Day 3: Simple collisions, multiple moving objects

Dylan <dylan@programvideogames.com> Wed, Jun 26, 2024 at 12:22 PM
To: brian.w.woodard@gmail.com

Welcome back! By the end of this lesson you will have learnt:

- How to handle multiple moving objects
- How to handle colliding objects

Multiple moving objects

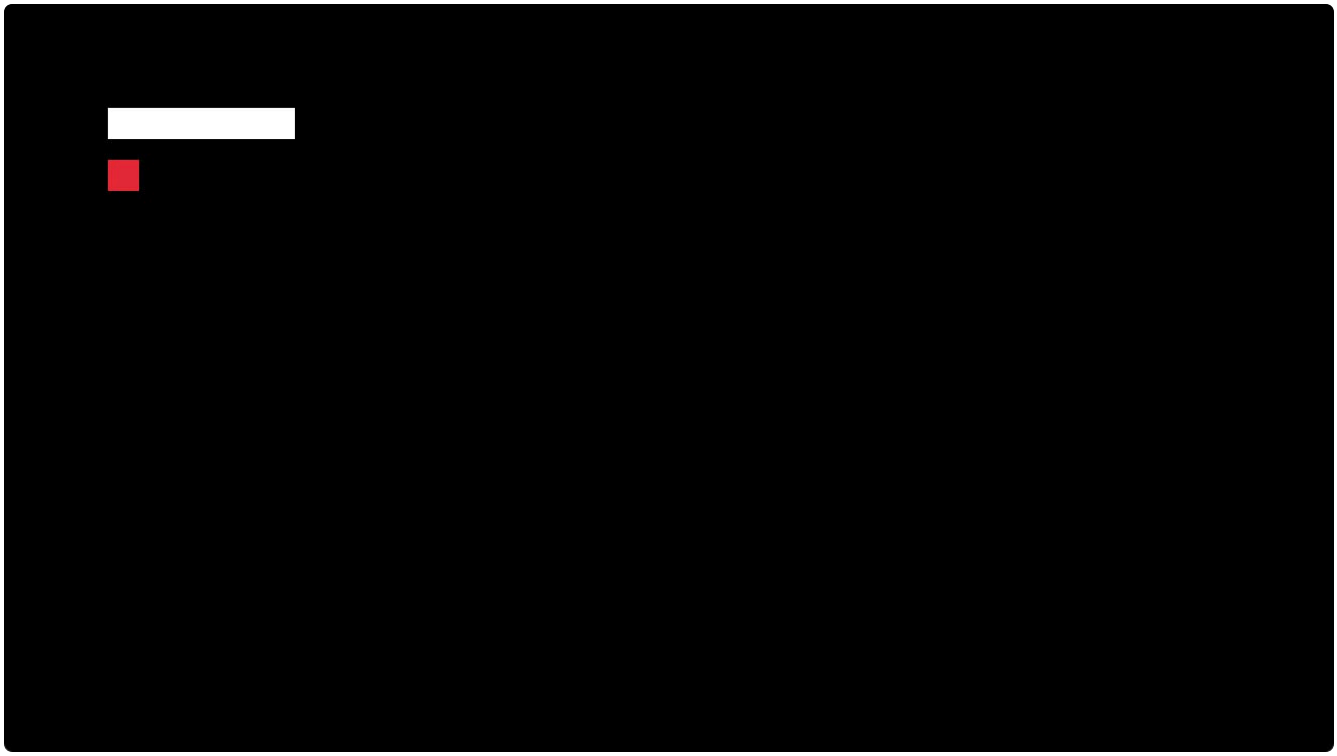
Let's introduce another object to the scene.

For this simple program, the solution is very straightforward - add another variable.

Where we declare `pos` we want another variable `pos2: i32 = 150`.

Just after our first rectangle, add `rl.DrawRectangle(100, pos2, 30, 30, rl.RED)`

You should see something like this when you run:



Rather than have the red square controlled by user input, let's make it move on it's own.

Just add `pos2 += 10` into the main loop before drawing.

Running the game, the red square will fall off the bottom of the screen.

What if instead we wanted it to change direction when it hits the edge of the screen?

Let's introduce another variable: `direction: i32 = 1`

Now, we want to multiply `pos2` by this direction: `pos2 += 10 * direction` .

Finally, we want to swap the direction to -1 when it hits the edge of the screen:

```
// after input, before drawing
next_pos2 := pos2 + 10 * direction
if next_pos2 >= 720 - 30 || next_pos2 <= 0 {
        direction *= -1 // flips -1 to 1 or 1 to -1
}
pos2 += 10 * direction
```

`next_pos2 := pos2 + 10 * direction` - store the next position in a variable

`if next_pos2 >= 720 - 30 ... {` - 720 should be the window height. -30 is the height of the square as the rectangle's origin - just like the screen's - is top left.

With this code, you should see a red square bouncing from top to bottom!

Before we get into the collisions, let's do a bit of refactoring.

We are drawing rectangles which raylib has a specific type for: `rl.Rectangle` and another drawing procedure that takes this type: `rl.DrawRectangleRec(rec, color)`

Let's condense our variables down and rename them:

```
paddle := rl.Rectangle{100, 100, 180, 30} // x, y, width, height
paddle_speed: f32 = 10

ball := rl.Rectangle{100, 150, 30, 30}
ball_dir := rl.Vector2{0, -1}
ball_speed: f32 = 10
```

You may have noticed we have swapped out our `i32` values for `f32`.

Rectangle's fields `x, y, width, height` are 32-bit floating point numbers rather than integers.

> Integer division always truncates the value. Floating point can represent decimals making it the usual choice for physical properties like position, velocity, acceleration, etc. int e.g. 3 / 2 = 1 float e.g. 3 / 2 = 1.5

There's another new type here: `rl.Vector2` that we are putting our direction into. A convenient way to hold two `f32` values allows us to do math on both fields at once later.

Here's the code after renaming the variables:

```
package main

import rl "vendor:raylib"

main :: proc() {
	paddle := rl.Rectangle{100, 100, 180, 30}
	paddle_speed: f32 = 10

	ball := rl.Rectangle{100, 150, 30, 30}
	ball_dir := rl.Vector2{0, -1}
	ball_speed: f32 = 10

	rl.InitWindow(1280, 720, "Pong")
	rl.SetTargetFPS(60)

	for !rl.WindowShouldClose() {
		if rl.IsKeyDown(rl.KeyboardKey.A) {
			paddle.x -= paddle_speed
		}
		if rl.IsKeyDown(rl.KeyboardKey.D) {
			paddle.x += paddle_speed
		}

		next_ball_rect := ball
		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
		}

		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()
	}
}
```

Collision detection

Raylib comes with a built in function that can detect whether two rectangles are overlapping: `rl.CheckCollisionRecs(a, b)`

> For games with fast moving obstacles, this approach is not robust enough. Imagine a bullet flying towards a wall. The bullet is close to the wall on one frame, and will be on

> the other side on the next frame. On either frame, the procedure will return false and the bullet will pass through the wall. I will cover subject this in an intermediate course.

We want to check if the ball is going to collide with the paddle this frame.

To properly set the ball's position when it bounces off the paddle, we will want to do this before we set the new ball position but after we calculate `next_ball_rect`.

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

if rl.CheckCollisionRecs(next_ball_rect, paddle) {
        ball_dir.y *= -1
}

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

With just these few lines, we have the ball bouncing off the paddle!

There's still a bit of work left to be done, but we are getting close.

Did you notice the Input→Processing→Output cycle in this lesson?

Input: Ball's next position, paddle's position (already adjusted this frame)
Processing: Check for overlap or hitting the edge of the screen
Output: Reverse the ball's direction

In the next lesson we are going to set up the screen a bit more like Pong and get the ball to bounce at different angles.

Good job so far, and see you in the next one.

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

Here's the code at the end of this lesson:

```
package main

import rl "vendor:raylib"

main :: proc() {
        paddle := rl.Rectangle{100, 100, 180, 30}
        paddle_speed: f32 = 10

        ball := rl.Rectangle{100, 150, 30, 30}
        ball_dir := rl.Vector2{0, -1}
        ball_speed: f32 = 10
```

```
	rl.InitWindow(1280, 720, "Pong")
	rl.SetTargetFPS(60)

	for !rl.WindowShouldClose() {
		if rl.IsKeyDown(rl.KeyboardKey.A) {
			paddle.x -= paddle_speed
		}
		if rl.IsKeyDown(rl.KeyboardKey.D) {
			paddle.x += paddle_speed
		}

		next_ball_rect := ball
		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 rl.CheckCollisionRecs(next_ball_rect, paddle) {
			ball_dir.y *= -1
		}

		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()
	}
}
```