MAKING A SIMPLE GAME IN MINIGL

In this tutorial, we will learn how to make a basic game using the MiniGL libarary. The goal is to be able to expose you to the library by walking through creating a simple game, with the hope that you will be armed with enough knowledge to build more complex programs later on. No prior knowledge with MiniGL or graphics programming is assumed for this tutorial, all that is required is a basic understanding of programming in C++.

A BRIEF INTRO TO MINIGL

It is appropriate to start by introducing MiniGL. MiniGL is a C++ graphics library that provides a minimal but expressive interface for 2D graphics. This library offers a pain free way to quickly render 2D graphics without any systems level or graphics expertise required. MiniGL's API is small, but it is surprising how expressive it is as we'll see in this tutorial. Specifically I will walk you through basic features of MiniGL like animations and events which will work towards building a simple pong-like game.

DRAWING BASIC SHAPES

First let's create a file called `pong.cpp`. You will need the source files of MiniGL to be able to build and run the program. See the README in the GitHub repo for more information about building and running. For now, just import the library and make sure you can build and run the program.

```cpp
#include <iostream>
#include "minigl2d.hpp"

using namespace minigl;

int main()
{
  std::cout << "in the pong program!\n";
}
```

The central construct of MiniGL is a shape, all of the applications that you create with MiniGL will revolve around the creation and manipulation of shapes. MiniGL offers built in shapes, but also allows users to easily define shapes (see the manual for more details). For now, we'll be working with the prebuilt shapes namely the `rectangle` and the `circle`. Before we get anything going, we need to first define a window. To create a window, use the `window2d` class as follows.

```cpp
#include <iostream>
#include "minigl2d.hpp"

using namespace minigl;

int main()
{
  std::cout << "in the pong program!\n";
  window2d win(1200_px, 800_px, color(colors::dark_grey), "Pong");
}
```

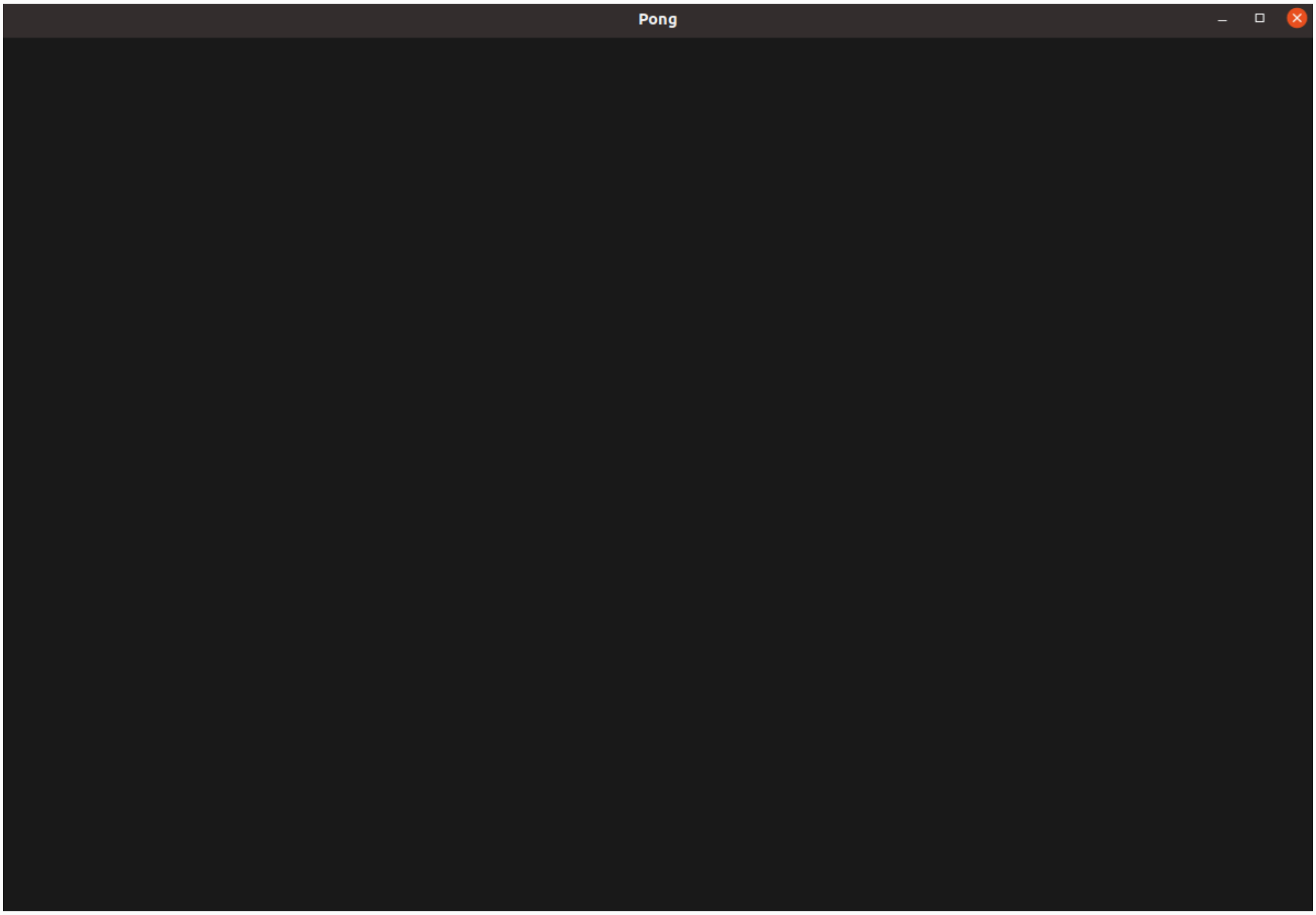
Here we have defined a window that is 1200 pixels wide and 800 pixels tall. The background color will be dark grey and will have a title called "Pong". You will notice that running this code does nothing. We have to feed this window object into one of the render functions in the `render2d` class. For now, let's use `render2d::draw` to display the window.

```
#include <iostream>
#include "minigl2d.hpp"

using namespace minigl;

int main()
{
  std::cout << "in the pong program!\n";
  window2d win(1200_px, 800_px, color(colors::dark_grey), "Pong");
  std::vector<shape> my_shapes = {};
  render2d::draw(win, my_shapes);
}
```

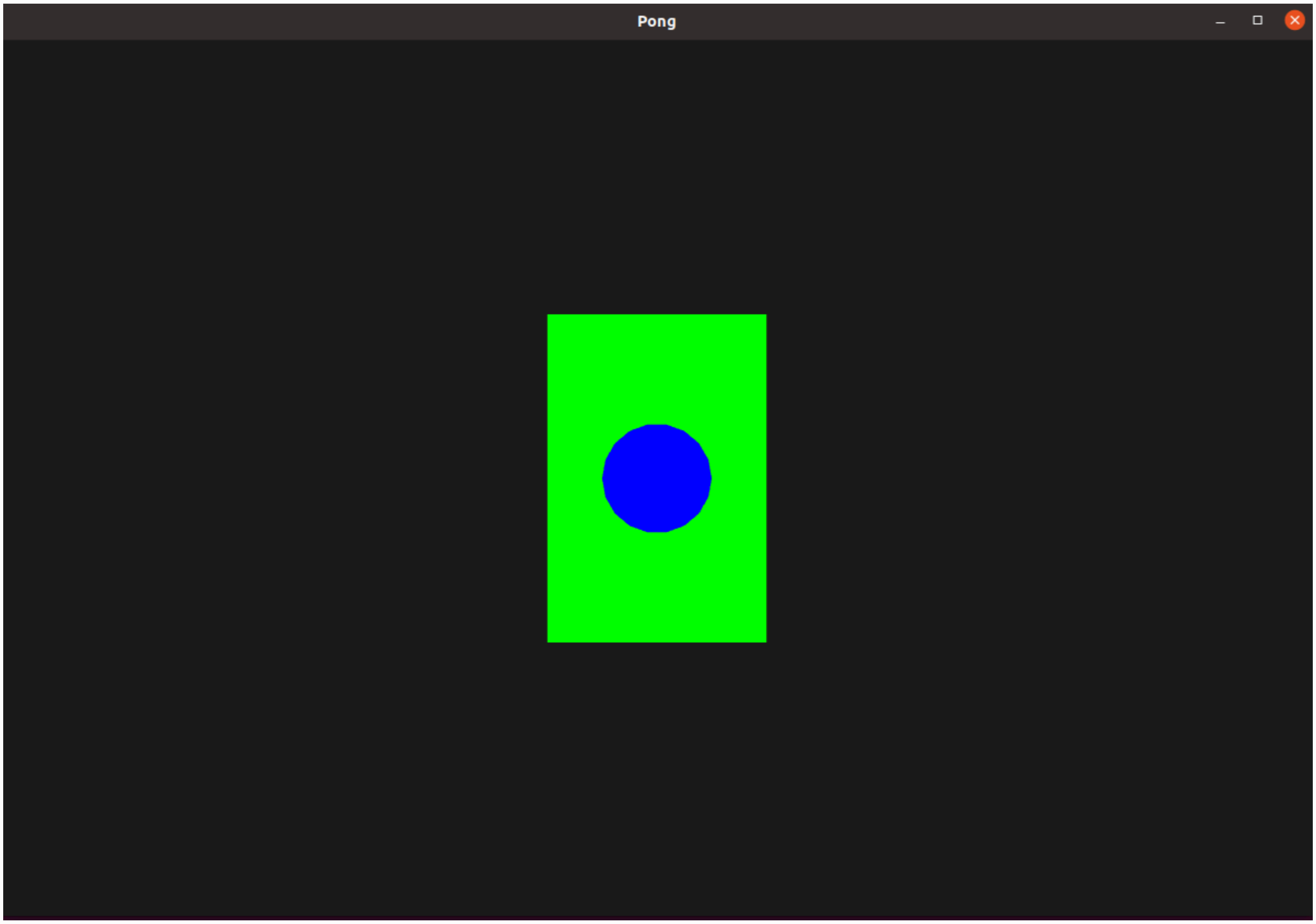
When running the above snippet, you should see a window that looks like one below appear on the screen.



Closing the window or pressing the escape key should terminate the program. A blank window isn't of much use for us though so let's display some shapes. Notice in the code snippet above, we defined `std::vector<shape> my_shapes = {}` but there is nothing in the vector. Let's populate that vector with a couple shapes. Put a circle and a rectangle in the vector.

```
std::vector<shape> my_shapes = {
  circle(50_px, colors::blue),
  rectangle(200_px, 300_px, colors::green),
};
```

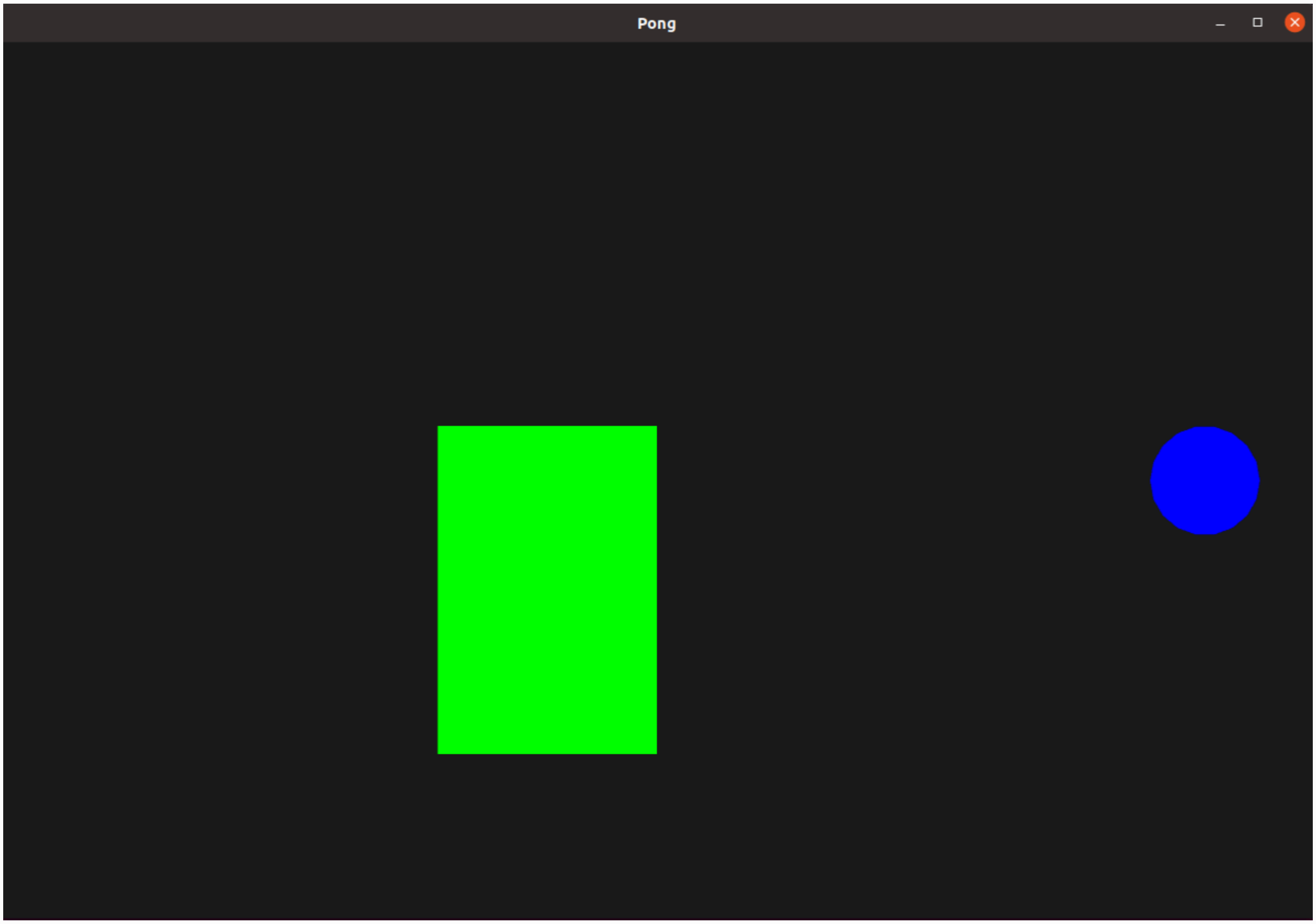
Notice that we are able to put a `circle` and a `rectangle` into a vector of `shape` s without the compiler complaining. This is because `circle` and `rectangle` are subclasses of `shape` . Running this now should display this window.



I encourage you to play around with the shapes and tweaking some parameters, for example changing the radius, changing the colors, adding more shapes to the vector and the order that the shapes appear in the vector. Notice that the MiniGL draws the shapes appearing in the beginning of the vector to be *in front*. That's why if we flip the ordering of the vector above, the circle will be hidden behind the rectangle. We can also adjust the position of these shapes by defining a `position` in the constructor like the following.

```
std::vector<shape> my_shapes = {
  circle(50_px, colors::blue, position(500, 0)),
  rectangle(200_px, 300_px, colors::green, position(-100, -100)),
};
```

Running the following should display this window.



Also notice that the origin is in the center of the screen. So setting a position of `position(0,0)` will put the shape right in the center. This is implicit if a position is not defined in the shape constructor. See the API manual for more details of all the available constructors.

BASIC ANIMATIONS

Plotting shapes on a screen is cool and all but what if we want to animate and interact with the shapes? In this case, we'd want to use `render2d::animate()` . Notice the function signature:

```
static void animate(
  window2d& win,
  int fps,
  std::vector<shape> shapes,
  std::function<void(std::vector<shape>&, events)> func);
```

We see that in addition to requiring a window and a vector of `shape` s, we also need to provide an `fps` and a function `func` that takes in a vector of shapes and an events object. fps is short for "frames per second" how many times the shapes are rendered on the screen per second. A higher fps will result in a smoother animation. Note however that most displays can only display at a max framerate of 60 fps.

Now let's talk about `func`. `func` defines how the shape in every frame. The best way to understand this is by using a simple example. Take a look at and run the following snippet of code.

```
#include <iostream>
#include "minigl2d.hpp"

using namespace minigl;
using namespace glm;
```

```
int main()
{
  window2d win(1200_px, 800_px, color(colors::dark_grey), "Pong");

  std::vector<shape> initial_world = {
    rectangle(200_px, 300_px, colors::blue),
  };

  // for each frame, move a circle 3 pixels to the right
  render2d::animate(win, 60, initial_world,
  [&](std::vector<shape> &world, events e) {
    world[0].translate(3,0);
  });
}
```

Above we are using `render2d::animate()` and we pass in an fps of 60 and a lambda function that translates the first (and only) element of the `world` to the right by 3 pixels for each frame. Running this should animate a rectangle moving to the right. Also notice that `func` is defined as a lambda. This is the recommended way to define `func` as it allows us to pass in state from outside of the function, which is useful in many circumstances.

A PONG-LIKE GAME

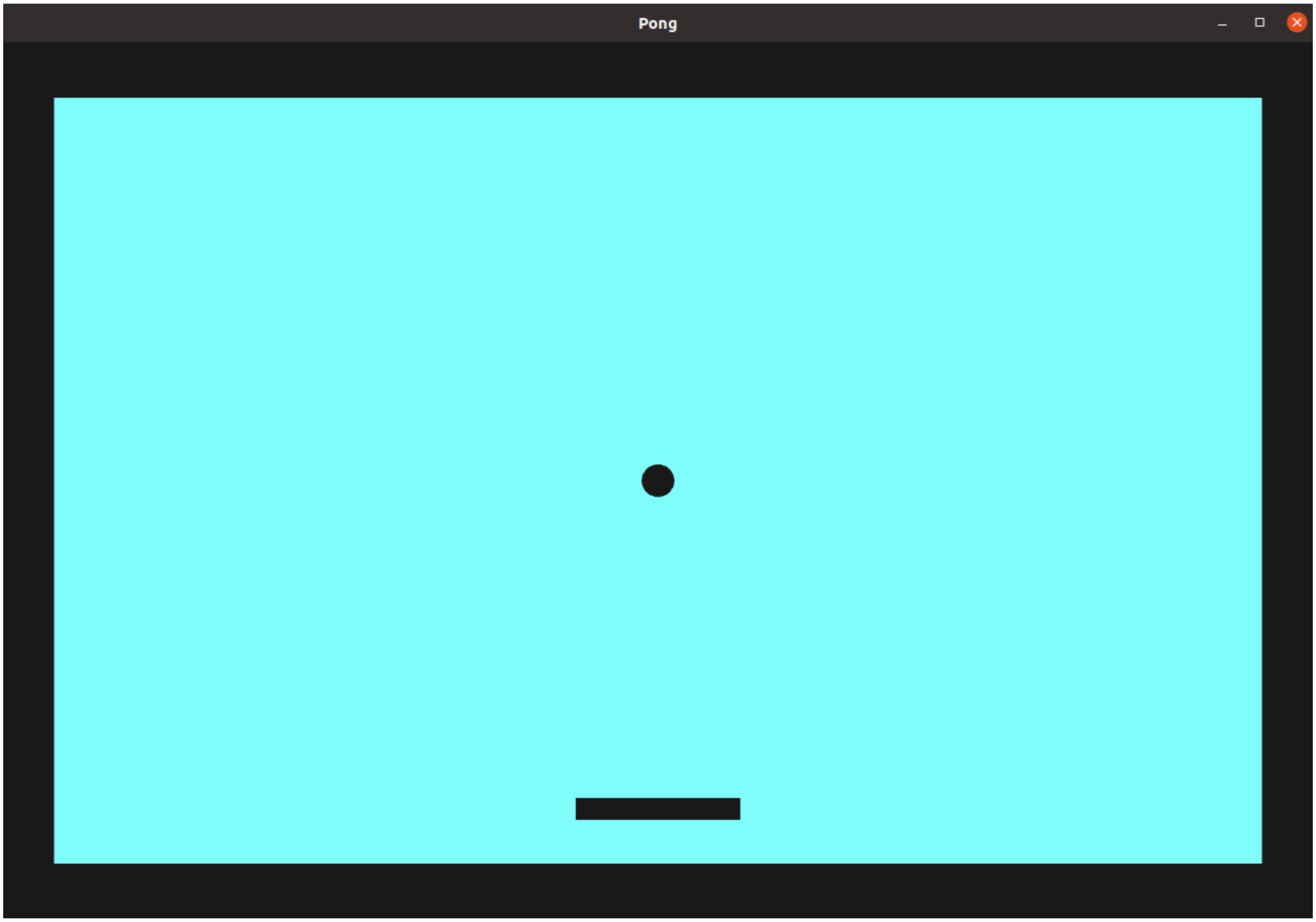
Now that we have a sense of how a simple animation works, let's make this look more like pong. Let's define 3 shapes -- 2 rectangles and one circle. The first rectangle will act as the border for our game. The second rectangle will be our paddle, the circle will be our ball. With this in mind, define the `initial_world` vector as follows:

```
std::vector<shape> initial_world {
  circle(15_px, colors::dark_grey), // the ball
  rectangle(150_px, 20_px, colors::dark_grey, position(0, -300)),
// the paddle
  rectangle(1100_px, 700_px, colors::cyan), // the border
};
```

Let's animate for 60fps like before and leave `func` empty for now. Your animate function should look like this:

```
render2d::animate(win, 60, initial_world,
[&](std::vector<shape> &world, events e) {
});
```

Leave the window dimensions the same as what we have before. Running you code should now display something that looks a lot like a single player version of pong.



Now we need to make the paddle actually react to our inputs. That is where the `events` struct in `func` comes in handy. `events` contains all the data we need to get the current cursor position as well as keyboard/mouse buttons that are currently being pressed. Let's make the paddle move by using the A key to go left and the D key to go right. To do this, we need to access the `pressed_keys` field of `events`. `pressed_keys` is an array of booleans that tells us if a button is being pressed. For instance, to check if the A key is being pressed, we can simply check the boolean result of the following `e.pressed_keys[A_key]`. So the idea is that when the A key is being pressed, we want to translate the paddle to the left, and when the D key is being pressed, we want to translate the paddle to the right. Our animate function with the paddle move code should look like this:

```
render2d::animate(win, 60, initial_world,
[&](std::vector<shape> &world, events e) {
  shape& paddle = world[1];
  if (e.pressed_keys[A_KEY])
    paddle.translate(position(-6, 0));
  if (e.pressed_keys[D_KEY])
    paddle.translate(position(6, 0));
});
```

Here I chose to translate by 6 pixels per frame but feel free to choose whatever speed you would like. Running the code should now allow you to move the paddle left and right using the A and D keys.

Now for the hard part, we need to make the ball move and bounce when colliding with the wall or paddle. There are many different ways to implement this. But the idea is to keep track of some variablble that stores the ball's current velocity. In my case I just defined this as a 2d glm vector `glm::vec2 ball_vel(7, 7);`. Again the initial vector of $(7, 7)$ is arbitrary and feel free to tweak the parameters however you like. Now for the bouncing, I am simply reversing the element normal to the wall's direction. For example, if there is a collision with the left or right wall then I do `ball_vel[0] = -ball_vel[0];` and if there is a collision with the top wall, bottom wall or paddle, then I do `ball_vel[1] = -ball_vel[1];`. We can check if we collided with the wall or paddle by getting the position of the ball. We can do this using `position shape::get_pos()`. After making all of these changes, my `pong.cpp` file looks like the following.

```
#include <iostream>
#include "minigl2d.hpp"

using namespace minigl;
using namespace glm;

int main()
{
```

```
  window2d win(1200_px, 800_px, color(colors::dark_grey), "Pong");
    const int border_width = 1100;
    const int border_height = 700;
    const int radius = 15;
    const int paddle_width = 150;
    const int paddle_height = 20;
    const int paddle_speed = 6;
    const int paddle_ypos = -300;
    glm::vec2 ball_vel(7, 7);
    std::vector<shape> initial_world {
      circle(pixels(radius), colors::dark_grey),
      rectangle(
        pixels(paddle_width),
        pixels(paddle_height),
        colors::dark_grey,
        position(0, paddle_ypos)),
      rectangle(pixels(border_width), pixels(border_height),
colors::cyan),
    };
    render2d::animate(win, 60, initial_world,
    [&](std::vector<shape> &world, events e) {
      shape& ball = world[0];
      shape& paddle = world[1];
      if (e.pressed_keys[A_KEY])
        paddle.translate(position(-paddle_speed, 0));
      if (e.pressed_keys[D_KEY])
        paddle.translate(position(paddle_speed, 0));
      // move ball
      ball.translate(ball_vel);
      // about to hit left wall or right wall
      int horiz_edge = border_width/2 - radius;
      int vert_edge = border_height/2 - radius;
      if (ball.get_pos()[0] + ball_vel[0] >= horiz_edge
        || ball.get_pos()[0] + ball_vel[0] <= -horiz_edge)
        ball_vel[0] = -ball_vel[0];
      // about to hit the top wall or the bottom wall
      if (ball.get_pos()[1] + ball_vel[1] >= vert_edge
        || ball.get_pos()[1] + ball_vel[1] <= -vert_edge)
        ball_vel[1] = -ball_vel[1];
      // about to hit top of paddle
```

```
        if (ball.get_pos()[1] + ball_vel[1] <= paddle_ypos +
paddle_height/2
            && ball.get_pos()[0] + ball_vel[0] >= paddle.get_pos()[0]
- paddle_width/2
            && ball.get_pos()[0] + ball_vel[0] <= paddle.get_pos()[0]
+ paddle_width/2)
            ball_vel[1] = -ball_vel[1];
    });
}
```

Running this should give a playable version of pong. It is clear why defining the `func` parameter of `render2d::animate()` as a lambda function is useful because it allows us to keep track of state outside of the function such as `ball_vel`. Additionally using the lambda gives us access to a lot of useful constants that we define outside of `func`. The code in the snippet above doesn't have an end state and I will leave that as an exercise for the reader to add. A simple idea could be to black the screen out if the ball hits the bottom wall (remove everything in `world` and replace it with a single black rectangle that covers the screen when the game ends).

It is worth noting that although this game is quite simple. The same primitives that we used here can be used to create more sophisticated animation/programs. See the manual for more insight on the capabilities of MiniGL along with advanced features such as defining own shapes/shaders and attaching textures.