keyname eq 'left' ) {
                $music_volume -= 5 unless $music_volume == 0;
            }
            elsif ( $keyname eq 'return' ) {
                my $playing = SDL::Mixer::Music::playing_music();
                my $paused  = SDL::Mixer::Music::paused_music();

                if ( $playing == 0 && $paused == 0 ) {
```

```
                        SDL::Mixer::Music::play_music( $background_music, 1 );
                        $music_status = 'playing';
                    }
                    elsif ( $playing && !$paused ) {
                        SDL::Mixer::Music::pause_music();
                        $music_status = 'paused'
                    }
                    elsif ( $playing && $paused ) {
                        SDL::Mixer::Music::resume_music();
                        $music_status = 'resumed playing';
                    }

                }

                SDL::Mixer::Channels::volume( -1, $channel_volume );
                SDL::Mixer::Music::volume_music($music_volume);

            }

        }

);
```

The above event handler fires the laser on pressing the 'Space' key. Go ahead and press it multiple times as if you are firing a gun in a game! You will notice that depending on how fast you fire the laser the application will still manage to overlap the sounds as needed. The sample overlapping is accomplished by requiring multiple channels in the `open_audio` call. If your game has lots of samples that may play at the same time you may need more channels allocated. Additionally you can see that the volume control is easily managed both on the channels and the music with just incrementing or decrementing a value and calling the appropriate function.

Finally it is worth noticing the various state the background music can be in.

Lets run this application and the make sure to clean up the audio on the way out. $app->run(); SDL::Mixer::Music::halt_music(); SDL::Mixer::close_audio;

9.2.5 Completed Code

```
use strict;
use warnings;

use Cwd;
use Carp;
use File::Spec;

use threads;
use threads::shared;

use SDL;
use SDL::Event;
use SDL::Events;

use SDL::Audio;
use SDL::Mixer;
use SDL::Mixer::Music;
use SDL::Mixer::Effects;

use SDLx::App;
my $app = SDLx::App->new(
    init   => SDL_INIT_AUDIO | SDL_INIT_VIDEO,
    width  => 800,
    height => 600,
    depth  => 32,
    title  => "Music Visualizer",
    eoq    => 1,
    dt     => 0.2,
);

# Initialize the Audio
unless ( SDL::Mixer::open_audio( 44100, AUDIO_S16, 2, 1024 ) == 0 ) {
    Carp::croak "Cannot open audio: " . SDL::get_error();
}

```

```
36  # Load our music files
37  my $data_dir = '.';
38  my @songs    = glob 'data/music/*.ogg';
39
40  my @stream_data : shared;
41
42  #  Music Effect to pull Stream Data
43  sub music_data {
44      my ( $channel, $samples, $position, @stream ) = @_;
45
46      {
47          lock(@stream_data);
48          push @stream_data, @stream;
49      }
50
51      return @stream;
52  }
53
54  sub done_music_data { }
55
56  my $music_data_effect_id =
57    SDL::Mixer::Effects::register( MIX_CHANNEL_POST, "main::music_data",
58      "main::done_music_data", 0 );
59
60  #  Music Playing Callbacks
61  my $current_song = 0;
62  my $lines = $ARGV[0] || 50;
63
64  my $current_music_callback = sub {
65      my ( $delta, $app ) = @_;
66
67      $app->draw_rect( [ 0, 0, $app->w(), $app->h() ], 0x000000FF );
68      $app->draw_gfx_text(
69          [ 5, $app->h() - 10 ],
70          [ 255, 0, 0, 255 ],
71          "Playing Song: " . $songs[ $current_song - 1 ]
72      );
73
```

```
74      my @stream;
75      {
76          lock @stream_data;
77          @stream      = @stream_data;
78          @stream_data = ();
79      }
80
81      # To show the right amount of lines we choose a cut of the stream
82      # this is purely for asthetic reasons.
83
84      my $cut = @stream / $lines;
85
86      # The width of each line is calculated to use.
87      my $l_wdt = ( $app->w() / $lines ) / 2;
88
89      for ( my $i = 0 ; $i < $#stream ; $i += $cut ) {
90
91          #  In stereo mode the stream is split between two alternating streams
92          my $left  = $stream[$i];
93          my $right = $stream[ $i + 1 ];
94
95          #  For each bar we calculate a Y point and a X point
96          my $point_y = ( ( ($left) ) * $app->h() / 4 / 32000 ) + ( $app->h / 2 );
97          my $point_y_r =
98            ( ( ($right) ) * $app->h() / 4 / 32000 ) + ( $app->h / 2 );
99          my $point_x = ( $i / @stream ) * $app->w;
100
101         # Using the parameters
102         #   Surface, box coordinates and color as RGBA
103         SDL::GFX::Primitives::box_RGBA(
104             $app,
105             $point_x - $l_wdt,
106             $app->h() / 2,
107             $point_x + $l_wdt,
108             $point_y, 40, 0, 255, 128
109         );
110         SDL::GFX::Primitives::box_RGBA(
111             $app,
```

```
112             $point_x - $l_wdt,
113             $app->h() / 2,
114             $point_x + $l_wdt,
115             $point_y_r, 255, 0, 40, 128
116         );
117
118     }
119
120     $app->flip();
121
122 };
123
124 my $cms_move_callback_id;
125 my $pns_move_callback_id;
126 my $play_next_song_callback;
127
128 sub music_finished_playing {
129     SDL::Mixer::Music::halt_music();
130
131     $pns_move_callback_id = $app->add_move_handler($play_next_song_callback)
132       if ( defined $play_next_song_callback );
133
134 }
135
136 $play_next_song_callback = sub {
137     return $app->stop() if $current_song >= @songs;
138     my $song = SDL::Mixer::Music::load_MUS( $songs[ $current_song++ ] );
139     SDL::Mixer::Music::play_music( $song, 0 );
140
141     $app->remove_move_handler($pns_move_callback_id)
142       if defined $pns_move_callback_id;
143 };
144
145 $app->add_show_handler($current_music_callback);
146 $pns_move_callback_id = $app->add_move_handler($play_next_song_callback);
147
148 $app->add_move_handler(
149     sub {
```

```
150         my $music_playing = SDL::Mixer::Music::playing_music();
151
152         music_finished_playing() unless $music_playing;
153
154     }
155 );
156
157 $app->add_event_handler(
158     sub {
159         my ( $event, $app ) = @_;
160         if ( $event->type == SDL_KEYDOWN && $event->key_sym == SDLK_DOWN ) {
161
162             # Indicate that we are done playing the music_finished_playing
163             music_finished_playing();
164         }
165     }
166 );
167
168 $app->run();
169
170 SDL::Mixer::Effects::unregister( MIX_CHANNEL_POST, $music_data_effect_id );
171 SDL::Mixer::Music::hook_music_finished();
172 SDL::Mixer::Music::halt_music();
173 SDL::Mixer::close_audio();
```

9.3 Music Visualizer

The music visualizer example processes real-time sound data–data as it plays–and displays the wave form on the screen. It will look something like:

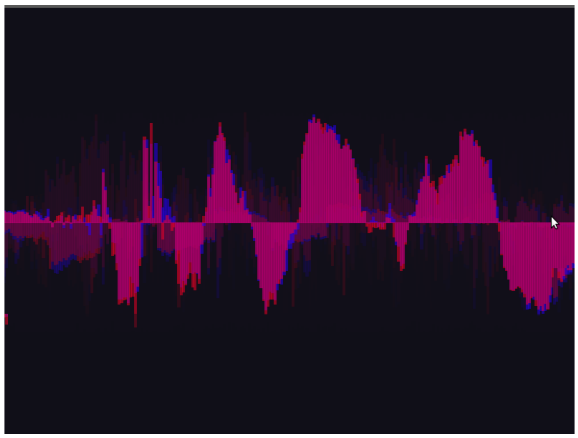

Figure 9.1: Simple Music Visualization

9.3.1 The Code and Comments

The program begins with the usual boilerplate of an SDL Perl application:

```
use strict;
use warnings;

use Cwd;
use Carp;
use File::Spec;

use threads;
use threads::shared;

use SDL;
use SDL::Event;
use SDL::Events;

use SDL::Audio;
use SDL::Mixer;
use SDL::Mixer::Music;
```

```
use SDL::Mixer::Effects;

use SDLx::App;
```

It then creates an application with both audio and video support:

```
my $app = SDLx::App->new(
        init   => SDL_INIT_AUDIO | SDL_INIT_VIDEO,
        width  => 800,
        height => 600,
        depth  => 32,
        title  => "Sound Event Demo",
        eoq    => 1,
        dt     => 0.2,
);
```

The application must initialize the audio system with a format matching the expected audio input. `AUDIO_S16` provides a 16-bit signed integer array for the stream data:

```
# Initialize the Audio
unless ( SDL::Mixer::open_audio( 44100, AUDIO_S16, 2, 1024 ) == 0 ) {
        Carp::croak "Cannot open audio: " . SDL::get_error();
}
```

The music player needs the music files from the *data/music/* directory:

```
# Load our music files
my $data_dir = '.';
my @songs = glob 'data/music/*.ogg';
```

A music effect reads stream data, then serializes it to share between threads:

```
my @stream_data : shared;

#  Music Effect to pull Stream Data
sub music_data {
```

```
    my ( $channel, $samples, $position, @stream ) = @_;

    {
        lock(@stream_data);
        push @stream_data, @stream;
    }

    return @stream;
}

sub done_music_data { }
```

... and that effect gets registered as a callback with `SDL::Mixer::Effects`:

```
    my $music_data_effect_id =
              SDL::Mixer::Effects::register( MIX_CHANNEL_POST, "main::music_data",
                    "main::done_music_data", 0 );
```

The program's single command-line option governs the number of lines to display in the visualizer. The default is 50.

```
my $lines = $ARGV[0] || 50;
```

The drawing callback for the `SDLx::App` runs while a song plays. It reads the stream data and displays it on the screen as a wave form. The math behind calculating the graphics to display is more detail than this article intends, but the graphic code is straightforward:

```
#  Music Playing Callbacks
my $current_song = 0;

my $current_music_callback = sub {
    my ( $delta, $app ) = @_;

    $app->draw_rect( [ 0, 0, $app->w(), $app->h() ], 0x000000FF );
    $app->draw_gfx_text(
        [ 5, $app->h() - 10 ],
```

```
    [ 255, 0, 0, 255 ],
    "Playing Song: " . $songs[ $current_song - 1 ]
);

my @stream;
{
    lock @stream_data;
    @stream      = @stream_data;
    @stream_data = ();
}

# To show the right amount of lines we choose a cut of the stream
# this is purely for asthetic reasons.

my $cut = @stream / $lines;

# The width of each line is calculated to use.
my $l_wdt = ( $app->w() / $lines ) / 2;

for ( my $i = 0 ; $i < $#stream ; $i += $cut ) {

    #  In stereo mode the stream is split between two alternating streams
    my $left  = $stream[$i];
    my $right = $stream[ $i + 1 ];

    #  For each bar we calculate a Y point and a X point
    my $point_y = ( ( ($left) ) * $app->h() / 4 / 32000 ) + ( $app->h / 2 );
    my $point_y_r =
      ( ( ($right) ) * $app->h() / 4 / 32000 ) + ( $app->h / 2 );
    my $point_x = ( $i / @stream ) * $app->w;

    # Using the parameters
    #   Surface, box coordinates and color as RGBA
    SDL::GFX::Primitives::box_RGBA(
        $app,
        $point_x - $l_wdt,
        $app->h() / 2,
        $point_x + $l_wdt,
```

```
            $point_y, 40, 0, 255, 128
        );
        SDL::GFX::Primitives::box_RGBA(
            $app,
            $point_x - $l_wdt,
            $app->h() / 2,
            $point_x + $l_wdt,
            $point_y_r, 255, 0, 40, 128
        );

    }

    $app->flip();

};
```

Whenever a song finishes `SDL::Mixer::Music::playing_music` returns `0`. We detect this change in state and call `music_finished_playing()` where the program attaches our `$play_next_song_callb` callback to switch to the next song gracefully:

```
my $cms_move_callback_id;
my $pns_move_callback_id;
my $play_next_song_callback;

sub music_finished_playing {
    SDL::Mixer::Music::halt_music();
    $pns_move_callback_id = $app->add_move_handler($play_next_song_callback)
      if ( defined $play_next_song_callback );

}

$play_next_song_callback = sub {
    return $app->stop() if $current_song >= @songs;
    my $song = SDL::Mixer::Music::load_MUS( $songs[ $current_song++ ] );
    SDL::Mixer::Music::play_music( $song, 0 );

    $app->remove_move_handler($pns_move_callback_id)
```

```
      if defined $pns_move_callback_id;
};
```

A move handler is attached to detect if music is playing or not:

```
$app->add_move_handler(
    sub {
        my $music_playing = SDL::Mixer::Music::playing_music();
        music_finished_playing() unless $music_playing;
    }
)
```

The first callback to trigger the `$play_next_song_callback` gets the first song:

```
    $app->add_show_handler($current_music_callback);
$pns_move_callback_id = $app->add_move_handler($play_next_song_callback);
```

... and a keyboard event handler for a keypress allows the user to move through songs:

```
    $app->add_event_handler(
    sub {
            my ($event, $app) = @_;

            if( $event->type == SDL_KEYDOWN && $event->key_sym == SDLK_DOWN)
            {
                    #Indicate that we are done playing the music_finished_playing
                    music_finished_playing();
            }

    }
    );
```

From there, the application is ready to run:

```
    $app->run();
```

... and the final code gracefully stops `SDL::Mixer`:

```
SDL::Mixer::Effects::unregister( MIX_CHANNEL_POST, $music_data_effect_id );
SDL::Mixer::Music::hook_music_finished();
SDL::Mixer::Music::halt_music();
SDL::Mixer::close_audio();
```

The result? Several dozen lines of code to glue together the SDL mixer and display a real-time visualization of the music.

10

CPAN

The Comprehensive Perl Archive Network (CPAN) is the other part of the Perl language. By now most Perl developers should be aware of how to search and get modules from CPAN. This chapter will focus on why to use CPAN for games. Next we will take a look in what domain (Model, View or Controller) does a module solve a problem for. Moreover we would want to look at what is criteria to pick one module from another, using the many tools provided by CPAN.

10.1 Modules

It is good to reuse code.

10.1.1 MVC Method

See where the module fits, Model, View or Controller

View

SDL will do most but helper module (Clipboard) are cool to have.

The *SDLx::Widget* bundle comes separately, but is meant to provide you with several common game elements such as menu, dialog boxes and buttons, all seamlessly integrated with SDL.

Model

The logic and modelling behind most popular games is already on CPAN, so you can easily plug them in to create a new game of Chess, Checkers, Go, Life, Minesweeping, Cards, etc. There are even classes for platform games (like *Games::Nintendo::Mario*), creating and solving mazes, generating random dungeon maps, you name it. Have a look at *Roguelike-Utils* and *Games::RolePlay::MapGen* for just a few of those.

If your game needs to store data, like objects and status for saved games or checkpoints, you can use *Storable* or any of the many data serializers available.

In fact, speaking of data structures, it is common to keep game data in standard formats such as JSON, YAML or XML, to make you able to import/export them directly from third-party tools like visual map makers or 3D modeling software. Perl provides very nice modules to handle the most popular formats - and some pretty unusual ones. Parsers vary in speed, size and thoroughness, so make sure to check the possible candidates and use the one that fits your needs for speed, size and accuracy.

If you need to roll a dice, you can use *Games::Dice*, that even lets you receive an array of rolled dice, and use RPG-like syntax (e.g. "2d6+1" for 2 rolls of a 6-side die, adding 1 to the result).

You can also use *Sub::Frequency* if you need to do something or trigger a particular action or event only sometimes, or at a given probability.

Your game may need you to mix words, find substrings or manipulate word permutations in any way (like when playing scrabble), in which case you might find the *Games::Word* module useful.

10.2 Picking Modules

So, you thought of a nice game, identified your needs, typed some keywords in `HTTP://Search.CPAN.Org`, and got tons of results. What now? How to avoid vaporware and find the perfect solution for your needs?

10.2.1 Documentation

Once you find a potential module for your application, make sure you will know how to use it. Take a look at the SYNOPSIS section of the module, it should contain some code snippets showing you how to use the module's main features. Are you comfortable with the usage syntax? Does it seem to do what you expect it to? Will it fit nicely to whatever it is you're coding?

Next, skim through the rest of the documentation. Is it solid enough for you? Does it look complete enough for your needs, or is it easily extendable?

10.2.2 License

It's useless to find a module you can't legally use. Most (if not all) modules in `HTTP://Search.CPAN.Org` are free and open source software, but even so each needs a license telling developers what they can and cannot do with it. A lot of CPAN modules are released "*under the same terms as Perl itself*", and this means you can pick between the Artistic License or the GPL (version 1).

Below is a short and incomplete list of some popular license choices by CPAN developers:

- Artistic License - `HTTP://Dev.Perl.Org/licenses/artistic.html`
- GPL (all versions and variations) - `HTTP://GNU.Org/licenses`
- MIT License - `HTTP://OpenSource.Org/licenses/mit-license.php`

See `HTTP://OpenSource.Org/licenses/alphabetical` for a comprehensive list with each license's full documentation.

You should be able to find the module's license by going to a "LICENSE AND COPYRIGHT" section, usually available at the bottom of the documentation, or by looking for a license file inside that distribution.

Note: Some modules might even be released into CPAN as *public domain*, meaning they are not covered by intellectual property rights at all, and you are free to use them as you see fit. Even so, it's usually considered polite to mention authors as a courtesy, you know, giving credit where credit is due.

10.2.3 Ratings

The CPAN Ratings is a service where developers rate modules they used for their own projects, and is a great way to have some actual feedback on how it was to use the code on a real application. The ratings are compiled into a 1 to 5 grade, and displayed below the module name on CPAN. You can click on the "*Reviews*" link right next to the rating stars to see any additional comments by the reviewers, praising, criticizing or giving some additional comments or the distribution and/or its competition.

10.2.4 Dependencies

Modules exist so you don't have to reinvent the wheel, and for that same reason each usually depends on one or more modules itself. Don't worry if a module depends on several others - code reusability is a good thing.

You may, however, be interested in **which** modules it depends on, or, more practically, in the likelihood of a clean installation by your users. For that, you can browse to `HTTP://Deps.CPANTesters.Org` and input the module's name on the search box.

The CPAN Testers is a collaborative matrix designed to help developers test their modules in several different platforms, with over a hundred testers each month making more than 3 million reports of CPAN modules. This particular CPAN Testers service will show you a list of dependencies and test results for each of them, calculating the average chance of all tests passing (for any platform).

While seeing all the dependencies and test results of a couple of modules that do the same thing might help you make your pick, it's important to realize that the "*chance of all tests passing*" information at the bottom of the results means very little. This is because test failures can rarely be considered independent events, and are usually tied to not running on a specific type of operating system, to the perl version, or even due to the tester running out of memory for reasons that may not even concern the module being evaluated. If you don't care about your application running on AIX or on perl 5.6.0, why would you dismiss a module that only fails on those conditions?

10.2.5 CPAN Testers Charts

So, how do you know the actual test results for a module on the CPAN? How can you tell if that module will run in your target machine according to architecture, operating system and perl version?

The CPAN Testers website at `HTTP://CPANTesters.Org` offers a direct search for distributions by name or author. To see the results for the SDL module, for instance, you can go to `HTTP://CPANTesters.Org/distro/S/SDL.html`. You can also find a test report summary directly on CPAN, by selecting the distribution and looking at the "*CPAN Testers*" line. If you click on the "*View Reports*" link, you'll be redirected to the proper CPAN Testers page, like the one shown above.

The first chart is a PASS summary, containing information about the most recent version of that module with at least one *PASS* report submitted, separated by platform and perl version.

Second is a list of selected reports, detailing all the submitted test results for the latest version of the given module. If you see a *FAIL* or *UNKNOWN* result that might concern you - usually at a platform you expect your application to run - you can click on it to see a verbose output of all the tests, to see why it failed.

Another interesting information displayed is the report summary on the left sidebar, showing a small colored graph of PASS-UNKNOWN-FAIL results for the latest versions of the chosen module. If you see a released version with lots of FAIL results, it might be interesting to dig deeper or simply require a greater version of that module in your application.

Bug Reports

When picking a module to use, it is very important to check out its bug reports. You can do that by either clicking on the "*View/Report Bugs*" link on the module's page on CPAN, or on the "*CPAN RT*" (for Request Tracker) box on the right side of the documentation page.

Look for open bugs and their description - i.e. if it's a bug or a whislist - and see if it concerns your planned usage for that module. Some bug reports are simple notices about a typo on the documentation or a very specific issue, so make sure you look around the ticket description to see if it's something that blocks your usage, or if you can live with it, at least until the author delivers an update.

It may also interest you to see how long the open bugs have been there. Distributions with bugs dating for more than two years might indicate that the author abandoned the module to pursue other projects, so you'll likely be on your own if you find any bumps. Of course, being free software, that doesn't mean you can't fix things yourself, and maybe even ask the author for maintainance privileges so you can update your fixes for other people to use.

10.2.6 Release Date

A old distribution might mean a solid and stable distribution, but it can also mean that the author doesn't care much about it anymore. If you find a module whose latest version is over 5 years old, make sure to double check test results and bug reports, as explained above.

10.3 Conclusion

CPAN is an amazing repository filled with nice modules ready for you to use in your games. More than often you'll find that 90% of your application is already done on CPAN, and all you have to do to get that awesome idea implemented is glue them together, worrying only about your application's own logic instead of boring sidework. This means faster development, and more fun!

10.4 Author

This chapter's content graciously provided by Breno G. de Oliveira (`garu`).

11

Pixel Effects

In this chapter we will look at how to use pixel effects in Perl. Pixel effects are operations that are done directly on the bank of a `SDL_Surface`'s pixel. These effects are used to do visual effects in games and applications, most notably by `Frozen Bubble`.

These effects can be done in purely in Perl, for 1 passes and non real time applications. Effects that need to be done real time will have to be done in C via XS. This chapter will show two methods of doing this.

11.1 Sol's Ripple Effect

For our first pixel effect we will be doing is a ripple effect from a well known SDL resource, `HTTP://Sol.Gfxile.Net/gp/ch02.html`. This effects uses `SDL::get_ticks` to animate a ripple effect across the surface as seen in the following figure.

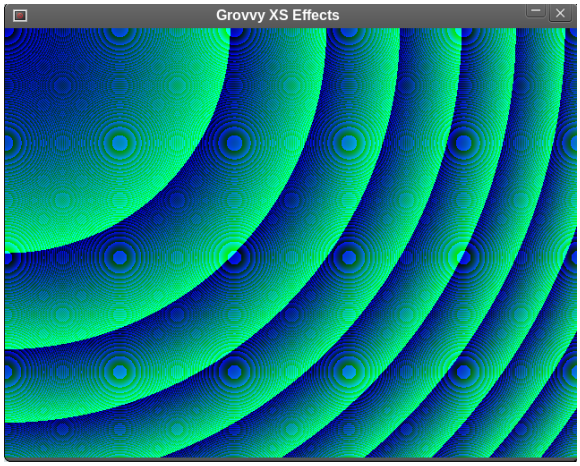

Figure 11.1: Sol's Chapter 01 Ripple Effect

11.1.1 Pure Perl

First lets make the effect in pure Perl. To do any operations with a `SDL::Surface` we must do `SDL::Video::lock_surface()` call as seen below. Locking the surface prevents other process in SDL from accessing the surface. The surface pixels can be accessed several ways from Perl. Here we are using the `SDL::Surface::set_pixels` which takes an offset for the `SDL_Surface` pixels array, and sets a value there for us. The actual pixel effect is just a time dependent (using `SDL::get_ticks` for time) render of a function. See `HTTP://Sol.Gfxile.Net/gp/ch02.html` for a deeper explanation.

```
1 use strict;
2 use warnings;
```

```
3
4  use SDL;
5  use SDLx::App;
6
7      # Render callback that we use to fiddle the colors on the surface
8  sub render {
9      my $screen = shift;
10     if ( SDL::Video::MUSTLOCK($screen) ) {
11         return if ( SDL::Video::lock_surface($screen) < 0 );
12     }
13
14     my $ticks = SDL::get_ticks();
15     my ( $i, $y, $yofs, $ofs ) = ( 0, 0, 0, 0 );
16     for ( $i = 0; $i < 480; $i++ ) {
17         for ( my $j = 0, $ofs = $yofs; $j < 640; $j++, $ofs++ ) {
18             $screen->set_pixels( $ofs, ( $i * $i + $j * $j + $ticks ) );
19         }
20         $yofs += $screen->pitch / 4;
21     }
22
23
24     SDL::Video::unlock_surface($screen) if ( SDL::Video::MUSTLOCK($screen) );
25
26     SDL::Video::update_rect( $screen, 0, 0, 640, 480 );
27
28     return 0;
29 }
30
31
32 my $app = SDLx::App->new( width => 640,
33                           height => 480,
34                           eoq => 1,
35                           title => "Grovvy XS Effects" );
36
37 $app->add_show_handler( sub{ render( $app ) } );
38
39 $app->run();
```

One you run this program you will find it pretty much maxing out the CPU and not running very smoothly. At this point running a loop through the entire pixel bank of a `640x480` sized screen is too much for Perl. We will need to move the intensive calculations to `C`.

11.1.2 Inline Effects

In the below example we use `Inline` to write Inline `C` code to handle the pixel effect for us. `SDL` now provides support to work with `Inline`. The `render` callback is now moved to `C` code, using `Inline C`. When the program first runs it will compile the code and link it in for us.

```
use strict;
use warnings;
use Inline with => 'SDL';
use SDL;
use SDLx::App;


my $app = SDLx::App->new( width => 640,
                          height => 480,
                          eoq => 1,
                          title => "Grovvy XS Effects" );

    # Make render a callback which has the expected signature from show_handlers
$app->add_show_handler( \&render);

$app->run();

use Inline C => <<'END';

    // Show handlers recieve both float and the SDLx::App which is a SDL_Screen
void render( float delta, SDL_Surface *screen )
{
    // Lock surface if needed
    if (SDL_MUSTLOCK(screen))
```

```
25          if (SDL_LockSurface(screen) < 0)
26              return;
27
28      // Ask SDL for the time in milliseconds
29      int tick = SDL_GetTicks();
30
31      // Declare a couple of variables
32      int i, j, yofs, ofs;
33
34      // Draw to screen
35      yofs = 0;
36      for (i = 0; i < 480; i++)
37      {
38          for (j = 0, ofs = yofs; j < 640; j++, ofs++)
39          {
40              ((unsigned int*)screen->pixels)[ofs] = i * i + j * j + tick;
41          }
42          yofs += screen->pitch / 4;
43      }
44
45      // Unlock if needed
46      if (SDL_MUSTLOCK(screen))
47          SDL_UnlockSurface(screen);
48
49      // Tell SDL to update the whole screen
50      SDL_UpdateRect(screen, 0, 0, 640, 480);
51  }
52
53  END
```

12

Additional Modules

12.1 PDL

The Perl Data Language (PDL) is a tool aimed at a more scientific crowd. Accuracy is paramount and speed is the name of the game. PDL brings to Perl fast matrix and numerical calculations. For games in most cases a accuracy is not critical, but speed and efficiency is a great concern. For this reason we will briefly explore how to share SDL texture data between PDL and OpenGL.

This example will do the following:

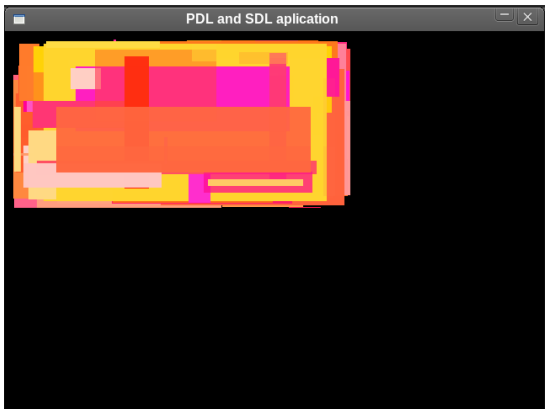

Figure 12.1: Not terribly interesting, but the speed is phenomenal

12.1.1 Make the application

Let's start an application to use with PDL. Make sure you do `use PDL`.

```
+ use strict;
+ use warnings;
+ use SDL;
+ use SDL::Video;
+ use SDLx::App;
+
+ use PDL;
+
+ my $app = SDLx::App->new(
+                           title => 'PDL and SDL application',
+                           width => 640, height => 480, depth => 32,
+                           eoq => 1);
```

12.1.2 Attaching the Piddle

PDL core object is something called a piddle. To be able to perform PDL calculations and show them on SDL surfaces, we need to share the memory between them. SDL Surface memory is stored in a `void *` block called `pixels`. `void *` memory has the property that allows Surfaces to have varying depth, and pixel formats. This also means that we can have PDL's memory as our `pixels` for our surface.

```
+ sub make_surface_piddle {
+ my ( $bytes_per_pixel, $width, $height) = @_;
+ my $piddle = zeros( byte, $bytes_per_pixel, $width, $height );
+ my $pointer = $piddle->get_dataref();
```

At this point we have a pointer to the `$piddle`'s memory with the given specifications. Next we have our surface use that memory.

```
+ my $s = SDL::Surface->new_form(
+                                      $pointer, $width, $height, 32,
+                                      $width * $bytes_per_pixel
+                                     );
+
+ #Wrap it into a SDLx::Surface for ease of use
+ my $surface = SDLx::Surface->new( surface => $s );
+
+ return ( $piddle, $surface );
+ }
```

Lets make some global variables to hold our `$piddle` and `$surface`.

```
+ my ( $piddle, $surface ) = make_surface_piddle( 4, 400, 200 );
```

12.1.3 Drawing and Updating

`make_surface_piddle()` will return to use an anonymous array with a `$piddle` and `$surface` which we can use with PDL and SDL. PDL will be used to operate on the `$piddle`. SDL will be used to update the `$surface` and render it to the `SDLx::App`.

```
+ $app->add_move_handler( sub {
+
+   SDL::Video::lock_surface($surface);
+
+   $piddle->mslice( 'X',
+   [ rand(400), rand(400), 1 ],
+   [ rand(200), rand(200), 1 ]
+   ) .= pdl( rand(225), rand(225), rand(225), 255 );
+
+   SDL::Video::unlock_surface($surface);
+ } );
```

`SDL::Video::lock_surface` prevents SDL from doing any operations on the `$surface` until `SDL::Video::unlock_surface` is called. Next we will blit this surface onto the `$app`.

In this case we use PDL to draw random rectangles of random color.

12.1.4 Running the App

Finally we blit the `$surface` and update the `$app`.

```
+ $app->add_show_handler( sub {
+
+    $surface->blit( $app, [0,0,$surface->w,$surface->h], [10,10,0,0] );
+    $app->update();
+
+ });
```

```
+ $app->run();
```

12.1.5 Complete Program

```
use strict;
use warnings;
use SDLx::App;

use PDL;

my $app = SDLx::App->new(
        title => "PDL and SDL aplication",
        width => 640, height => 480, eoq => 1 );


sub make_surface_piddle {
    my ( $bytes_per_pixel, $width, $height) = @_;
    my $piddle = zeros( byte, $bytes_per_pixel, $width, $height );
    my $pointer = $piddle->get_dataref();
    my $s = SDL::Surface->new_from(
        $pointer, $width, $height, 32,
        $width * $bytes_per_pixel
    );

    my $surface = SDLx::Surface->new( surface => $s );

    return ( $piddle, $surface );
}


my ( $piddle, $surface ) = make_surface_piddle( 4, 400, 200 );

$app->add_move_handler( sub {

        SDL::Video::lock_surface($surface);
```

```
32
33          $piddle->mslice( 'X',
34              [ rand(400), rand(400), 1 ],
35              [ rand(200), rand(200), 1 ]
36              ) .= pdl( rand(225), rand(225), rand(225), 255 );
37
38          SDL::Video::unlock_surface($surface);
39          } );
40
41
42  $app->add_show_handler( sub {
43
44      $surface->blit( $app, [0,0,$surface->w,$surface->h], [10,10,0,0] );
45      $app->update();
46
47  });
48
49  $app->run();
```

12.2 OpenGL and SDL

OpenGL is a cross platform library for interactive 2D and 3D graphics applications. However OpenGL specifies only the graphics pipeline and doesn't handle inputs and events. SDL can hand over the graphics component of an application over to OpenGL and take control over the event handling, sound, and textures. In the first example we will see how to set up Perl's `OpenGL` module with `SDLx::App`.

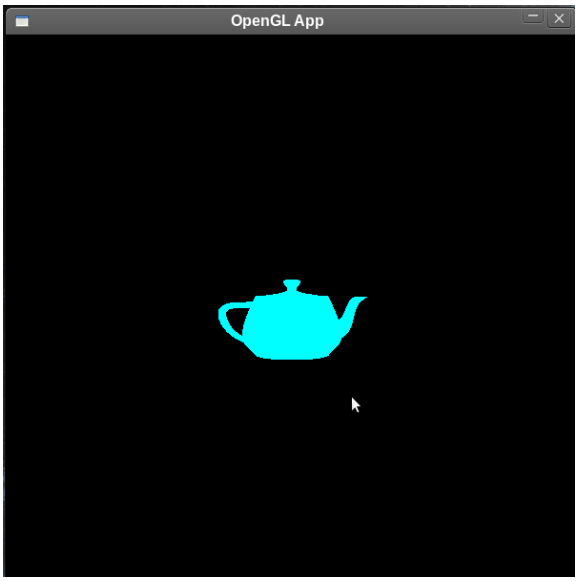

Figure 12.2: The lovely blue teapot

12.2.1 SDL Setup

```
use strict;
use warnings;
use SDL;
use SDLx::App;

use OpenGL qw/:all/;

my $app = SDLx::App->new(
        title  => "OpenGL App",
        width  => 600,
        height => 600,
        gl     => 1,
        eoq    => 1
   );
```

```
$app->run();
```

Enabling `OpenGL` mode is as simple as adding the `gl` flag to the `SDLx::App` constructor.

12.2.2 OpenGL Setup

Next we will make a `OpenGL` perspective with the `$app`'s dimensions:

```
glEnable(GL_DEPTH_TEST);
glMatrixMode(GL_PROJECTION);
glLoadIdentity;
gluPerspective(60, $app->w/$app->h, 1, 1000 );
glTranslatef( 0,0,-20);
```

Additionally we will be initializing `glut`, but just to draw something quick.

```
#Using glut to draw something interesting really quick
glutInit();
```

12.2.3 The Render Callback

Now we are prepared to put something on the screen.

```
$app->add_show_handler(
        sub{
                my $dt = shift;

                #clear the screen
                glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
                glColor3d(0,1,1);
```

```
                glutSolidTeapot(2);

                #sync the SDL application with the OpenGL buffer data
                $app->sync;


        }
    );
```

At this point there should be a light blue teapot on the screen. The only special thing to notice here is that we need to call the `sync()` method on `$app`. This will flush the buffers and update the SDL application for us.

12.2.4 Event handling

Event handling is the same as any other `SDLx::App`. We will use the mouse motion changes to rotate the teapot.

First add a global variable to hold your rotate values. And then use those values to rotate our teapot.

```
  glutInit();

+ my $rotate = [0,0];

  $app->add_show_handler(
    sub{
    my $dt = shift;

    #clear the screen
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glColor3d(0,1,1);

+   glPushMatrix();
```

```
    +       glRotatef($rotate->[0], 1,0,0);
+   glRotatef($rotate->[1], 0,1,0);

    glutSolidTeapot(2);

    #sync the SDL application with the OpenGL buffer data
    $app->sync;

    glPopMatrix();
    }
);
```

Next we will add an event handler to the app to update the rotate values for us.

```
$app->add_event_handler(

    sub {
        my ($e ) = shift;

        if( $e->type == SDL_MOUSEMOTION )
            {
                $rotate =   [$e->motion_x,  $e->motion_y];
            }

    }

);
```

Finally we run the application.

```
$app->run();
```

12.2.5 Complete Code

```
use strict;
use warnings;
use SDL;
use SDLx::App;
use SDL::Event;

use OpenGL qw/:all/;

my $app = SDLx::App->new(
        title  => "OpenGL App",
        width  => 600,
        height => 600,
        gl     => 1,
        eoq    => 1
        );

glEnable(GL_DEPTH_TEST);
glMatrixMode(GL_PROJECTION);
glLoadIdentity;
gluPerspective(60, $app->w/$app->h, 1, 1000 );
glTranslatef( 0,0,-20);
glutInit();

my $rotate = [0,0];

$app->add_show_handler(
        sub{
        my $dt = shift;

#clear the screen
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
        glColor3d(0,1,1);

        glPushMatrix();

```

```
        glRotatef($rotate->[0], 1,0,0);
        glRotatef($rotate->[1], 0,1,0);

        glutSolidTeapot(2);

#sync the SDL application with the OpenGL buffer data
        $app->sync;

        glPopMatrix();
        }
        );

$app->add_event_handler(

        sub {
        my ($e ) = shift;

        if( $e->type == SDL_MOUSEMOTION )
        {
        $rotate =   [$e->motion_x,  $e->motion_y];
        }

        }

        );

$app->run();
```

13

Free Resources

When developing a game, coding is unfortunately not everything. Not by a very, very long shot. To make up (a little) for that, below is a list of free resources you can use in your games, either in full or simply as inspiration for your own productions, in case you have an artistic vein yourself.

Make sure to check the licence for the resource and use it accordingly, giving the original author proper credit.

Note: websites come and go, so if you find any of the links broken, or know a nice free resource that's not listed here, please let us know so we can update the list.

13.1 Art and Sprites

- HTTP://CGTextures.Com
- HTTP://Mayang.Com/textures
- HTTP://GRSites.Com/archive/textures
- HTTP://ImageAfter.Com
- HTTP://AbsoluteCross.Com/graphics/textures
- HTTP://FreeFoto.Com
- HTTP://Noctua-Graphics.De
- HTTP://M3Corp.Com/a/download/3d_textures/pages
- HTTP://ReinersTileSet.4Players.De/englisch.html
- HTTP://VirtualWorlds.Wikia.Com
- HTTP://Lunar.LostGarden.Com/labels/free%20game%20graphics.html
- HTTP://PDGameResources.WordPress.Com
- HTTP://GamingGroundZero.Com
- HTTP://FlyingYogi.Com/fun/spritelib.html
- HTTP://PixelPoke.Com

13.2 Music and Sound Effects

- HTTP://FreeSound.Org
- HTTP://CCMixter.Org
- HTTP://Jamendo.Com
- HTTP://8BC.Org
- HTTP://Sakari-Infinity.Net
- HTTP://FindSounds.Com
- HTTP://GRSites.Com/archive/sounds

13.3 Fonts

- HTTP://DAFont.Com
- HTTP://FontSquirrel.Com
- HTTP://TheLeagueOfMoveableType.Com
- HTTP://OpenFontLibrary.Org
- HTTP://AcidFonts.Com
- HTTP://GRSites.Com/archive/fonts
- HTTP://UrbanFonts.Com

13.4 DIY

HTTP://GameSoundDesign.Com has several tips on making game music, including several sources for inspiration.

If you want to create 3D models, either for cutscenes or to integrate into your game via OpenGL, there are several nice libraries out there for you:

Blender - A free 3D graphics application for modeling, texturing, water and smoke simulations, rendering, etc. HTTP://Blender.Org

OGRE - An open-source graphics rendering engine, used in a large number of production projects. It can be easily integrated via Scott Lanning's *Ogre* Perl bindings, on CPAN. HTTP://Ogre3D.Org

13.5 Author

This chapter's content graciously provided by Breno G. de Oliveira (garu).

Index