Lecture 07

Player Input

26 Prairial, Year CCXXX

Song of the day: ***Winter I*** *by Antonio Vivaldi (1723) re-imagined by Max Richter (2022).*

Sections

Part 1: *Player Input*

Let's touch base for a bit and take a look at our game loop one more time:

```
int main(int argc, char* argv[])
{
    initialise();

    while (game_is_running)
    {
        process_input();
        update();
        render();
    }

    shutdown();
    return 0;
}
```

Code Block 1: Where it all started.

We have:

- Created a starting scene in `initialise()`.
- Created beautiful objects in `render()`.
- Animated these objects in `render()`.

If we considered the functions in here that we have, at least once, touched on or covered, we can see see that (aside from `shutdown()`) there is one that we have conspicuously not touched yet:

`process_input()`. You might find this funny, since the only thing that differentiates simple computer graphics from a video game is the ability for a player to make decision and convey those decision to the game.

Well that changes today. We have learned enough for us to graduate from the respectable practice of creating OpenGL art and into the realm of interactive computer graphics. *Id est*: video games.

Part 2: *Keyboard Input*

Alright, let's check out the current state of our `process_input()`:

```
void process_input()
{
    SDL_Event event;
    while (SDL_PollEvent(&event))
    {
        if (event.type == SDL_QUIT || event.type == SDL_WINDOWEVENT_CLOSE)
        {
            game_is_running = false;
        }
    }
}
```

Code Block 2: The lackluster state of our input-processing mechanism.

In order, here is a recap of what is happening:

1. We have a `while`-loop that is polling for a specific event, which is stored in an `SDL_Event` object.
2. The two events that we are current polling for are:
    1. `SDL_QUIT`: Quitting the game.
    2. `SDL_WINDOWEVENT_CLOSE`: Closing the window of the game.
3. If either of those events are polled, `game_is_running` becomes `false` and the game is over.

Polling for two events is pretty minimal and, while we're not going to get fancy to the point that we'll be accepting hundreds of inputs, using an `if`-statement will probably not be ideal. Thankfully `event.type` is an enumerator, meaning that we can use a `switch` statement to check for its value instead:

```
void process_input()
{
    SDL_Event event;
    while (SDL_PollEvent(&event))
    {
        switch (event.type)
        {
            case SDL_QUIT:
            case SDL_WINDOWEVENT_CLOSE:
                game_is_running = false;
                break;
        }
        }
    }
```

Code Block 3: The slightly-better-but-still-lackluster state of our input-processing mechanism.

Keystrokes

Alright, let's get started. The first event that we'll be covering is **`SDL_KEYDOWN`**. This event is basically triggered by a key on a keyboard being pressed all the way down. It's counterpart **`SDL_KEYUP`** is, conversely, triggered when a key is released. The usefulness of the first is probably pretty obvious, but `SDL_KEYUP` is also pretty handy when, for instance, a user clicks on the wrong thing by accident, and drags the mouse pointer away from that location to safely release it.

What's interesting about `SDL_KEYDOWN` is that, inside it's `SDL_EVENT` object itself contains information within it—namely, which key was pressed. And there's a **bunch** of these codes:

```
case SDL_KEYDOWN:
    switch (event.key.keysym.sym) {
        case SDLK_RIGHT:
            player_x += 1;
            break;
        case SDLK_SPACE:
            player_jump();
            break;
        // Etc...
    }
    break;
```

Code Block 4: A couple of keystroke examples. `K`, of course, stands for "key".

These two are used for "one-frame" keystrokes—great for shooting or jumping actions, but we also want options for when a player holds a key down over a period of several frames. Maybe they are running across a platform, or holding the jump button for a greater jump height. For that, we need to keep track of the keyboard's state.

Keyboard State

While checking for a keystroke is helpful for quick actions, there are occasions where we want to check the state of the **entire keyboard** i.e. check which keys are pressed. We actually have a pretty handy function that does this: `SDL_GetKeyboardState()`:

> Returns a pointer to an array of key states.
>
> A array element with a value of `1` means that the key is pressed and a value of `0` means that it is not. Indexes into this array are obtained by using `SDL_Scancode` values.
>
> The pointer returned is a pointer to an internal SDL array. It will be valid for the whole lifetime of the application and should not be freed by the caller.

— ***SDL Wiki***

```c
const Uint8 *key_state = SDL_GetKeyboardState(NULL); // if non-NULL,
receives the length of the returned array

if (key_state[SDL_SCANCODE_LEFT])
{
    player_left();
}

if (key_state[SDL_SCANCODE_RIGHT])
{
    player_right();
}

/* Etc... */
```

Code Block 5: For example. Notice that the above indices use the prefix `SDL_SCANCODE_` and not `SDLK_`.

We'll apply this, as well at the keystroke stuff from code block 4, in just a bit. Let's quickly cover mouse input first.

Part 3: *Mouse Input*

Just like there is an event for keyboard events, there is an event for mouse event that is only triggered when the mouse moves, or when one of its buttons is pressed:

```c
case SDL_MOUSEMOTION:
    // event.motion.x
    // event.motion.y
    // event.button.button
    break;
```

We can also **poll** the mouse for its current location. This isn't quite like polling for an event as we know it from the game-loop. This, just like the keyboard polling event, go **outside of the game loop**.

```c
int x, y;

SDL_GetMouseState(&x, &y);
```

This is a little tricky, and it gets more confusing when we consider what coordinate system the mouse follows: it is neither following a model's nor the world's coordinate system, but the **screen's**. This actually makes sense if you consider what a mouse should do. It should move according to axes that are independent of how the world is moving and/or changing. These coordinates are, thus, measured in pixels and follow the schema below:

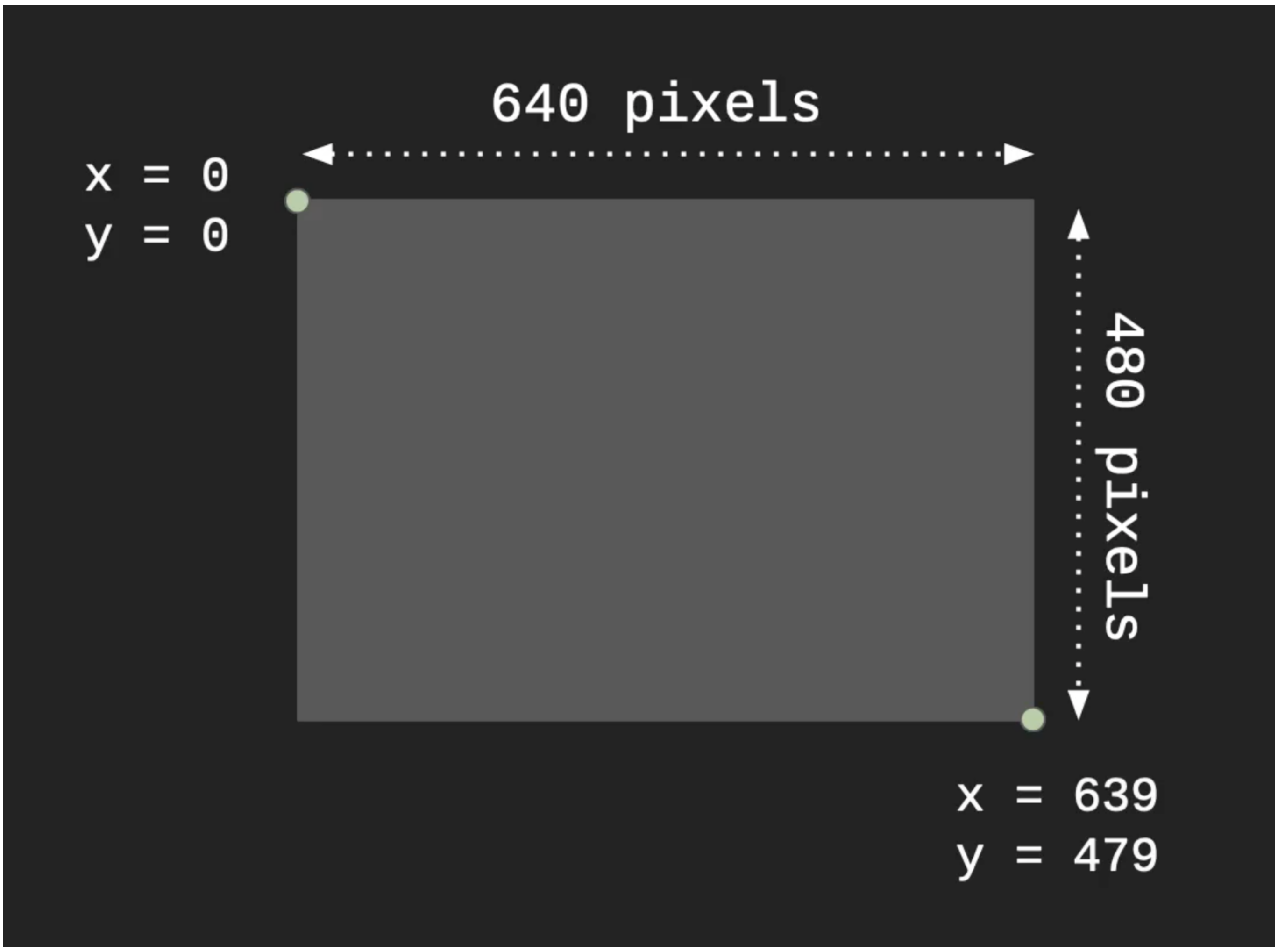

Figure 1: Mouse coordinates in our 640px by 480px screen.

This may be a little confusing, since we have been looking at our world according to our projection matrix, which covers a vertical and horizontal distance of only 10.0 by 7.5:

```
glm::ortho(-5.0f, 5.0f, -3.75f, 3.75f, -1.0f, 1.0f);
```

```
(0, 0)                  (-5.0, 3.75)
+-------------+               +------+------+
|             |               |      |      |
|             |               +------+------+
|             |               |      |      |
+-------------+               +------+------+
         (639, 479)                   (5.0, -3.75)
```

Figure 2: Screen vs world coordinates.

This necessitates that we translate our mouse coordinates on the screen to its coordinates in the world. Thankfully, there is a formula for that:

```
// Convert mouse x, y to world unit x, y

unit_x = ((x / width) * ortho_width) - (ortho_width / 2.0);
unit_y = (((height - y) / height) * ortho_height) - (ortho_height / 2.0);
```

We could make this into C++ functions for ease of use:

```
enum Coordinate { x_coordinate, y_coordinate };

const int WINDOW_WIDTH  = 640,
          WINDOW_HEIGHT = 480;

const Coordinate X_COORDINATE = x_coordinate;
const Coordinate Y_COORDINATE = y_coordinate;

const float ORTHO_WIDTH  = 7.5f,
            ORTHO_HEIGHT = 10.0f;

float get_screen_to_ortho(float coordinate, Coordinate axis)
{
    switch(axis)
    {
        case x_coordinate: return ((coordinate / WINDOW_WIDTH) *
ORTHO_WIDTH) - (ORTHO_WIDTH / 2.0);
        case y_coordinate: return (((WINDOW_HEIGHT - coordinate) /
WINDOW_HEIGHT) * ORTHO_HEIGHT) - (ORTHO_HEIGHT / 2.0);
        default          : return 0.0f;
    }
}
```

Code Block 6: A nice function using **enums** to translate screen to orthographic coordinates.

Part 4: *Controller Input*

Finally, for controller input we need to let OpenGL know that we have some additional hardware that it needs to take into consideration. This is done in the `initialisation()` function:

```
SDL_Joystick *playerOneController;

void initialise()
{
    // Initialise video AND the joystick subsystem
    SDL_Init(SDL_INIT_VIDEO | SDL_INIT_JOYSTICK);

    // Open the 1st controller found. Returns null on error
    playerOneController = SDL_JoystickOpen(0);
    // The rest of initialise...
}
```

Code Block 7: OpenGL's controller API.

Because the controller is technically outside of OpenGL's scope of control, we need to also shut down it down eventually. Since `SDL_JoystickOpen` returns a pointer, we need to release that memory in `shutdown()` with the following function. DON'T FORGET:

```
void shutdown()
{
    SDL_JoystickClose(playerOneController);
    SDL_Quit();
}
```

Code Block 8: Starting to look like a real function.

We can also check for the number of players using `SDL_NumJoysticks()`, and you can create as many `SDL_Joystick` as necessary.

Of course, just like with the keyboard and with the mouse, we have controller events:

- **`SDL_JOYAXISMOTION`**: One of the axes on the joystick has moved.
- **`SDL_JOYBUTTONDOWN`**: A joystick's button has been pressed.
- **`SDL_JOYBUTTONUP`**: A joystick's button has been released.

And a bunch of enumerations for the axes and buttons:

- **Axes**:
    - **0**: Left stick (x-axis)
    - **1**: Left stick (y-axis)
    - **3**: Right stick (x-axis)
    - **4**: Right stick (y-axis)
    - **2**: Left trigger
    - **5**: Right trigger
- **Buttons**:
    - **0**: A
    - **1**: B
    - **2**: X
    - **3**: Y
    - **4**: LB
    - **5**: RB
    - **6**: L3/LS
    - **7**: R3/RS
    - **8**: Start
    - **9**: Select

- **10**: Home
- **11**: D-pad up
- **12**: D-pad down
- **13**: D-pad left
- **14**: D-pad right

The reason why triggers are axes and not buttons is because OpenGL actually measures the amount of "press" you do on it. So if pressing the trigger all the way makes something go faster, for instance, you can poll for that using the triggers. The code for all of these looks as follows:

```
case SDL_JOYAXISMOTION:
    // event.jaxis.which    ; which controller (usually 0)
    // event.jaxis.axis     ; which axis
    // event.jaxis.value    ; -32768 to 32767
    break;
case SDL_JOYBUTTONDOWN:
    // event.jbutton.which  ; which controller
    // event.jbutton.button ; which button
    break;
```

Code Block 9: Incorporating joystick/controller controls into our polling. Note that this *does* go inside the game-loop.

And, of course, we can poll for sustained button or axis pressed like we do with keyboard strokes:

```
SDL_JoystickGetAxis(playerOneController, axisIndex);
SDL_JoystickGetButton(playerOneController, buttonIndex);
```

Code Block 10: Recall that these go *outside* the game loop.

Part 5: *Keeping Track Of The Player's Motion*

So far, we've been keeping track of our sprite's motion with simple `float` variables, but now it's time to get a little more finessed. Let's graduate to vectors:

```
// Start at 0, 0, 0
glm::vec3 player_position = glm::vec3(0, 0, 0);

// Don't go anywhere (yet)
glm::vec3 player_movement = glm::vec3(0, 0, 0);
```

The way we update these is inside `process_input()`, with code similar to code block 5:

```
const Uint8 *key_state = SDL_GetKeyboardState(NULL);

if (key_state[SDL_SCANCODE_LEFT])
```

```
{
    player_movement.x = -1.0f;
}
else if (key_state[SDL_SCANCODE_RIGHT])
{
    player_movement.x = 1.0f;
}
```

Code Block 11: Keeping track of *how much the player will move* during this frame.

And then, we update the player's position in `update()`:

```
void update()
{
    float ticks = (float) SDL_GetTicks() / MILLISECONDS_IN_SECOND; // get the current number of ticks
    float delta_time = ticks - previous_ticks; // the delta time is the difference from the last frame
    previous_ticks = ticks;

    // Add direction * units per second * elapsed time
    player_position += player_movement * player_speed * delta_time;

    model_matrix = glm::mat4(1.0f);
    model_matrix = glm::translate(model_matrix, player_position);
}
```

Code Block 11: Here, `player_speed` is how many units you want your player to move.

The last thing we have to take care of is to normalise the player movement at the end of `process_input()`. Normalisation, grossly oversimplified, makes sure than none of the values in `player_movement` are over `1.0f`. This can happen when players break a game and find a way to increase the amount of units they can move per second. This is just a simple OpenGL function:

```
// This makes sure that the player can't "cheat" their way into moving faster
if (glm::length(player_movement) > 1.0f)
{
    player_movement = glm::normalize(player_movement);
}
```