Day 5: Putting It All Together: Building a Simple Game

Dylan <dylan@programvideogames.com> Fri, Jun 28, 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 covered:

- AI Player
- Score
- Sound effects
- Bonus: Boost mechanic

AI Player

The first thing we need to do is give the AI it's own paddle.

We'll add a filed to the game state to represent it:

```
Game_State :: struct {
        window_size: rl.Vector2,
        paddle: rl.Rectangle,
        ai_paddle: rl.Rectangle,
        // ...
}
```

Now we'll want to initialise this new paddle with the size:

```
gs := Game_State {
        window_size = {1280, 720},
        paddle = {width = 30, height = 80},
        ai_paddle = {width = 30, height = 80},
        // ...
}
```

The paddle needs to be drawn to the screen, so head down to the main loop:

```
rl.DrawRectangleRec(paddle, rl.WHITE)
rl.DrawRectangleRec(ai_paddle, rl.WHITE)
```

Finally, the position needs to be reset.

While we are here, we'll put the paddle margin into a variable in case you want to play around with sizes.

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

```
        if rand.int_max(100) % 2 == 0 do angle += 180 // <---- also a new line
        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_margin: f32 = 50

        paddle.x = window_size.x - (paddle.width + paddle_margin)
        paddle.y = window_size.y / 2 - paddle.height / 2

        ai_paddle.x = paddle_margin
        ai_paddle.y = window_size.y / 2 - ai_paddle.height / 2
}
```

`if rand.int_max(100) % 2 == 0 do angle += 180` - Since we are adding a 2nd player, we want them to sometimes get served the ball first.

To do that we'll generate a random number and use the modulo operator `%` to check if it's an even number. Basically giving us a coin toss. Then we add 180 degrees to flip the direction.

Play the game now and you should see another paddle on the left side, and the ball should sometimes spawn and travel to the left.

Next up, we want the AI player to try to hit the ball.

Your first instinct may be to move the paddle towards the ball each frame.

However, this creates an unbeatable player!

We need to add some way for the computer to fail.

Let's give it a try anyway, just to see what happens. We want to see why it doesn't work and also doesn't look good.

```
paddle.y = linalg.clamp(paddle.y, 0, window_size.y - paddle.height)
// new code below
diff := ai_paddle.y + ai_paddle.height / 2 - ball.y + ball.height / 2
if diff < 0 {
        ai_paddle.y += paddle_speed
}
if diff > 0 {
        ai_paddle.y -= paddle_speed
}

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

If you run the game now, you'll see a flickering paddle that follows the ball accurately.

We haven't got collisions working for the AI, so let's add those.

At the same time I'll show you a cool language feature: `or_else` .

Since our collision code will be the same for each paddle, we'll move it into a procedure.

```
ball_dir_calculate :: proc(ball: rl.Rectangle, paddle: rl.Rectangle) -> (rl.Vector2, bool) {
        if rl.CheckCollisionRecs(ball, paddle) {
                ball_center := rl.Vector2{ball.x + ball.width / 2, ball.y + ball.height / 2}
                paddle_center := rl.Vector2{paddle.x + paddle.width / 2, paddle.y +
paddle.height / 2}
                return linalg.normalize0(ball_center - paddle_center), true
        }
        return {}, false
}
```

This is our first procedure that has a return type: `(rl.Vector2, bool)`.

There's actually two types here as Odin allows multiple return values from a procedure.

The `bool` tells us whether or not the ball direction should change.

The code is almost the same as our previous code from the loop, we just move it down here.

If the ball is colliding with the paddle, we do the calculation and return the new direction and true.

If not, we return a "zero-value" using `{}`.

> I only use `{}` for structs and arrays, for pointers I prefer `nil` , for numbers `0` , for strings `""` and for bools `false`.

In this case, it's the same as writing `return rl.Vector2{0, 0}, false` .

Now in our loop code, we want to update the ball's direction.

I'll show you three ways to write the same thing:

```
// 1
ball_dir = ball_dir_calculate(next_ball_rect, paddle) or_else ball_dir

// 2
new_dir, did_hit := ball_dir_calculate(next_ball_rect, paddle)
if did_hit {
        ball_dir = new_dir
}

// 3
new_dir, did_hit := ball_dir_calculate(next_ball_rect, paddle); did_hit {
        ball_dir = new_dir
}
```

If you have used the Go programming language, then 2 and 3 are probably familiar to you.

What happens in #1 is we set the direction to the new value if the bool returned is true, otherwise we set it to the same value it is right now.

Duplicate that for `ai_paddle` and we have collisions working for both paddles:

```
ball_dir = ball_dir_calculate(next_ball_rect, paddle) or_else ball_dir
ball_dir = ball_dir_calculate(next_ball_rect, ai_paddle) or_else ball_dir
```

Running the game now, we are facing a flickering, unbeatable opponent!

Better feeling and looking AI

There are a number of ways to achieve AI that seems a bit fairer.

The simplest way is to give the AI player's paddle a lower movement speed.

Simply multiply the `paddle_speed` by `0.5` in the movement code:

```
diff := ai_paddle.y + ai_paddle.height / 2 - ball.y + ball.height / 2
if diff < 0 {
        ai_paddle.y += paddle_speed * 0.5
}
if diff > 0 {
        ai_paddle.y -= paddle_speed * 0.5
}
```

Playing the game now, we can actually win!

It's not the best AI - it's easily beatable by bouncing the ball at sharp angles rather than shooting straight on.

To make it act a little more human, we're going to add two things:

1. A delay to its movement
2. A bit of inaccuracy

```
Game_State :: struct {
        // ...
        ai_target_y: f32,
        ai_reaction_delay: f32,
        ai_reaction_counter: f32,
}

// in main
gs := Game_State {
        // ...
        ai_reaction_delay = 0.1,
}
```

Then we'll update our AI code:

```
// AI movement
// increase timer by time between last frame and this one
ai_reaction_timer += rl.GetFrameTime()
// if the timer is done:
if ai_reaction_timer >= ai_reaction_delay {
        // reset the timer
        ai_reaction_timer = 0
        // use ball from last frame for extra delay
        ball_mid := ball.y + ball.height / 2
        // if the ball is heading left
        if ball_dir.x < 0 {
                // set the target to the ball
                ai_target_y = ball_mid - ai_paddle.height / 2
                // add or subtract 0-20 to add inaccuracy
                ai_target_y += rand.float32_range(-20, 20)
        } else {
                // set the target to screen middle
                ai_target_y = window_size.y / 2 - ai_paddle.height / 2
        }
}

// calculate the distance between paddle and target
ai_paddle_mid := ai_paddle.y + ai_paddle.height / 2
target_diff := ai_target_y - ai_paddle.y
// move either paddle_speed distance or less
// won't bounce around so much
ai_paddle.y += linalg.clamp(target_diff, -paddle_speed, paddle_speed) * 0.65
// clamp to window_size
ai_paddle.y = linalg.clamp(ai_paddle.y, 0, window_size.y - ai_paddle.height)
```

This time I opted for adding comments within the code itself.

Comments inside the code block

I don't mind

Running the game now, I think the opponent feels much better to play against.

It moves perfectly back to the centre between hits, but that's fine. It's beatable and seems pretty realistic.

Score

Let's add a simple scoring mechanism to the game.

Raylib comes with text rendering built in, so first we'll add two fields to our game state.

```
Game_State :: struct {
        // ...
        score_player: int,
        score_cpu; int,
}
```

Now, down in our code where we detect if the ball has gone off the screen, we need to update the score.

```
if next_ball_rect.x >= window_size.x - ball.width {
        score_cpu += 1
        reset(&gs)
}

if next_ball_rect.x < 0 {
        score_player += 1
        reset(&gs)
}
```

Finally, we should draw the scores onto the screen.

Down in the drawing code, add these lines:

```
rl.DrawText(fmt.ctprintf("{}", score_cpu), 12, 12, 32, rl.WHITE)
rl.DrawText(fmt.ctprintf("{}", score_player), i32(window_size.x) - 28, 12, 32, rl.WHITE)
```

What is `fmt.ctprintf` ?

If you remember back in lesson 1 we used `fmt.println` to print text to the terminal.

In this case, we're printing text to "temporary storage" (the t).

The small c means it'll be a C-String - a string compatible with the C programming language which Raylib expects.

So, `ct` is C + temporary storage.

What is "{}"?

`ctprintf` takes a variable amount of arguments. A format is first, followed by any number of variables to be used in the string.

`{}` in this case tells the compiler to automatically figure out how to convert the variable into a string representation.

You can print structs, arrays, and all kinds of stuff and the compiler will usually figure out a good representation.

If it can't, there are specific codes you can look up here:

```
fmt.ctprintf("{} -> {}", 42, 23)
// => "42 -> 23"
```

Temporary Storage

There is an invisible struct that gets passed into each procedure of Odin programs.

It's called the context and it has a few fields, one of which is `temp_allocator`.

We won't go further into memory allocation in this series, but what we need to understand is:

- `temp_allocator` is designed for "short-lived" memory allocations. Exactly like this score string that we are creating every frame.
- Given that we are creating a string every frame, more memory will be used as the game runs. This is a memory leak.

To remedy the memory leak, we can simply call `free_all(context.temp_allocator)` at the end of our loop.

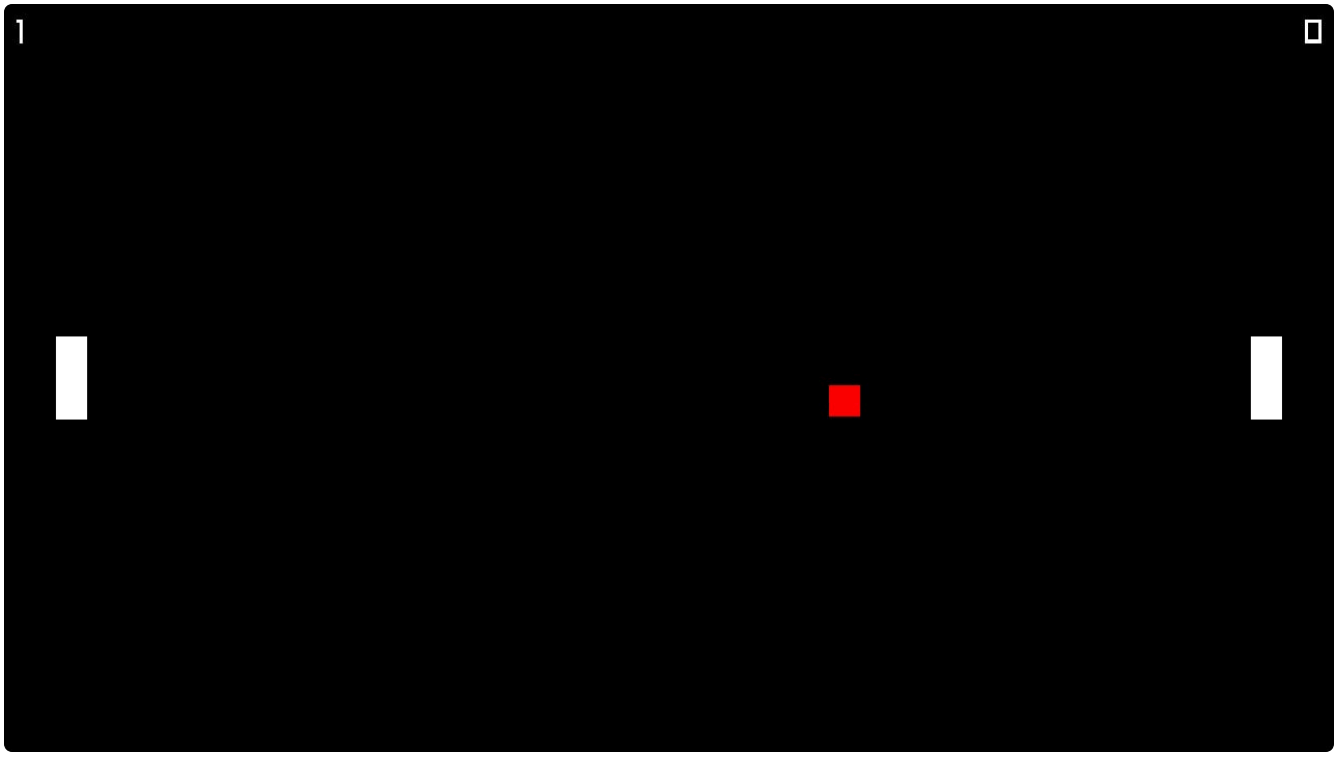
```
rl.EndDrawing()
free_all(context.temp_allocator)
```

This allocator is designed to be very cheap to allocate and free memory - and it's designed to free all memory at once.

Perfect for these kinds of use-cases in which we need some memory for one operation but don't want to worry about it.

> There are a bunch of different ways to create strings in the `fmt` package. I recommend checking it out.

Playing the game now should show a score in each corner, as such:



Sound Effects

You can download the sound effects from here: https://github.com/Falconerd/programvideogames

Oft times the final thing to be added by programmers is sound effects. And this project is no exception.

I have prepared some simple 8-bit sound effects for us to use, available here: (sfx link)

Raylib comes with sound capabilities so this will be easy. No manual audio mixing required.

Just after we initialise and set our target FPS, we can initialise our audio device.

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

rl.InitAudioDevice()
defer rl.CloseAudioDevice()
```

`defer` - You'll notice this is the first time we have used this defer keyword. What it does is defer the execution of this line until the end of the scope.

The scope is defined by pairs of braces, so in this case it's like this:

```
main :: proc() { // <-- main scope starts
        // ...
        for ... { // <-- loop scope starts
        } // <-- loop scope ends
} // <-- main scope ends
```

Since we are inside `main` our deferred line will get called after the loop, before our program exits.

I like using defer so I can put the opening and closing together - makes it less likely to forget.

So, we have an initialised audio device, now we need to load our sound effects.

On the next few lines, define these variables:

```
sfx_hit := rl.LoadSound("hit.wav")
sfx_win := rl.LoadSound("win.wav")
sfx_lose := rl.LoadSound("lose.wav")
```

You can put them into a separate folder (resources, perhaps?) if you like.

For this simple program I just leave them in the same folder as pong.odin.

Now, to play the sounds... Let's first add them to when the ball goes off the screen.

```
if next_ball_rect.x >= window_size.x - ball.width {
        score_cpu += 1
        rl.PlaySound(sfx_lose)
        reset(&gs)
}

if next_ball_rect.x < 0 {
        score_player += 1
        rl.PlaySound(sfx_win)
        reset(&gs)
}
```

Now, to add the sound for when the ball is hit by the paddle...

The code we have at the moment looks like this:

```
ball_dir = ball_dir_calculate(next_ball_rect, paddle) or_else ball_dir
ball_dir = ball_dir_calculate(next_ball_rect, ai_paddle) or_else ball_dir
```

We probably don't want to put the `rl.PlaySound` call inside the `ball_dir_calculate` procedure.

What does playing a sound have to do with calculating a direction?

We could do a number of things.

I'll give you two options and you choose the one you prefer, or come up with your own solution.

1. We could save the `ball_dir` before these lines then check if it changed.

```
last_ball_dir := ball_dir
// ...
if last_ball_dir != ball_dir {
        rl.PlayRound(sfx_hit)
}
```

2. We could use one of the other patterns instead of `or_else`

```
if new_dir, ok := ball_dir_calculate(next_ball_rect, paddle); ok {
        ball_dir = new_dir
        rl.PlaySound(sfx_hit)
} else if new_dir, ok := ball_dir_calculate(next_ball_rect, ai_paddle); ok {
        ball_dir = new_dir
        rl.PlaySound(sfx_hit)
}
```

That's it for sound effects!

Let's move on to the final part of this tutorial.

Boost mechanic

Playing the game now... It's a bit dull.

I thought it'd be cool to add another timing mechanic to the game.

Mechanic: By pressing the spacebar when the ball is hitting our paddle, we can increase the ball's velocity.

First thing's first - let's add a boost timer to the game state:

```
Game_State :: struct {
        // ...
        boost_timer: f32,
}
```

Now, since we have two timers, it's prudent to store the time between frames in a variable rather than call `rl.GetFrameTIme` each time.

> I looked into it and GetFrameTime is just a lookup on a struct. So, not that expensive but still an extra procedure call.

```
for !rl.WindowShouldClose() {
        delta := rl.GetFrameTime()

        boost_timer -= delta

        // ...

        ai_reaction_timer += delta
```

> Yeah, one of them is counting down and one up. This will make some math we're doing later a bit simpler.

Now we want to check if the player presses the boost key and start the countdown.

```
// Just after other input code:
if rl.IsKeyPressed(.SPACE) {
        if boost_timer < 0 {
                boost_timer = 0.2
        }
}
```

In order to increase the speed of the ball, we can update the `ball_dir` on hit.

We award more speed the closer the timer is full.

If you chose style 1 above:

```
last_ball_dir := ball_dir
// expand this case out
new_dir, did_hit := ball_dir_calculate(next_ball_rect, paddle)
if did_hit {
        // if the button was pressed in the last 0.2 seconds
        if boost_timer > 0 {
                // boost_timer / 0.2 will give us a percentage (let's say 30%)
                // we add 1 because we want to increase the speed (130%)
                d := 1 + boost_timer / 0.2
                new_dir *= d
        }
        ball_dir = new_dir
}
ball_dir = ball_dir_calculate(next_ball_rect, ai_paddle) or_else ball_dir
if last_ball_dir != ball_dir {
        rl.PlayRound(sfx_hit)
}
```

If you chose style 2 above:

```
if new_dir, ok := ball_dir_calculate(next_ball_rect, paddle); ok {
        // if the button was pressed in the last 0.2 seconds
        if boost_timer > 0 {
                // boost_timer / 0.2 will give us a percentage (let's say 30%)
                // we add 1 because we want to increase the speed (130%)
                d := 1 + boost_timer / 0.2
                new_dir *= d
        }
        ball_dir = new_dir
```

```
        rl.PlaySound(sfx_hit)
} else if ... // same as before
```

The final thing to do is to add is some visual confirmation.

1. We'll make the ball turn red → yellow depending on the speed.
2. We'll make the paddle flash cyan and fade back to white over 0.2s when space is pressed.

```
if boost_timer > 0 {
        rl.DrawRectangleRec(paddle, {u8(255 * (0.2 / boost_timer)), 255, 255, 255})
} else {
        rl.DrawRectangleRec(paddle, rl.WHITE)
}
// ...
rl.DrawRectangleRec(ball, {255, u8(255 - 255 / linalg.length(ball_dir)), 0, 255})
```

Run the game and try it out now.

I think it's much better, if a bit easy.

You could adjust the AI movement speed penalty to account for the new ball speed.

That concludes the last lesson of this mini-course.

We touched on a number of vital topics for programming games, including:

- Rendering
- Input handling
- Playing sounds
- Collisions
- Rudimentary score system
- Game loops
- Procedures
- Game state

Each of these could have an entire course dedicated to them.

There's a lot to learn, but I hope you can see that it's not impossible to get something going in a short time.

I bet you could recreate Pong in a couple of hours or less now!

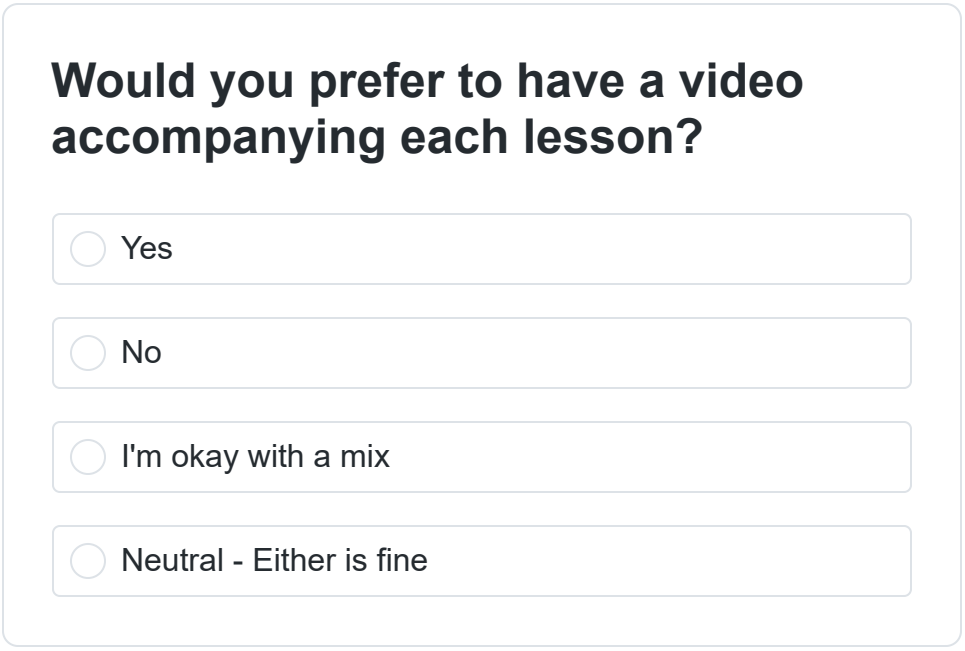
If you are confused about anything in this course, please come and ask me questions in the Discord server.

Keep an eye out for new content on my YouTube channel as I'll be covering more game programming topics.

I wish you all the best, sincerely, in your game programming endeavours.

Cheers,

— Dylan



Here's the code I have at the end:

```
package main

import "core:fmt"
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,
        ai_paddle:         rl.Rectangle,
        paddle_speed:      f32,
        ball:              rl.Rectangle,
        ball_dir:          rl.Vector2,
        ball_speed:        f32,
        ai_target_y:       f32,
        ai_reaction_delay: f32,
        ai_reaction_timer: f32,
        score_player:      int,
        score_cpu:         int,
        boost_timer:       f32,
}

main :: proc() {
        gs := Game_State {
                window_size = {1280, 720},
                paddle = {width = 30, height = 80},
                ai_paddle = {width = 30, height = 80},
```

```
        paddle_speed = 10,
        ball = {width = 30, height = 30},
        ball_speed = 10,
        ai_reaction_delay = 0.1,
}
reset(&gs)

using gs

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

rl.InitAudioDevice()
defer rl.CloseAudioDevice()

sfx_hit := rl.LoadSound("hit.wav")
sfx_win := rl.LoadSound("win.wav")
sfx_lose := rl.LoadSound("lose.wav")

for !rl.WindowShouldClose() {
        delta := rl.GetFrameTime()

        boost_timer -= delta

        // Player movement
        if rl.IsKeyDown(.UP) {
                paddle.y -= paddle_speed
        }

        if rl.IsKeyDown(.DOWN) {
                paddle.y += paddle_speed
        }

        if rl.IsKeyPressed(.SPACE) {
                if boost_timer < 0 {
                        boost_timer = 0.2
                }
        }

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

        // AI movement
        // increase timer by time between last frame and this one
        ai_reaction_timer += delta
        // if the timer is done:
        if ai_reaction_timer >= ai_reaction_delay {
                // reset the timer
                ai_reaction_timer = 0
                // use ball from last frame for extra delay
                ball_mid := ball.y + ball.height / 2
                // if the ball is heading left
                if ball_dir.x < 0 {
                        // set the target to the ball
                        ai_target_y = ball_mid - ai_paddle.height / 2
                        // add or subtract 0-20 to add inaccuracy
                        ai_target_y += rand.float32_range(-20, 20)
                } else {
                        // set the target to screen middle
                        ai_target_y = window_size.y / 2 - ai_paddle.height / 2
                }
        }

        // calculate the distance between paddle and target
        ai_paddle_mid := ai_paddle.y + ai_paddle.height / 2
        target_diff := ai_target_y - ai_paddle.y
```

```
                // move either paddle_speed distance or less
                // won't bounce around so much
                ai_paddle.y += linalg.clamp(target_diff, -paddle_speed, paddle_speed) * 0.65
                // clamp to window_size
                ai_paddle.y = linalg.clamp(ai_paddle.y, 0, window_size.y - ai_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 {
                        score_cpu += 1
                        rl.PlaySound(sfx_lose)
                        reset(&gs)
                }

                if next_ball_rect.x < 0 {
                        score_player += 1
                        rl.PlaySound(sfx_win)
                        reset(&gs)
                }

                if new_dir, ok := ball_dir_calculate(next_ball_rect, paddle); ok {
                        if boost_timer > 0 {
                                d := 1 + boost_timer / 0.2
                                new_dir *= d
                        }
                        ball_dir = new_dir
                        rl.PlaySound(sfx_hit)
                } else if new_dir, ok := ball_dir_calculate(next_ball_rect, ai_paddle); ok {
                        ball_dir = new_dir
                        rl.PlaySound(sfx_hit)
                }

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

                rl.BeginDrawing()

                rl.ClearBackground(rl.BLACK)

                if boost_timer > 0 {
                        rl.DrawRectangleRec(paddle, {u8(255 * (0.2 / boost_timer)), 255, 255,
255})
                } else {
                        rl.DrawRectangleRec(paddle, rl.WHITE)
                }
                rl.DrawRectangleRec(ai_paddle, rl.WHITE)
                rl.DrawRectangleRec(ball, {255, u8(255 - 255 / linalg.length(ball_dir)), 0,
255})

                rl.DrawText(fmt.ctprintf("{}", score_cpu), 12, 12, 32, rl.WHITE)
                rl.DrawText(fmt.ctprintf("{}", score_player), i32(window_size.x) - 28, 12,
32, rl.WHITE)

                rl.EndDrawing()
                free_all(context.temp_allocator)
        }
}

ball_dir_calculate :: proc(ball: rl.Rectangle, paddle: rl.Rectangle) -> (rl.Vector2, bool) {
```

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

reset :: proc(using gs: ^Game_State) {
        angle := rand.float32_range(-45, 46)
        if rand.int_max(100) % 2 == 0 do angle += 180
        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_margin: f32 = 50

        paddle.x = window_size.x - (paddle.width + paddle_margin)
        paddle.y = window_size.y / 2 - paddle.height / 2

        ai_paddle.x = paddle_margin
        ai_paddle.y = window_size.y / 2 - ai_paddle.height / 2
}
```