Day 4: Handling Input, Reset, Ball Bounce

Dylan <dylan@programvideogames.com> Thu, Jun 27, 2024 at 12:23 PM
To: brian.w.woodard@gmail.com

Welcome back to Programming Video Games.

By the end of this lesson, we'll have:

- Player paddle movement
- Ball collision and bounce
- Reset the game state

Player paddle movement

First up, our paddle is the wrong size and orientation.

To get it looking something like Pong, we'll first change the position and size.

But before we get to that, let's put the window size into a variable to make calculations that use it easier.

In fact, let's move all our game state into a new type to keep it organised.

After the imports and before main, create a new struct.

> A struct is used to group data together so it can be referred to as one "thing".

```
Game_State :: struct {
        window_size: rl.Vector2,
        paddle: rl.Rectangle,
        paddle_speed: f32,
        ball: rl.Rectangle,
        ball_dir: rl.Vector2,
        ball_speed: f32,
}
```

> Notice that in our struct we haven't set any values. That's because we are just defining the shape of the data. Setting the values will be done when we create our `Game_State` variable.

In main, we'll create our state variable and assign values:

```
main :: proc() {
        gs := Game_State {
                window_size = {1280, 720},
                paddle = {width = 30, height = 80},
                paddle_speed = 10,
                ball = {width = 30, height = 30},
                ball_speed = 10,
        }
        reset(&gs)

        using gs

        rl.InitWindow(i32(window_size.x), i32(window_size.y), "Pong")
        rl.SetTargetFPS(60)
        // ...
```

For `paddle` and `ball` we didn't set `x` or `y` because we're going to set them shortly in a new procedure called `reset`.

In `rl.InitWindow` we have to cast our window size values to `i32` now that we put them into a vector of f32s. The trade-off is any math that needs the window size won't need casting.

`reset(&gs)` - Using & before the name of a variable means the procedure can access the same instance.

> Usually, variables are copied into functions and the original isn't modified. There are rules that differ from language to language about what variables will be copied versus what will be "passed by reference".

`using gs` - `using` is a pretty interesting keyword. It basically creates a "shortcut" to get the values without *using* the prefix. So rather than type `gs.window_size.x` we can type `window_size.x` and the `gs.` becomes implicit.

I recommend sparing use of implicit functionality in programming. If one were to read just the loop of our function, they would have no idea where `window_size` is coming from until they read `using gs`

I just thought it'd be a nice feature to show off and it means we don't need to edit our code and add `gs.` in a bunch of places.

Reset the game state

Since we want to reset the game state to some default value each time, it's a good idea to wrap this in a procedure. It will be called from multiple places in code.

In order to use some procedures coming up, we'll have to import a few packages.

All of our imports:

```
import "core:math"
import "core:math/linalg"
import "core:math/rand"
import rl "vendor:raylib"
```

Then, above or below main, it doesn't matter: we create a new procedure:

```
reset :: proc(using gs: ^Game_State) {
        angle := rand.float32_range(-45, 46)
        r := math.to_radians(angle)

        ball_dir.x = math.cos(r)
        ball_dir.y = math.sin(r)

        ball.x = window_size.x / 2 - ball.width / 2
        ball.y = window_size.y / 2 - ball.height / 2

        paddle.x = window_size.x - 80
        paddle.y = window_size.y / 2 - paddle.height / 2
}
```

First, we'll discuss the single parameter: `using gs: ^Game_State`. Let's break it down.

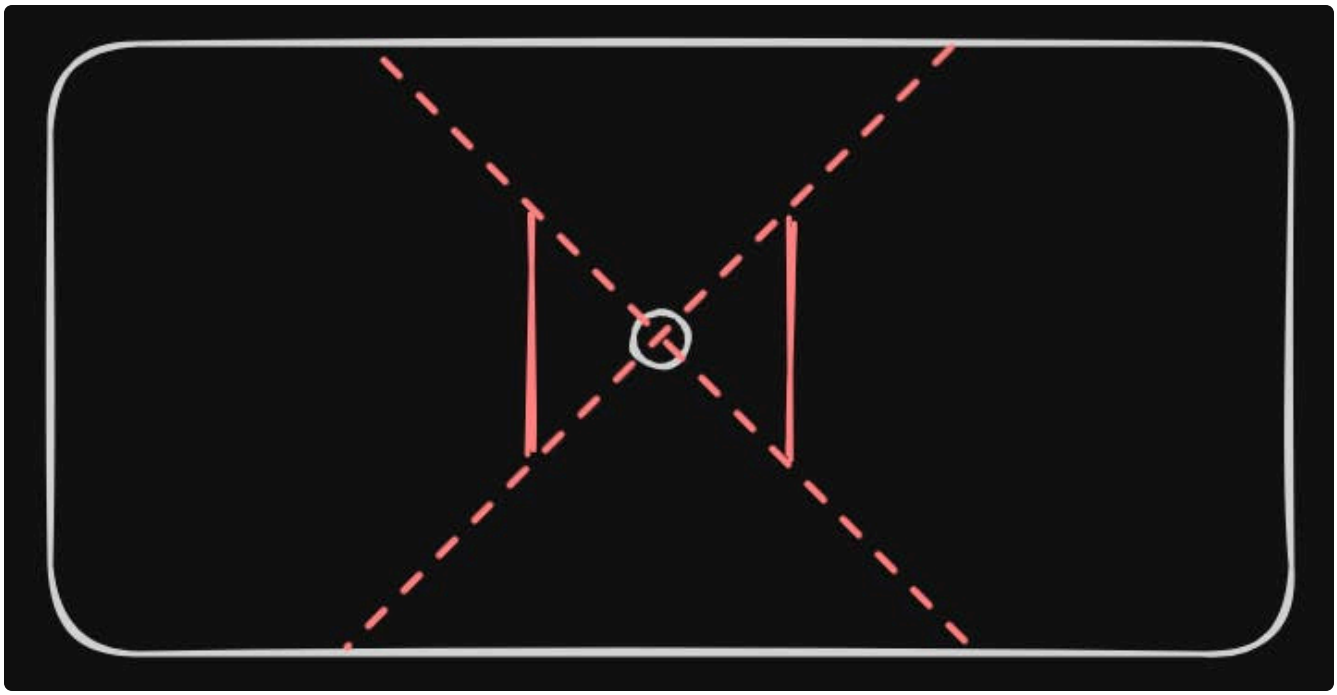
`using` - putting using here is the same as if we put `using gs` in the body of the procedure.

`gs:` - just like a struct, we put `<name>: <type>` in the parameter list of a procedure.

> Jargon: parameter list is what we define at the procedure, arguments are the values passed in when we call the procedure. You may see programmers refer to arguments as parameters and vice versa

`^Game_State` - This type means "pointer to a Game_State instance". Since we pass the Game_State in by reference, we need a pointer that tells us where the variable is.

We want our ball to travel in a random direction when it's spawned, but within a range that doesn't let it go too far up or down lest it bounce many times before a player gets to hit.



`angle := rand.float32_range(-45, 46)` - get a value between -45 and 45 (the top of the range is exclusive so we must add 1 if we want the full range). `r := math.to_radians(angle)` - convert degrees to radians, a more common way of representing angles in games.

To get the direction in a vector form so we can use it, we need to use a bit of simple trigonometry.

```
ball_dir.x = math.cos(r); ball_dir.y = math.sin(r)
```

> Trigonometry and linear algebra are the two most useful branches of mathematics for game programming in my experience. You don't need to know all the formulas by heart or anything - "just in time" learning is good enough if you can't remember this stuff from school.

Finally, we set the ball position to the centre, offset by the size / 2 as it's origin is top left.

Then, for the paddle we do the offset for y and a flat -80 for x. You could change x be related to the paddle width but I didn't see the need.

If we run the game now, we'll see a few issues.

1. Our paddle moves the wrong way
2. Our paddle can move off screen
3. The ball doesn't move horizontally

Paddle moves the wrong way

A simple fix, and an opportunity to show another language feature.

```
if rl.IsKeyDown(.UP) {
        paddle.y -= paddle_speed
}
if rl.isKeyDown(.DOWN) {
        paddle.y += paddle_speed
}
```

Here we've changed the keys and used a shortcut for enums.

Rather than `rl.KeyboardKey.UP` we can simply use `.UP` .

The compiler will infer the type as the `rl.IsKeyDown` procedure takes a `rl.KeyboardKey` enum.

Otherwise, we simply swap `.x` to `.y` .

Paddle can move off the screen

We'll add one line below the input code:

`paddle.y = linalg.clamp(paddle.y, 0, window_size.y - paddle.height)` - `linalg.clamp` is a procedure that takes a value, a minimum and a maximum. If the value is below the minimum, the minimum is returned. If the value is above the maximum, the maximum is returned. Otherwise, the value is returned unchanged.

Ball doesn't move horizontally

Simply duplicate the lines that update the ball position and `next_ball_rect` and add the X axis.

```
next_ball_rect.x += ball_speed * ball_dir.x
next_ball_rect.y += ball_speed * ball_dir.y
// ...
ball.x += ball_speed * ball_dir.x
ball.y += ball_speed * ball_dir.y
```

With these problems fixed, if we run the game we'll see that the ball flies through the paddle.

As well as that, we haven't called our reset procedure when the ball goes off screen.

Reset

Missing the ball and having to close and open the program to test is a pain.

Let's add some simple logic to reset the game state when the ball goes off screen on either side.

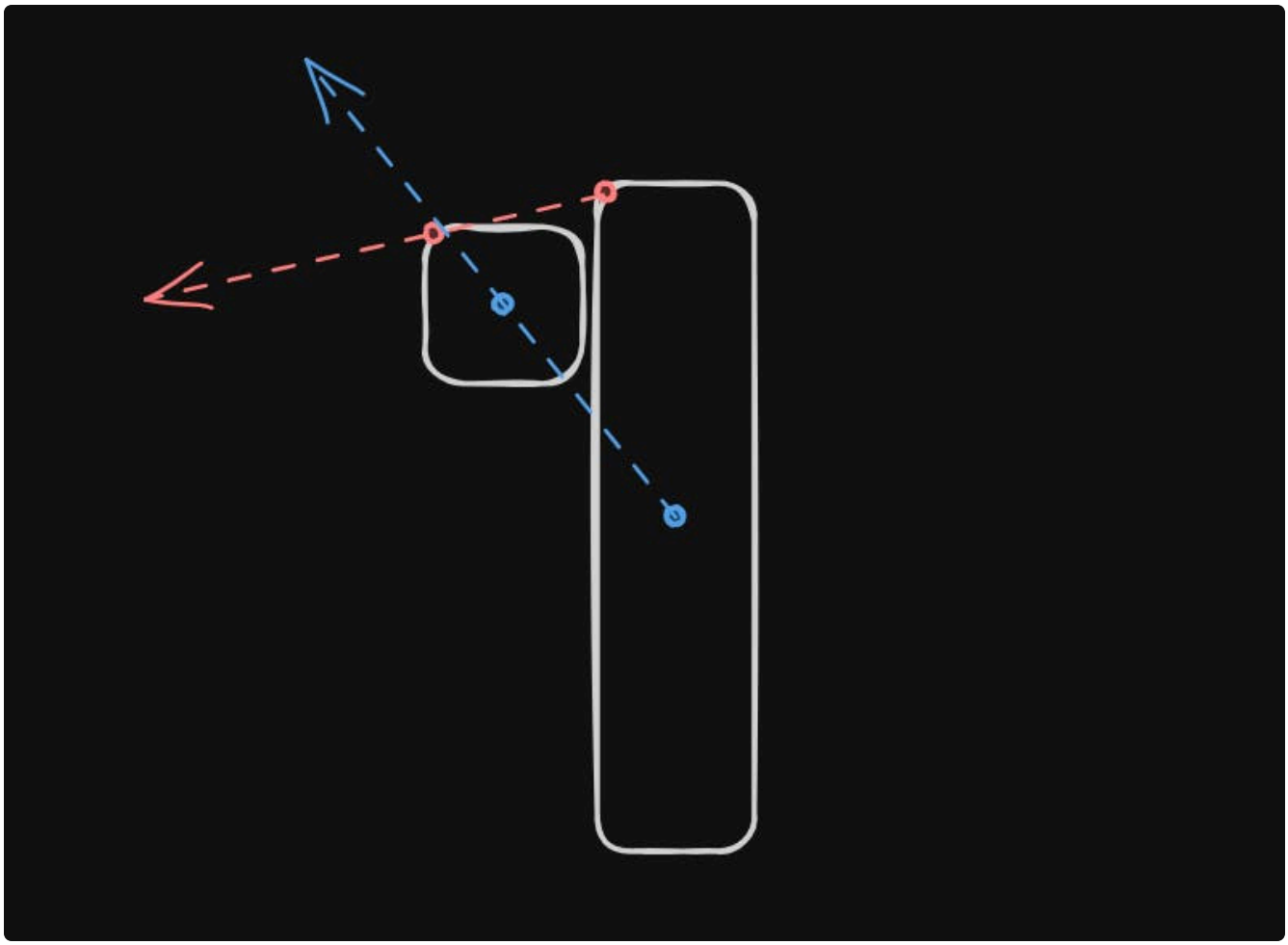
```
if next_ball_rect.x >= window_size.x - ball.width {
        reset(&gs)
}

if next_ball_rect.x < 0 {
        reset(&gs)
}
```

Collisions, again

Let's make our collision more robust. It won't be perfect, if you were to increase the speed of the ball too much there would be issues. But, it's good enough for now.

```
if rl.CheckCollisionRecs(next_ball_rect, paddle) {
        ball_center := rl.Vector2 {
                next_ball_rect.x + ball.width / 2,
                next_ball_rect.y + ball.height / 2,
        }
        paddle_center := rl.Vector2{paddle.x + paddle.width / 2, paddle.y + paddle.height /
2}
        ball_dir = linalg.normalize0(ball_center - paddle_center)
}
```

There is a bit going on here that would be better explained with a picture... So, let me draw you one.



If we used the origins of the ball and the rectangle for our vector math, we'd end up with the red line as our direction.

Instead, what we want is to figure out the centre of each shape and use those, then we end up with the blue line.

If the ball hits the exact middle of the paddle, it will be sent straight across the screen.

A crude solution, and not realistic, but it gives the player control over the direction of the ball.

So, first we find the centre positions, marked in blue.

Subtract the ball's centre from the paddle's to get the direction.

If we left it there, we'd increase the speed of the ball by the distance between the two points…

To prevent that, we use a procedure called `normalize0` - it normalises the length of a vector to 1.

If the direction is, for example, exactly northwest, then the direction vector would be -0.5, -0.5.

If the direction is straight down, it'd be 0, 1.

Playing the game now we can see that the ball is bouncing as expected.

It'll even bounce off the top or bottom of the paddle.

Let's quickly go over the main Input→Processing→Output cycle.

Now that we have game state, you can see what I meant by game state being part of the input.

Input: Game state, keyboard buttons

Processing: Our entire game loop

Output: What's displayed on the screen

This lesson was a bit of a long one - we are nearly there!

Next lesson we'll be wrapping this up with AI, a score system, and sound effects.

If you have any questions, make sure to check out the Discord server.

How do you feel about the current length of the daily lessons?

- ( ) They are just right
- ( ) I'd prefer shorter lessons
- ( ) I'd prefer longer lessons
- ( ) I'm struggling to keep up
- ( ) I can take on more content each day

Here's the code so far:

```
package main

import "core:math"
import "core:math/linalg"
import "core:math/rand"
import rl "vendor:raylib"

Game_State :: struct {
	window_size:  rl.Vector2,
	paddle:       rl.Rectangle,
	paddle_speed: f32,
	ball:         rl.Rectangle,
	ball_dir:     rl.Vector2,
	ball_speed:   f32,
}

main :: proc() {
	gs := Game_State {
		window_size = {1280, 720},
		paddle = {width = 30, height = 80},
		paddle_speed = 10,
		ball = {width = 30, height = 30},
		ball_speed = 10,
	}
	reset(&gs)

	using gs

	rl.InitWindow(i32(window_size.x), i32(window_size.y), "Pong")
	rl.SetTargetFPS(60)
```

```
	for !rl.WindowShouldClose() {
		if rl.IsKeyDown(.UP) {
			paddle.y -= paddle_speed
		}
		if rl.IsKeyDown(.DOWN) {
			paddle.y += paddle_speed
		}

		paddle.y = linalg.clamp(paddle.y, 0, window_size.y - paddle.height)

		next_ball_rect := ball
		next_ball_rect.x += ball_speed * ball_dir.x
		next_ball_rect.y += ball_speed * ball_dir.y

		if next_ball_rect.y >= 720 - ball.height || next_ball_rect.y <= 0 {
			ball_dir.y *= -1
		}

		if next_ball_rect.x >= window_size.x - ball.width {
			reset(&gs)
		}

		if next_ball_rect.x < 0 {
			reset(&gs)
		}

		if rl.CheckCollisionRecs(next_ball_rect, paddle) {
			ball_center := rl.Vector2 {
				next_ball_rect.x + ball.width / 2,
				next_ball_rect.y + ball.height / 2,
			}
			paddle_center := rl.Vector2{
				paddle.x + paddle.width / 2,
				paddle.y + paddle.height / 2
			}
			ball_dir = linalg.normalize0(ball_center - paddle_center)
		}

		ball.x += ball_speed * ball_dir.x
		ball.y += ball_speed * ball_dir.y

		rl.BeginDrawing()

		rl.ClearBackground(rl.BLACK)

		rl.DrawRectangleRec(paddle, rl.WHITE)
		rl.DrawRectangleRec(ball, rl.RED)

		rl.EndDrawing()
	}
}

reset :: proc(using gs: ^Game_State) {
	angle := rand.float32_range(-45, 46)
	r := math.to_radians(angle)

	ball_dir.x = math.cos(r)
	ball_dir.y = math.sin(r)

	ball.x = window_size.x / 2 - ball.width / 2
	ball.y = window_size.y / 2 - ball.height / 2

	paddle.x = window_size.x - 80
```

```
        paddle.y = window_size.y / 2 - paddle.height / 2
}
```