Welcome to this course! I'm Nomin, your instructor. In this course, we will be using Rust and the Bevy engine to create a simple Pong game. The game will feature a ball and two paddles, one controlled by the player via the keyboard, and the other controlled by the computer. Additionally, we will implement a basic scoring system to keep track of the game's progress.

Course Objectives

- Expand your knowledge of Rust programming by building on basic concepts to tackle more complex applications.
- Understand the fundamentals of the Bevy engine and learn how to utilize its features for game creation.
- Apply your knowledge to develop a mini Pong game, reinforcing your learning through practical application.

Prerequisites

Before starting this course, you should be familiar with the following programming fundamentals:

- Variables
- Loops
- Conditions
- Functions

Additionally, you should have an understanding of data structures, including:

- Arrays
- Vectors
- Dictionaries

Prior experience with Rust is also necessary. You should be comfortable with Rust's syntax, ownership, error handling, and using Cargo for project management.

Required Tools

To follow along with this course, ensure you have the following tools ready:

- Cargo
- Bevy
- A code editor (We will be using Visual Studio Code, but feel free to use any editor you prefer)

About Zenva

Zenva is an online learning academy with over a million learners. We offer a wide range of courses suitable for both beginners and those looking to learn something new. Our courses are versatile, allowing you to learn in various ways, such as watching video tutorials, reading lesson summaries, or following along with the instructor using the included project files.

Let's get started!

Welcome back everyone. Today, we're going to explore Bevy, a simple data-driven game engine built in Rust. Let's dive into some core concepts that make Rust and Bevy powerful tools for game development.

Why Rust for Game Development?

- Rust is designed for system-level programming, offering low-level control without sacrificing high-level convenience.
- Rust's ownership model guarantees thread safety and enables fearless concurrency, allowing you to write concurrent code more easily.
- Rust ensures memory safety and eliminates data races while maintaining performance close to that of C.

Why Choose Bevy?

- Bevy uses ECS (Entity Component System) architecture, which decouples entities, components, and systems, making it easy to manage game state logic.
- Cross-platform deployment: Write your game once and deploy it across multiple platforms, including Windows, Mac, Linux, and web.
- Bevy provides powerful tools for real-time rendering, making it suitable for graphic-intensive applications.

Key Concepts in Rust

- Rust ensures memory safety without a garbage collector. Each value has a single owner, and when the owner goes out of scope, the value is dropped.
- Rust allows references to data without taking ownership.
- Lifetimes prevent dangling references, ensuring data is safely accessed.
- Rust includes a powerful feature with match statements for handling different cases, making the code more expressive and robust.

Entity Component System (ECS) in Bevy

- **Entity:** A unique identifier for an object in the game. Think of them as game objects.
- **Components:** Data containers for specific attributes of entities, such as position, velocity, or health.
- **Systems:** Logic processes that operate on entities with specific components, driving the behavior and interaction in your game.

This separation of data and behavior allows for a flexible and efficient game design.

Basic Syntax in Rust for Bevy Game Development

- mut: Makes a variable mutable. By default, variables are immutable in Rust.
- fn: Defines functions. Functions can have parameters and return values.
- struct: Creates custom data types. Structs are used to bundle related data together.
- enum: Defines a type by enumerating its possible values. Enums are used for types that can be one of several variants.

Primary Scalar Types in Rust

1. Integers
2. Floating point numbers
3. Booleans
4. Characters

Understanding these basic data types is crucial for managing and manipulating data effectively in your Rust programs.

By mastering these concepts and syntax, you'll be well on your way to creating powerful and efficient games using Bevy and Rust. Happy coding!

Welcome back, everyone. In this lesson, let’s go ahead and set up our Bevy project. Before we start, you must ensure that you have Rust installed. If not, you can always get it from the official web page.

Setting Up the Bevy Project

To begin, open the terminal in the directory where you want to create your game. Utilize the command cargo new my_bevy_game and navigate inside this folder.

```
cargo new my_bevy_game
```

Let’s go ahead and open this in our text editor, which is Visual Studio Code in my case. Here we can see that our project has been created with a source directory containing a main.rs file. And in here, there’s only one function, which is to print the line “hello world”.

```
fn main() {
    println!("hello world");
}
```

In the Cargo.toml file, it contains information about the nested packages and dependencies we need for this game. Under the dependencies, we want to add bevy = "0.14", which is the version we’ll be utilizing.

```
[dependencies]
bevy = "0.14"
```

Open the terminal once again, we utilize cargo run. This is a command provided by Cargo, the Rust package management build system. This will compile our Rust project, and through all the source files are up to date. It compiles both the project code and the dependencies listed in our Cargo.toml file. As this is the first time running, it will need to fetch all the dependencies, so this might take a while. But once it has finished, we can see that it has printed the line “hello world”.

```
cargo run
```

This completes the setup of our Bevy project. In the next lessons, we will start building our game by adding more functionalities and components.

Welcome back everyone. In this lesson, we'll be focusing on the project structure, which is designed to modularize the code for better organization, maintainability, and scalability. We'll be organizing the project in a way that aligns with the Entity-Component-System (ECS) architecture we previously mentioned. This structure will help ensure that the code is easy to follow and manage as the game grows in complexity.

Project Structure Overview

The project structure is designed to separate concerns and centralize related functionalities. Here's an overview of the key files and directories we'll be creating:

Components

The first file we want to create is components.rs. This file will define all the ECS components used in the game, such as player, ball, and paddle. Centralizing these components ensures a single place to manage all the data types attached to entities, making it easier to track and modify.

Systems Directory

Next, we want to create a systems directory. This directory will contain all the individual modules for different systems in the game. Each module handles a specific aspect of the game logic, ensuring separation of concerns.

- mod.rs: The module file that re-exports all the systems. This allows easy import of all systems in main.rs just by referencing the system module.
- ball.rs: Contains systems related to the ball's behavior, such as spawning, movement, and scoring detection.
- paddle.rs: Manages the system for the paddle behavior, including input from player one and player two.
- scoreboard.rs: Handles systems related to the scoreboard, such as updating and displaying scores.
- collision.rs: Contains the collision system and logic to handle interactions between the ball and paddle.

Bundles

Within the source directory, we create a bundles.rs file. This file defines the entity bundles, which are collections of components that are frequently used together. For example, a ball bundle might include ball and position components. Using these bundles helps streamline entity creation, ensuring that all the required components are included.

Constants

Lastly, we have the constants.rs file, which contains the constant values used throughout the game, such as speed, dimensions, and other configurations. Centralizing these values ensures that changes to these values are easy to manage and propagate throughout the codebase.

Benefits of This Project Structure

This overall project structure allows us to break down the code into modules based on functionality. By separating components, systems, and bundles, each file has a clear responsibility. This helps manage complexity and reduces the likelihood of errors. As the game grows in complexity, new features and systems can be added without disrupting the existing code significantly.

Summary

In summary, organizing your project structure in this modular way aligns with the ECS architecture and promotes better organization, maintainability, and scalability. By following this structure, you'll find it easier to manage and extend your game as it grows.

Welcome back everyone. Now that we've created our project structure and laid the foundation, let's start coding by defining our components.

Imports

We begin by making the necessary imports with `use bevy::prelude::*;` and `use bevy::math::Vec2;`. These imports bring in all the common Bevy types and functions, as well as the `Vec2` type from Bevy's math module. `Vec2` is a vector type with two components, x and y, which is commonly used for 2D coordinates.

```
use bevy::prelude::*;
use bevy::math::Vec2;
```

Defining Components

Next, we want to utilize the `#[derive(Component)]` macro. This macro automatically implements the necessary traits for the structure to be used as a component in Bevy's Entity Component System (ECS).

Player Components

The first component we define is `Player1`, which will serve as a tag or marker component to identify entities representing player 1. Similarly, we define `Player2` for player 2.

```
#[derive(Component)]
pub struct Player1;

#[derive(Component)]
pub struct Player2;
```

Score Components

We also define components to keep track of the scores for each player:

```
#[derive(Component)]
pub struct Player1Score;

#[derive(Component)]
pub struct Player2Score;
```

Scorer Enum

Next, we define a public enum `Scorer` with two variants, `Player1` and `Player2`. This will be used to indicate which player scored.

```
pub enum Scorer {
    Player1,
    Player2,
```



```
}
```

Game Object Components

We then define components for the game objects, which will be a ball, a paddle, and the boundary that contains everything:

```
#[derive(Component)]
pub struct Ball;

#[derive(Component)]
pub struct Paddle;

#[derive(Component)]
pub struct Boundary;
```

Position and Velocity Components

We also define components to store the position and velocity of entities. This is where our Vec2 import will come into play:

```
#[derive(Component)]
pub struct Position(pub Vec2);

#[derive(Component)]
pub struct Velocity(pub Vec2);
```

Shape Component

The `Shape` component is used to store the size or dimensions of an entity in a 2D space:

```
#[derive(Component)]
pub struct Shape(pub Vec2);
```

Scored Event

We define an event `Scored` that takes in the `Scorer` enum. This event is used to create events indicating that a player has scored:

```
#[derive(Event)]
pub struct Scored(pub Scorer);
```

Score Resource

Finally, we define a `Score` resource that will keep track of the scores for both players. This resource is derived from `Resource` and `Default` traits:

```
#[derive(Resource, Default)]
pub struct Score {
    pub player1: u32,
    pub player2: u32,
}
```

By defining these components and resources, we have laid the groundwork for our game's entities and their properties. In the next lessons, we will use these components to create and manage the game's entities and logic.

Now that we've created the different components needed for our game, let's define all the constants, which will be utilized throughout our development. Constants are essential as they allow us to easily manage and update values that are used frequently in our code.

Defining Constants

In Rust, we use the pub keyword to make an item public, meaning it's accessible by other modules. Let's start by defining the size of the popcorns (balls) and the paddle dimensions.

Ball Size

We'll define the size of the ball as a floating-point number (f32).

```
pub const BALL_SIZE: f32 = 5.0;
```

Paddle Dimensions

Next, we'll define the width and height of the paddles. Initially, we'll set both to 5, but we'll adjust the paddle width to 10 and the height to 50 to make it a rectangle shape.

```
pub const PADDLE_WIDTH: f32 = 10.0;
pub const PADDLE_HEIGHT: f32 = 50.0;
```

Speeds

We also need to define the speeds for the ball and the paddles. These speeds will determine how fast the ball and paddles move in the game.

```
pub const BALL_SPEED: f32 = 5.0;
pub const PADDLE_SPEED: f32 = 5.0;
```

Boundary Height

Lastly, we'll define the height of the boundaries at the top and bottom of the game window. This will act as a border for our game.

```
pub const BOUNDARY_HEIGHT: f32 = 20.0;
```

Complete Constants File

Here is the complete constants.rs file with all the defined constants:

```
pub const BALL_SIZE: f32 = 5.0;
pub const PADDLE_WIDTH: f32 = 10.0;
pub const PADDLE_HEIGHT: f32 = 50.0;

pub const BALL_SPEED: f32 = 5.0;
pub const PADDLE_SPEED: f32 = 5.0;
```



```
pub const BOUNDARY_HEIGHT: f32 = 20.0;
```

By defining these constants, we make our code more maintainable and easier to update. If we need to change the size of the ball or the speed of the paddles, we only need to update the values in this file, and the changes will be reflected throughout the entire game.

In this lesson, we will create bundles in Bevy, which are collections of components that are often used together. Bundles help in reducing boilerplate code when spawning different entities. We will define and implement bundles for the ball, paddle, and boundary entities in our game.

Importing Necessary Modules

First, let's begin by making the necessary imports. We need to import Bevy's prelude, math module for vector operations, and our custom components and constants modules.

```
use bevy::prelude::*;



use bevy::math::Vec2;



use crate::components::*;



use crate::constants::*;
```

Defining the Ball Bundle

Bundles in Bevy are defined using the #[derive(Bundle)] attribute. This attribute provides the necessary traits for the struct to be used as a bundle in Bevy's ECS (Entity Component System). We'll start by making our ball bundle, for which we'll need to know it's shape, velocity, and position. As such, our bundle will attach those components to our ball.

```
#[derive(Bundle)]
pub struct BallBundle {
    pub ball: Ball,
    pub shape: Shape,
    pub velocity: Velocity,
    pub position: Position,
}
```

Next, we implement the BallBundle with a constructor function to create a new instance of the bundle. The shape will use our constants variables in this case since we already defined the ball size.

```
impl BallBundle {
    pub fn new(x: f32, y: f32) -> Self {
        Self {
            ball: Ball,
            shape: Shape(Vec2::new(BALL_SIZE, BALL_SIZE)),
            velocity: Velocity(Vec2::new(x, y)),
            position: Position(Vec2::new(0.0, 0.0)),
        }
    }
}
```


Defining the Paddle Bundle

Similarly, we define the PaddleBundle for the paddle entity, which also needs a shape, position, and velocity component.

```
#[derive(Bundle)]
pub struct PaddleBundle {
    pub paddle: Paddle,
    pub shape: Shape,
    pub position: Position,
    pub velocity: Velocity,
}
```

We also implement the PaddleBundle with a constructor function to create an instance. Likewise to the ball bundle, we can use our constants variables to set the shape of the bundle.

```
impl PaddleBundle {
    pub fn new(x: f32, y: f32) -> Self {
        Self {
            paddle: Paddle,
            shape: Shape(Vec2::new(PADDLE_WIDTH, PADDLE_HEIGHT)),
            position: Position(Vec2::new(x, y)),
            velocity: Velocity(Vec2::new(0.0, 0.0)),
        }
    }
}
```

Defining the Boundary Bundle

Finally, we define the BoundaryBundle for the boundary entity. In this case, as our boundary won't move, it doesn't need a velocity component.

```
#[derive(Bundle)]
pub struct BoundaryBundle {
    pub boundary: Boundary,
    pub shape: Shape,
    pub position: Position,
}
```

We implement the BoundaryBundle with a constructor function that takes the width as a parameter, allowing for dynamic window sizes.

```
impl BoundaryBundle {
    pub fn new(x: f32, y: f32, width: f32) -> Self {
        Self {
            boundary: Boundary,
            shape: Shape(Vec2::new(width, BOUNDARY_HEIGHT)),
            position: Position(Vec2::new(x, y)),
```



```
        }
    }
}
```

Summary

In this lesson, we created bundles for the ball, paddle, and boundary entities in our game. These bundles help in reducing boilerplate code when spawning entities by grouping related components together. In the next lessons, we will use these bundles to spawn and manage our game entities.

Welcome back everyone. In this lesson, we will focus on setting up the camera in our Bevy game. Let's get started by opening the camera.rs file.

Importing Bevy's Preload

First, we begin by importing Bevy's preload once again. This is essential for loading resources in our game.

```
use bevy::prelude::*;
```

Creating the Spawn Camera Function

Next, we want to create a public function called spawn_camera, which takes in a mutable commands parameter. The commands parameter is a Bevy resource that allows you to create, modify, or delete entities and their components.

```
pub fn spawn_camera(mut commands: Commands) {
    commands.spawn(Camera2dBundle::default());
}
```

In this function, we use commands.spawn to create a new entity of the default 2D camera. This will include the camera component and a transform component that positions the camera.

Understanding Cameras in Bevy

In Bevy, cameras are entities that have one or more components defining their properties. They allow us to render the view, meaning the camera determines what part of the game world is visible on the screen. The transform component of the camera defines its position and orientation in the game world. Bevy supports multiple cameras, each potentially rendering different parts of the game world or used for different purposes, such as split-screen or multiplayer.

Updating the Main Function

Now, let's move to the main.rs file to make some necessary updates to initialize our app. We begin by making the necessary imports:

```
use bevy::prelude::*;
mod systems;

use systems::*;
```

Initializing the App

We remove the print line statement and modify the code to initialize our app:

```
fn main() {
    App::new()
        .add_plugins(DefaultPlugins)
        .add_systems(Startup)
        .run();
```



```
}
```

Here, we create a new instance of the app using App::new(). We then add the default plugins, which provide essential functionalities required for a typical game, including window creation, event handling, input management, and rendering.

Adding Systems

Next, we add systems and specify that the spawn_camera system will be modified at the start of the game:

```
.add_systems(Startup, spawn_camera)
```

Modifying the Modules

We also need to modify the mods.rs file to define and re-export the necessary modules:

```
pub mod ball;
pub mod paddle;
pub mod camera;

pub use ball::*;
pub use paddle::*;
pub use camera::*;
```

This makes the components, bundles, and systems accessible from outside the current module.

Finalizing the Main Function

Heading back to the main.rs file, we use .run() to start running the app. Our final function will look something like this:

```
fn main() {
    App::new()
    .add_plugins(DefaultPlugins)
    .add_systems(Startup, spawn_camera)
    .run();
}
```

Running the App

Once we run the app using cargo run, we can see that a new window has loaded, displaying our game with the camera set up.

That’s it for this lesson! You’ve successfully set up the camera in your Bevy game. In the next lessons, we will continue building our game by adding more components and systems.

Welcome back everyone. Now that we can see our window, let's modify it to be more personal to our usage. We will start by customizing the window title and then add a dotted line in the middle to represent the net, similar to what you might see in the physical world.

Customizing the Window Title

To customize the window title, we need to modify the windows.rs file. Follow these steps:

1. Open the windows.rs file.
2. Make the necessary imports at the top of the file:

```
use bevy::prelude::*;
```

3. Create a public function called create_window that returns a WindowPlugin:

```
pub fn create_window() -> WindowPlugin {
    WindowPlugin {
        primary_window: Some(Window {
            title: "My Pong Game".to_string(),
            ..default()
        }),
        ..default()
    }
}
```

4. In the main.rs file, add the create_window function to the default plugins:

```
App::new()
    .add_plugins(DefaultPlugins.set(create_window()))
```

5. Ensure that the mod.rs file defines and uses the window module:

```
pub mod window;
pub use window::*;
```

When you run the command cargo run, you should see the window load with the title "My Pong Game".

Adding a Dotted Line

Next, let's add a dotted line in the middle of the window to represent the net. Follow these steps:

1. Define a public function called spawn_dotted_line in the windows.rs file. We'll define a color, size, and gap for the dots on our net, as well as the number of dots we'll use. We then loop through based on the number of dots calculated and instantiate each dot at a new position based on the size and gap:

```
pub fn spawn_dotted_line(mut commands: Commands) {
    let dot_color = Color::srgb(1.0, 1.0, 1.0);
    let dot_size = Vec3::new(5.0, 20.0, 1.0);
    let gap_size = 10.0;
    let num_dots = (constants::WINDOW_HEIGHT / (dot_size.y + gap_size)) as i32;
```



```
    for i in 0..num_dots {
        commands.spawn(SpriteBundle {
            sprite: Sprite {
                color: dot_color,
                ..default()
            },
            transform: Transform {
                translation: Vec3::new(0.0, i as f32 * (dot_size.y + gap_size) - cons
tants::WINDOW_HEIGHT / 2.0, 0.0),
                scale: dot_size,
                ..default()
            },
            ..default()
        });
    }
}
```

2. Ensure that the constants.rs file defines the window height:

```
pub const WINDOW_HEIGHT: f32 = 600.0;
```

3. Import the constants module in the windows.rs file:

```
use crate::constants::*;
```

4. Add the spawn_dotted_line function to the startup systems in the main.rs file:

```
.add_systems(Startup, (spawn_dotted_line, spawn_camera))
```

When you run the command cargo run, you should see the window load with a dotted line at the center of it.

As a small challenge, you can modify the size and shape of the dots. We'll cover the solution for this in the next lesson!

Welcome back everyone. In this lesson, we will modify the dotted lines in our game to ensure we understand how different entities are being spawned. We will begin by changing the color, gap size, and dot size within the windows.rs file.

Modifying the Dotted Lines

Let's start by changing the color of the dots. Instead of white, we will modify the second component to be zero, making the color more pink.

```
let dot_color = Color::srgb(1.0, 0.0, 1.0);
```

Next, we will adjust the gap size. Instead of 10, we will set it to 20, making the gaps between the dots a little bit larger.

```
let gap_size = 20.0;
```

Finally, we will change the dot size. We will modify both the X and Y components to make the dots smaller.

```
let dot_size = Vec3::new(3.0, 15.0, 1.0);
```

Here is the complete modified code for the dotted lines:

```
pub fn spawn_dotted_line(mut commands: Commands) {
    let dot_color = Color::srgb(1.0, 0.0, 1.0);
    let dot_size = Vec3::new(3.0, 15.0, 1.0);
    let gap_size = 20.0;
    let num_dots = (constants::WINDOW_HEIGHT / (dot_size.y + gap_size)) as i32;

    for i in 0..num_dots {
        commands.spawn(SpriteBundle {
            sprite: Sprite {
                color: dot_color,
                ..default()
            },
            transform: Transform {
                translation: Vec3::new(0.0, i as f32 * (dot_size.y + gap_size) - cons
tants::WINDOW_HEIGHT / 2.0, 0.0),
                scale: dot_size,
                ..default()
            },
            ..default()
        });
    }
}
```

When you run the game using cargo run, you will see the changes have been made. The color is now

more pink, the gaps are bigger, and the lines are thinner. This exercise demonstrates the importance of clearly defined variables and components. If we need to make any specific changes in the future, it’s easy to modify one component without affecting the others.

For the sake of cleanliness, we will revert the elements back to what we had previously. However, feel free to play around with the values to find something that suits you.

```
let dot_color = Color::srgb(1.0, 1.0, 1.0);
let dot_size = Vec3::new(5.0, 20.0, 1.0);
let gap_size = 10.0;
```

This concludes our lesson on modifying the dotted lines. By understanding how to change these properties, you gain a better grasp of how entities are spawned and managed in the game.

Welcome back everyone. Now that we have the window set for our game, let's start creating our ball component.

Importing Modules

We begin in the ball.rs file, and we create the necessary imports, which will be use bevy::prelude::*, but we also want to import use crate::components::*, as well as use crate::constants::*, and the BallBundle, as they'll be utilized to spawn the ball. And lastly, we want to import from bevy::sprite::MaterialMesh2dBundle. This is the bundle of components used for rendering a 2D mesh with a material.

```
use bevy::prelude::*;
use crate::components::*;
use crate::constants::*;
use crate::BallBundle;
use bevy::sprite::MaterialMesh2dBundle;
```

Defining the Spawn Ball Function

Next, we define the public function of spawn_ball. This takes in three parameters, first of which is the mutable commands, then mutable meshes, which will be a resource type of assets::Mesh, allowing the function to add new mesh assets to the game. And lastly, materials, which takes in the mutable resource type of assets::ColorMaterial, allowing the function to add new material assets to the game.

```
pub fn spawn_ball(
    mut commands: Commands,
    mut meshes: ResMut<Assets<Mesh>>,
    mut materials: ResMut<Assets<ColorMaterial>>,
) {
    // Function implementation will go here
}
```

Creating the Ball Shape and Color

Now let's define some variables. First, let shape equals Mesh::from(Circle::new(BALL_SIZE)). This creates a new mesh circle with the radius defined by the constant BALL_SIZE. This represents the shape of the ball.

```
let shape = Mesh::from(Circle::new(BALL_SIZE));
```

Next, let's define let color = ColorMaterial::from(Color::srgb(1.0, 0.0, 0.0)). This means it creates a new color material from the sRGB color values, which 1.0, 0.0, 0.0 corresponds to the color red.

```
let color = ColorMaterial::from(Color::srgb(1.0, 0.0, 0.0));
```

Adding the Mesh and Material to the Asset

We define let mesh_handle = meshes.add(shape). This adds the previously created shape to the

mesh’s asset storage and returns a handle to the mesh.

```
let mesh_handle = meshes.add(shape);
```

Then, let material_handle = materials.add(color). This adds the previously created color to the materials.

```
let material_handle = materials.add(color);
```

Spawning the Ball Entity

Now we want to spawn a new entity and provide its components. We want this to be a BallBundle, which will be a new entity, with the initial velocity of 1.0, 0.0. Then we also want to have the MaterialMesh2dBundle, which has mesh: mesh_handle.into(), meaning we’re setting the mesh component of the entity to the mesh handle we created earlier. We do the same for the material with material: material_handle.

```
commands.spawn((
    BallBundle::new(1.0, 0.0),
    MaterialMesh2dBundle {
        mesh: mesh_handle.into(),
        material: material_handle,
        ..default()
    }
));
```

Adding the Spawn Ball Functionality to the Main Function

Heading over to the main.rs file, within the add_systems of Startup, we want to add the functionality of spawn_ball. We also want to make sure to import the mod components and that we’re able to use the components. We also want to do the same for the mod bundles and components as the spawn_ball functionality uses components and bundles.

```
use bevy::prelude::*;
mod systems;
mod constants;
mod components;
mod bundles;

use systems::*;
use components::*;
use bundles::*;

fn main() {
    App::new()
    .add_plugins(DefaultPlugins.set(create_window()))
    .add_systems(Startup, (spawn_dotted_line, spawn_ball, spawn_camera))
    .run();
}
```


Within our bundle of BallBundle, we want to say x: f32, meaning x is a floating type. And once we modify this and run cargo run, we can see that in the middle of our screen, a red dot has been created, which represents our ball.

```
#[derive(Bundle)]
pub struct BallBundle {
    pub ball: Ball,
    pub shape: Shape,
    pub velocity: Velocity,
    pub position: Position,
}

impl BallBundle {
    pub fn new(x: f32, y: f32) -> Self {
        Self {
            ball: Ball,
            shape: Shape(Vec2::new(BALL_SIZE, BALL_SIZE)),
            velocity: Velocity(Vec2::new(x, y)),
            position: Position(Vec2::new(0.0, 0.0)),
        }
    }
}
```

This concludes the lesson on creating the ball component in our game. In the next lesson, we will continue to build upon this foundation and add more functionality to our game.

Welcome back everyone. Now that we can see the ball on our screen, let’s go ahead and create two paddles that represent our two different players. Anything related to the paddle component will be done in the paddle.rs file.

Setting Up the Paddle Component

To begin with, we create the necessary array imports. These are similar imports that we created in our ball.rs file. But here, instead of a ball bundle, we just make sure we import the paddle bundle.

```
use bevy::prelude::*;
use crate::components::*;
use crate::constants::*;
use crate::PaddleBundle;
use bevy::sprite::MaterialMesh2dBundle;
```

Spawn Paddles Function

We define our public function of spawn_paddles. This takes in three different parameters:

- A mutable command parameter, which again is used to create or manage entities and their components.
- Meshes, which is a mutable resource type of assets mesh.
- Materials, which takes in a mutable resource type of assets color material.
- Different to the ball bundle, this will take in a window, which is the query of window, meaning this takes in a query to get the window property, such as width.

```
pub fn spawn_paddles(
    mut commands: Commands,
    mut meshes: ResMut<Assets>,
    mut materials: ResMut<Assets>,
    window: Query,
) {
    // Function implementation
}
```

Fetching Window Dimensions

In this function, let’s go ahead and fetch some relevant information about the window dimension. We state if let Ok(window) = window.get_single() meaning it tries to get a single window entity and if it is successful it will proceed within this code block. We also use this opportunity to define some padding as well as the x placement of the individual paddles (since we’ll want on the left and one on the right).

```
if let Ok(window) = window.get_single() {
    let window_width = window.resolution.width();
    let padding = 50.0;
    let right_paddle_x = window_width / 2.0 - padding;
    let left_paddle_x = -window_width / 2.0 + padding;
}
```

Creating the Paddle Mesh

Next, let's create the paddle mesh. We do let mesh = Mesh::from(Rectangle::new(PADDLE_WIDTH, PADDLE_HEIGHT)), meaning creating a rectangular mesh with a specified paddle width and height. Then we state let mesh_handle = meshes.add(mesh), meaning we're adding the rectangular mesh to the meshes assets.

```
let mesh = Mesh::from(Rectangle::new(PADDLE_WIDTH, PADDLE_HEIGHT));
let mesh_handle = meshes.add(mesh);
```

Spawning Player Paddles

Now, we want to spawn player one paddle. So we do commands.spawn, meaning we're spawning a new entity with the provided components. We add the Player1 component to the entity, identifying it as the first player. Then we create a new paddle with the calculated x-coordinate for the right paddle and the y-coordinate set to zero. And the paddle bundle is something we previously defined ourselves, which took in paddle, position, vector, and shape.

Next, we use MaterialMesh2dBundle to render our 2D mesh material. Here we say mesh: mesh_handle.clone().into(), meaning we're using .clone() because we want to clone the handle so it can be reused later on for the player too. For the material, we want to add the color material and from color use srgb(0.0, 1.0, 0.0) meaning this would be a green panel.

```
commands.spawn((
    Player1,
    PaddleBundle::new(right_paddle_x, 0.0),
    MaterialMesh2dBundle {
        mesh: mesh_handle.clone().into(),
        material: materials.add(ColorMaterial::from(Color::srgb(0.0, 1.0, 0.0))),
        ..default()
    },
));
```

We'll copy this over for player 2. We want to change who the paddle belongs to and change the position it spawns in to be the left paddle. We'll also change the color value to srgb(0.0, 0.0, 1.0)– meaning this will be a blue material which allows us to differentiate between the players.

```
commands.spawn((
    Player2,
    PaddleBundle::new(left_paddle_x, 0.0),
    MaterialMesh2dBundle {
        mesh: mesh_handle.into(),
        material: materials.add(ColorMaterial::from(Color::srgb(0.0, 0.0, 1.0))),
        ..default()
    },
));
```

Spawning Paddles in the Main Function

Heading over to the main functionality, as well as we're spawning our dotted line and ball on

camera, we want to spawn our paddles here.

```
fn main() {
    App::new()
    .add_plugins(DefaultPlugins.set(create_window()))
    .add_systems(Startup, (spawn_dotted_line, spawn_ball, spawn_paddles, spawn_camera
))
    .run();
}
```

When we use the command cargo run, we can see that it’s flickering but that there are two different paddles as well as our balls. In the next lesson, we’ll go ahead and position this according to the window by adding movement to our ball.

Welcome back everyone. In this lesson, we will create some movement for the ball and ensure that the components are in the right places in the window. Let's start by defining a function to move the ball within the balls.rs file.

Defining the moveBall Function

We begin by defining a public function called moveBall. This function takes a parameter of a mutable ball, which is a query that retrieves entities with both position and velocity components, specifically those that will have a ball component. This means that only the ball entity will be affected by the system.

```
pub fn move_ball(
    mut ball: Query<(&mut Position, &Velocity), With<Ball>>
) {
    if let Ok((mut position, velocity)) = ball.get_single_mut() {
        position.0 += velocity.0 * BALL_SPEED;
    }
}
```

Here's a breakdown of the code:

- The function move_ball takes a mutable query of entities that have both Position and Velocity components and are marked with the Ball component.
- The if let Ok statement attempts to get a single mutable reference to the ball's entity position and velocity components.
- If successful, it destructures the result into mutable position and velocity.
- The ball's position is updated by adding the product of the velocity times the constant BALL_SPEED.

Updating Entity Positions

Next, let's create a function called update_entity_positions. This function updates the transform component of entities so that the visual representation is in sync with the logical position.

```
pub fn update_entity_positions(
    mut ball: Query<(&mut Transform, &Position)>
) {
    for (mut transform, position) in ball.iter_mut() {
        transform.translation = position.0.extend(0.);
    }
}
```

Here's a breakdown of the code:

- The function update_entity_positions takes a mutable query of entities that have both Transform and Position components.
- It iterates over all the entities that match the query criteria, providing mutable references to the Transform and Position components.
- The translation field of the Transform component is updated to match the Position component.
- The .extend(0.) converts the 2D vector to a 3D vector by extending a z value of 0. This is

necessary because Transform uses a 3D vector for its translations.

Adding Systems to the Main Function

Finally, we need to add these systems to the main function to ensure they run throughout the game. We will add the move_ball and update_entity_positions systems to the Update stage of the Bevy app.

```
fn main() {
    App::new()
    .add_plugins(DefaultPlugins.set(create_window()))
    .add_systems(Startup, (spawn_dotted_line, spawn_ball, spawn_paddles, spawn_camera
))
    .add_systems(Update, (
        move_ball,
        update_entity_positions.after(move_ball),
    ))
    .run();
}
```

Here’s a breakdown of the code:

- The add_systems method is used to add systems to the Update stage.
- The move_ball system is added first.
- The update_entity_positions system is added after the move_ball system using the .after method to ensure it runs after the ball’s position has been updated.

When you run the game using the command cargo run, you should now see the two paddles on the left and right-hand side, as well as the ball moving to the right-hand side.

This concludes our lesson on creating movement for the ball and updating entity positions. In the next lesson, we will get the paddles moving.

Welcome back, everyone. Now that the ball is moving, in this lesson, let's focus on the movement of the paddles.

Defining the movePlayer1Paddle Function

In the paddles.rs file, we define our first function of movePlayer1Paddle. It'll take in two parameters:

- A resource of button input that takes in a key code – meaning this resource allows us to access the current state of the keyboard input.
- A mutable reference of our paddle which retrieves the velocity component of entities that also have a Player1 component.

Heading into the function, we check if let Ok(mut velocity) = paddle.getSingleMut() – meaning we're attempting to get a single mutable reference to the velocity of player one's paddle. The getSingleMut retrieves a mutable reference to a single entity that matches the query criteria.

If this check passes, it then checks if the up or down arrow keys are being pressed and adjusts the paddle's velocity accordingly. If the up arrow key is pressed, the paddle's y-velocity is set to 1, moving it upwards. If the down arrow key is pressed, the paddle's y-velocity is set to -1, moving it downwards.

If neither key is being pressed, the paddle's y-velocity is set to 0, stopping its movement. This process allows Player 1 to control their paddle's movement using the arrow keys.

```
fn move_player1_paddle(
    keyboard_input: Res<ButtonInput<KeyCode>>,
    mut paddle: Query<&mut Velocity, With<Player1>>,
) {
    if let Ok(mut velocity) = paddle.get_single_mut() {
        if keyboard_input.pressed(KeyCode::ArrowUp) {
            velocity.0.y = 1.0;
        } else if keyboard_input.pressed(KeyCode::ArrowDown) {
            velocity.0.y = -1.0;
        } else {
            velocity.0.y = 0.0;
        }
    }
}
```

Defining the movePlayer2Paddle Function

For player two, we can copy over much of the same functionality in their move paddle function. However, we need to alter the keycodes so that they use W and S instead of the up and down arrow keys.

```
fn move_player2_paddle(
    keyboard_input: Res<ButtonInput<KeyCode>>,
    mut paddle: Query<&mut Velocity, With<Player2>>,
) {
    if let Ok(mut velocity) = paddle.get_single_mut() {
        if keyboard_input.pressed(KeyCode::KeyW) {
            velocity.0.y = 1.0;
        } else if keyboard_input.pressed(KeyCode::KeyS) {
```



```
            velocity.0.y = -1.0;
        } else {
            velocity.0.y = 0.0;
        }
    }
}
```

Defining the movePaddles Function

Next, we want to create a function called movePaddles. This function updates the position of all the paddles based on the velocity and ensures they stay within the window bounds.

It takes two parameters: a Query for the paddles' positions and velocities, and a Query for the game window. The function first retrieves the game window and calculates its height.

Then, it iterates over each paddle and calculates its new position by adding its velocity multiplied by the PADDLE_SPEED constant. If the new position is within the boundaries of the window, the paddle's position is updated. This ensures that the paddles move smoothly and stay within the game area, preventing them from going off-screen.

```
fn move_paddles(
    mut paddle: Query<(&mut Position, &Velocity), With<Paddle>>,
    window: Query<&Window>,
) {
    if let Ok(window) = window.get_single() {
        let window_height = window.resolution.height();

        for (mut position, velocity) in &mut paddle {
            let new_position = position.0 + velocity.0 * PADDLE_SPEED;
            if new_position.y.abs() < window_height / 2.0 - PADDLE_HEIGHT / 2.0 {
                position.0 = new_position;
            }
        }
    }
}
```

Updating the Main Function

Next, we need to incorporate all three of our functions into the main function in main.rs. We want the movePaddles function only after movePlayer1Paddle the functionality has been used.

```
fn main() {
    App::new()
    .add_plugins(DefaultPlugins.set(create_window()))
    .init_resource::<Score>()
    .add_event::<Scored>()
    .add_systems(Startup, (spawn_dotted_line, spawn_ball, spawn_paddles, spawn_camera
))
    .add_systems(Update, (
        move_ball,
        move_player1_paddle,
        move_player2_paddle,
```



```
        update_entity_positions.after(move_ball),
        move_paddles.after(move_player1_paddle),
        handle_collisions.after(move_ball),
    ))
    .run();
}
```

If we run cargo run again, we can see the two paddles have loaded. And, using the keyboard, we can move the right-hand side with the up and down arrow, and the left-hand side with the keycode we created.

With these updates, you should now be able to control both paddles using the specified keys and see them move within the window bounds.

Welcome back, everyone. In this lesson, let's focus on handling the collision between the ball and other objects in the game, like the boundaries and the paddle. We will be working in the collisions.rs file in our systems directory.

Imports and Enum Definition

First, we need to make the necessary imports. As usual, we start with:

```
use bevy::prelude::*;
```

Next, we want to import the components we previously created:

```
use crate::components::*;
```

We also need to import specific types and traits relating to the bounding volumes and collision detection that the Bevy module provides:

```
use bevy::math::{bounding::{Aabb2d, BoundingCircle, BoundingVolume, IntersectsVolume}
};
```

Defining the Collision Enum

We want to derive an enum named Collision with four variants: Left, Right, Top, and Bottom. These represent the sides of an object that can be collided with. We will use the following traits for the enum:

```
#[derive(Debug, PartialEq, Eq, Copy, Clone)]
pub enum Collision {
    Left,
    Right,
    Top,
    Bottom,
}
```

Handling Collisions

Next, we will create a public function called handleCollision. This function will check for collisions between the ball and other objects and update the ball's velocity based on the collision. The function will take in the following parameters:

- A mutable ball query which retrieves entities with velocity, position, and shape, specifically those that have a ball component.
- A query that retrieves all the entities with position and shape but excludes all those with a ball component.

For each object, the code checks if a collision occurs between the ball and the object using the detect_collision function (which we'll make next). If a collision is detected, the respond_to_collision function is called to update the ball's velocity accordingly (which we'll make in the future as well).

This process efficiently detects and responds to collisions between the ball and other game object.

Here is the function definition:

```
pub fn handle_collisions(
    mut ball_query: Query<(&mut Velocity, &Position, &Shape), With>,
    other_things: Query<(&Position, &Shape), Without>,
) {
    if let Ok((mut ball_velocity, ball_position, ball_shape)) = ball_query.get_single
_mut() {
        for (position, shape) in &other_things {
            if let Some(collision) = detect_collision(ball_position, ball_shape, posi
tion, shape) {
                respond_to_collision(&mut ball_velocity, collision);
            }
        }
    }
}
```

Detecting Collisions

Now, let's define our detect_collision function. This function will take in the ball's position and shape, as well as the position and shape of other objects. It will return an option of Collision.

The function creates a bounding circle for the ball and an axis-aligned bounding box (AABB) for the other object, and then checks if the ball intersects with the AABB using the intersects method.

If an intersection is detected, the function calculates the closest point on the AABB to the ball's center and determines the offset between the two points. Based on the offset, the function returns a Collision enum value indicating the direction of the collision (left, right, top, or bottom). If no intersection is detected, the function returns None. This process enables the game to accurately detect and respond to collisions between the ball and other game objects.

```
fn detect_collision(
    ball_position: &Position,
    ball_shape: &Shape,
    other_position: &Position,
    other_shape: &Shape,
) -> Option {
    let ball = BoundingCircle::new(ball_position.0, ball_shape.0.x);
    let wall = Aabb2d::new(other_position.0, other_shape.0 / 2.0);

    if !ball.intersects(&wall) {
        return None;
    }

    let closest_point = wall.closest_point(ball.center());
    let offset = ball.center() - closest_point;

    if offset.x.abs() > offset.y.abs() {
        if offset.x < 0.0 {
            Some(Collision::Left)
        } else {
```



```
            Some(Collision::Right)
        }
    } else {
        if offset.y < 0.0 {
            Some(Collision::Top)
        } else {
            Some(Collision::Bottom)
        }
    }
}
```

Responding to Collisions

Finally, let's create the respond_to_collision function, which updates the ball's velocity based on the side of the collision. This function takes in a mutable reference to the ball's velocity component and the side of the collision.

```
fn respond_to_collision(velocity: &mut Velocity, collision: Collision) {
    match collision {
        Collision::Left | Collision::Right => velocity.0.x *= -1.0,
        Collision::Top | Collision::Bottom => velocity.0.y *= -1.0,
    }
}
```

Updating the Main Function

Now that we have defined all the necessary functions for collision handling and collision response, we want to move on to the main function and update handleCollision after moveBall. This means we want to handle any collision after the ball has started moving.

```
fn main() {
    App::new()
    .add_plugins(DefaultPlugins.set(create_window()))
    .add_systems(Startup, (spawn_dotted_line, spawn_ball, spawn_paddles, spawn_camera
))
    .add_systems(Update, (
        move_ball,
        move_player1_paddle,
        move_player2_paddle,
        update_entity_positions.after(move_ball),
        move_paddles.after(move_player1_paddle),
        handle_collisions.after(move_ball),
    ))
    .run();
}
```

We also need to ensure that the collision module is referenced by other parts of the project. To do this, head over to the mod.rs file and ensure that we have:

```
pub mod collision;
```



```
pub use collision::*;
```

Running the Game

Finally, run the game using cargo run. If there are any small errors, fix them and run the game again. You should see the ball move to the right and reverse its velocity when it hits the right-hand paddle, moving back to the left.

That’s it for this lesson! You’ve successfully implemented collision detection and response in your game. Keep practicing and exploring the code to deepen your understanding.

Welcome back, everyone. Now that we have the different components needed for a game of the ball movement, handling collision and paddle movement, we want to have some kind of scoring system in place. So in this lesson, let's create a scoreboard.

Setting Up the Scoreboard

Starting in the scoreboard.rs file, we begin as usual with making the necessary imports. We want to use bevy::prelude as well as use crate components. As we want to import all the components module.

```
use bevy::prelude::*;
use crate::components::*;
```

Defining the Spawn Scoreboard Function

Next, we want to define the public function of spawn_scoreboard. This will take in a parameter of multiple commands, which will allow us to create and modify the entities within this function.

```
pub fn spawn_scoreboard(mut commands: Commands) {
```

Spawning Player One Scoreboard

We utilize commands.spawn to spawn a new entity with the provided components. We want to have a text bundle creation using from_section, zero and text style. This means the text bundle will be created with the initial text of zero and the specified text style.

```
commands.spawn((
    TextBundle::from_section(
        "0",
        TextStyle {
            font_size: 64.0,
            color: Color::WHITE,
            ..default()
        }
    )
    .with_text_justify(JustifyText::Center)
    .with_style(Style {
        position_type: PositionType::Absolute,
        top: Val::Px(5.0),
        right: Val::Percent(45.0),
        ..default()
    }),
    Player1Score
));
```

Spawning Player Two Scoreboard

Next, we want to spawn the scoreboard for player two. The code is similar to player one, but we change the position to be from the left-hand side.

```
commands.spawn((
    TextBundle::from_section(
        "0",
        TextStyle {
            font_size: 64.0,
            color: Color::WHITE,
            ..default()
        }
    )
    .with_text_justify(JustifyText::Center)
    .with_style(Style {
        position_type: PositionType::Absolute,
        top: Val::Px(5.0),
        left: Val::Percent(45.0),
        ..default()
    }),
    Player2Score
));
```

Adding the Scoreboard to the Main Functionality

Now that we have this function, we want to move on to the main functionality. We want to add this to the system of startup and head over to the mod.rs ensuring we've done pub mod scoreboard, pub use scoreboard, so then it can be used by the main function.

```
pub mod scoreboard;
pub use scoreboard::*;
```

In main.rs, we need to make sure to spawn the scoreboard in our startup commands.

```
fn main() {
    App::new()
    .add_plugins(DefaultPlugins.set(create_window()))
    .add_systems(Startup, (spawn_dotted_line, spawn_ball, spawn_paddles, spawn_camera
, spawn_scoreboard))
    .add_systems(Update, (
        move_ball,
        move_player1_paddle,
        move_player2_paddle,
        update_entity_positions.after(move_ball),
        move_paddles.after(move_player1_paddle),
        handle_collisions.after(move_ball),
    ))
    .run();
}
```


Once that’s done, we can run cargo run again. When the code loads, we can see that on the page, there are two scores of 0 representing player one and the other representing player two.

That’s it for this lesson! We’ve successfully created a scoreboard for our game. In the next lessons, we’ll continue to build upon this foundation and actually make scoring possible.

Welcome back everyone. Now that we have the scoreboard display in our window, let's go ahead and create the functionalities that will keep track of the score and add it when one of the players has scored.

Creating the Scoring Functionality

We begin in the ball.rs file and we want to create a public function called detect_scoring. This function will have three parameters:

- mut ball_query: A mutable query that retrieves the position components of the entities with the Ball component.
- window_query: A query that retrieves the window component.
- mut score_event_writer: An event writer taking in Scored. This is a mutable event writer that allows the sending of scored events.

Within the body of the function, it first retrieves the window's width and calculates its half-width, which serves as the scoring threshold. Then, it retrieves the ball's current position and checks if it has crossed the threshold.

If the ball's x-coordinate exceeds the half-width, a score is awarded to Player 2. Conversely, if the ball's x-coordinate is less than the negative half-width, a score is awarded to Player 1. In either case, a Scored event is sent to update the game state accordingly – simply specifying *who* scored.

```
pub fn detect_scoring(
    mut ball_query: Query<&mut Position, With<Ball>>,
    window_query: Query<&Window>,
    mut score_event_writer: EventWriter<Scored>,
) {
    let window = match window_query.get_single() {
        Ok(w) => w,
        Err(_) => return,
    };

    let half_window_width = window.resolution.width() / 2.0;
    let ball_position = match ball_query.get_single_mut() {
        Ok(bp) => bp,
        Err(_) => return,
    };

    if ball_position.0.x > half_window_width {
        score_event_writer.send(Scored(Scorer::Player2));
    } else if ball_position.0.x < -half_window_width {
        score_event_writer.send(Scored(Scorer::Player1));
    }
}
```

Updating the Scoreboard

Now that we can detect any sort of scoring, we want to move on to the scoreboard.rs file and update the scoreboard. We have the function update_scoreboard that updates the text of the scoreboard we created to reflect the current scores of player one and player two. This function will take in the following parameters:

- mut player1_score: A mutable query of text with the Player1Score component.
- mut player2_score: A mutable query of text with the Player2Score component, excluding those with Player1Score.
- score: A resource that holds the current score for both players.

Within the body of the function, we want to check if the score resource has changed. If it has, we want to update the text of the scoreboard. Here is the code snippet for the update_scoreboard function:

```
pub fn update_scoreboard(
    mut player1_score: Query<&mut Text, With<Player1Score>>,
    mut player2_score: Query<&mut Text, (With<Player2Score>, Without<Player1Score>)>,
    score: Res<Score>,
) {
    if score.is_changed() {
        if let Ok(mut player1_score) = player1_score.get_single_mut() {
            player1_score.sections[0].value = score.player1.to_string();
        }
        if let Ok(mut player2_score) = player2_score.get_single_mut() {
            player2_score.sections[0].value = score.player2.to_string();
        }
    }
}
```

Updating the Score

Next, we want to have a function that updates the score itself. Let's name this a public function of update_score. This function will update the resource based on the scoring events. It will take in the following parameters:

- mut score: A mutable resource of the score.
- mut events: An event reader of Scored, aka the event reader that reads the scored event indicating when a player has scored.

Within the body of the function, we want to iterate through all the scored events and update the score accordingly. Here is the code snippet for the update_score function:

```
pub fn update_score(
    mut score: ResMut<Score>,
    mut events: EventReader<Scored>,
) {
    for event in events.read() {
        match event.0 {
            Scorer::Player1 => score.player1 += 1,
            Scorer::Player2 => score.player2 += 1,
        }
    }
    println!("Score: {} - {}", score.player1, score.player2);
}
```

Adding Systems to the Main Function

Heading over to the main.rs file, we want to include these two functions in our add system of update. We want to say, we want to detect the score. And then we want to update the score after there has been a detect score. And also update the scoreboard after detect scoring. We also want to initialize the score resource here. Using .initResource, this sets up the initial state for tracking the score for both players. We also want to add the event of scored, meaning we're registering the scored event. This allows the game to generate and listen to the scoring events.

Here is the code snippet for adding the systems to the main function:

```
fn main() {
    App::new()
    .add_plugins(DefaultPlugins.set(create_window()))
    .init_resource::<Score>()
    .add_event::<Scored>()
    .add_systems(Startup, (spawn_dotted_line, spawn_ball, spawn_paddles, spawn_camera
, spawn_scoreboard))
    .add_systems(Update, (
        move_ball,
        move_player1_paddle,
        detect_scoring,
        move_player2_paddle,
        update_score.after(detect_scoring),
        update_scoreboard.after(update_score),
        update_entity_positions.after(move_ball),
        move_paddles.after(move_player1_paddle),
        handle_collisions.after(move_ball),
    ))
    .run();
}
```

Resetting the Ball

If we run Cargo Run we can see when the ball moves we can see the score functionality in action. However, the score here continuously keeps increasing as the ball is out of window. So in the next lesson, we want to reset the ball so when someone has scored, the ball resets back to the center of the window.

Welcome back everyone, in this lesson let's focus on resetting the ball. In our initial `move_ball` function, we were just moving it to the right-hand side. However, to make the game more fun and less predictable, we want to randomize where the ball resets to.

Adding the Rand Package

We begin in the `Cargo.toml` file and under the dependencies, we want to ensure we have the `rand` package and specify the version we want. This `rand` package is just a Rust crate used for generating random numbers, and we'll utilize this to randomize the velocity when the ball resets.

```
[dependencies]
bevy = "0.14"
rand = "0.8"
```

Importing the Rand Function

Heading over to the `ball.rs` function, we want to ensure we're importing the `rand` function with `use rand::Rng;`.

```
use rand::Rng;
```

Defining the RespawnBall Function

We want to define the public function `RespawnBall`. This takes in two parameters:

- `ball_query`: A mutable query that retrieves the entities with both position and velocity components, and specifically those that have a ball component. This allows us to modify the position and the velocity of the ball.
- `events`: An event reader of `Scored` meaning we want to have an event reader that reads the scored events which again just indicates when a player has scored.

```
pub fn respawn_ball(
    mut ball_query: Query<(&mut Position, &mut Velocity), With>,
    mut events: EventReader,
) {
    if let Ok((mut position, mut velocity)) = ball_query.get_single_mut() {
        for event in events.read() {
            position.0 = Vec2::new(0.0, 0.0);

            let mut rng = rand::thread_rng();
            let angle: f32 = match event.0 {
                Scorer::Player1 => rng.gen_range(-std::f32::consts::PI / 4.0..std::f3
2::consts::PI / 4.0),
                Scorer::Player2 => rng.gen_range(3.0 * std::f32::consts::PI / 4.0..5.
0 * std::f32::consts::PI / 4.0),
            };
            let speed: f32 = 1.0;
            velocity.0 = Vec2::new(angle.cos() * speed, angle.sin() * speed);
        }
    }
```



```
}
```

Explanation of the RespawnBall Function

Inside the `RespawnBall` function, we perform the following steps:

1. Check if we can get a single mutable reference to the ball's position and velocity component.
2. If successful, we enter the block to process the scoring events.
3. For each event in the event reader, we reset the ball's position to the center of the screen at the (0, 0) coordinate.
4. We initialize a random number generator with `let mut rng = rand::thread_rng();`.
5. We determine the direction of the ball's velocity based on who has scored:
    - If Player 1 scored, the ball will reset at a random angle between -π/4 and π/4 radians.
    - If Player 2 scored, the ball will reset at a random angle between 3π/4 and 5π/4 radians.
6. We define the speed of the ball to be 1.0.
7. We calculate the new velocity vector for the ball based on the random angle and speed and assign it to the ball's velocity.

Adding the RespawnBall System

Once this is all done, we move on to the `main` function and within the function of `add_systems_of_update`, we want to say we want to respawn the ball after we've detected a scoring meaning if someone has scored we want to reset the ball.

```
.add_systems(Update, (
    move_ball,
    move_player1_paddle,
    detect_scoring,
    move_player2_paddle,
    respawn_ball.after(detect_scoring),
    update_score.after(detect_scoring),
    update_scoreboard.after(update_score),
    update_entity_positions.after(move_ball),
    move_paddles.after(move_player1_paddle),
    handle_collisions.after(move_ball),
))
```

Testing the Implementation

When we run `cargo run` again, we can see that the ball is moving as usual. And when we move the paddle to miss, meaning the other player scores, the score updates by one now, and the ball resets to the center and moves at a random angle.

The one thing we can see from here is when the ball leaves the screen from the top of the screen or the bottom, the score gets updated. However, this is a slight flaw as the two paddles will never be able to reach these spaces to save the ball. So this is when the boundary comes in, which we'll be creating in the next lesson.

Welcome back everyone. In this lesson, we will create the necessary boundaries for our game. Let's begin by making the necessary imports to create our boundary in our collisions.rs file.

Imports

First, we need to import the required modules and constants. We will use the following imports:

- The constant for the boundary height.
- The `Mesh2dBundle` and `MaterialMesh2dBundle` for creating the boundary box.
- The `BoundaryBundle` for spawning the boundary.

```
use crate::constants::BOUNDARY_HEIGHT;
use bevy::sprite::MaterialMesh2dBundle;
use crate::bundles::BoundaryBundle;
```

Creating the Spawn Boundary Function

Next, we will create a public function called `spawn_boundary`. This function will have multiple parameters:

- `mut commands`: Used to create and manage entities and their components.
- `mut meshes`: An immutable resource for managing mesh assets.
- `mut materials`: An immutable resource for managing color material assets.
- `window_query`: A query that retrieves the window component.

```
pub fn spawn_boundary(
    mut commands: Commands,
    mut meshes: ResMut<Assets>,
    mut materials: ResMut<Assets>,
    window_query: Query,
) {
    if let Ok(window) = window_query.get_single() {
        let window_width = window.resolution.width();
        let window_height = window.resolution.height();

        let top_boundary_y = window_height / 2.0 - BOUNDARY_HEIGHT / 2.0;
        let bottom_boundary_y = -window_height / 2.0 + BOUNDARY_HEIGHT / 2.0;

        let top_boundary = BoundaryBundle::new(0.0, top_boundary_y, window_width);
        let bottom_boundary = BoundaryBundle::new(0.0, bottom_boundary_y, window_widt
h);

        let mesh = Mesh::from(Rectangle::from_size(top_boundary.shape.0));
        let material = ColorMaterial::from(Color::srgb(0.0, 0.0, 0.0));

        let mesh_handle = meshes.add(mesh);
        let material_handle = materials.add(material);

        commands.spawn((
            top_boundary,
            MaterialMesh2dBundle {
                mesh: mesh_handle.clone().into(),
                material: material_handle.clone(),
```



```
                ..default()
            },
        ));

        commands.spawn((
            bottom_boundary,
            MaterialMesh2dBundle {
                mesh: mesh_handle.clone().into(),
                material: material_handle.clone(),
                ..default()
            },
        ));
    }
}
```

Explanation

Let’s break down the steps in the `spawn_boundary` function:

1. We retrieve the window dimensions using the `window_query`.
2. We calculate the y-coordinates for the top and bottom boundaries.
3. We create the boundary bundles for the top and bottom boundaries using the `BoundaryBundle` struct.
4. We create a mesh for the boundary using the size of the boundary bundle.
5. We create a black color material for the boundary.
6. We add the mesh and material to the asset storage and return handles to them.
7. We spawn the top and bottom boundaries using the `commands.spawn` functionality, adding the necessary components.

Adding the Spawn Boundary Function to the Startup System

Finally, we need to add the `spawn_boundary` function to the startup system so that it runs at the initial loading of the game.

```
fn main() {
    App::new()
        .add_plugins(DefaultPlugins.set(create_window()))
        .init_resource::()
        .add_event::()
        .add_systems(Startup, (spawn_dotted_line, spawn_ball, spawn_paddles, spawn_ca
mera, spawn_scoreboard, spawn_boundary))
        .add_systems(Update, (
            move_ball,
            move_player1_paddle,
            detect_scoring,
            move_player2_paddle,
            respawn_ball.after(detect_scoring),
            update_score.after(detect_scoring),
            update_scoreboard.after(update_score),
            update_entity_positions.after(move_ball),
            move_paddles.after(move_player1_paddle),
            handle_collisions.after(move_ball),
        ))
```

```
        .run();
}
```

By following these steps, we have successfully created the boundaries for our game. When you run the game, you should see the ball bouncing off the top and bottom boundaries instead of leaving the screen.

Congratulations on completing your course! You have now gained a deeper understanding of Rust and the fundamentals of the Bevy game engine. Throughout this course, you have applied your knowledge to create a mini Pong game where you control a paddle and play against the computer. This is just the beginning of your journey with Bevy, and there are numerous ways you can enhance your project.

Enhancing Your Pong Game

- Improve collision handling for more realistic gameplay.
- Add sound effects to make the game more engaging.
- Introduce visual effects to enhance the gaming experience.
- Develop an AI player where the computer controls the other paddle, adding a new level of challenge.

About Zenva

Zenva is an online learning academy with over a million learners. We offer a wide range of courses designed for both beginners and those looking to learn something new. Our courses are versatile and cater to various learning styles, allowing you to:

1. Watch video tutorials for a visual learning experience.
2. Read lesson summaries for quick reference and understanding.
3. Follow along with the instructor using the included project files for hands-on practice.

Thank you for watching, and we wish you the best on your Bevy journey. Continue exploring and expanding your skills with Zenva’s comprehensive resources.

In this lesson, you can find the full source code for the project used within the course. Feel free to use this lesson as needed – whether to help you debug your code, use it as a reference later, or expand the project to practice your skills!

You can also download all project files from the **Course Files** tab via the project files or downloadable PDF summary.

bundles.rs

Found in folder: **/src**

This code defines three bundles of components in a Bevy game: `BallBundle`, `PaddleBundle`, and `BoundaryBundle`. Each bundle represents a game entity and contains a set of components that define its properties. The `BallBundle` has components for the ball's shape, velocity, and position. The `PaddleBundle` has components for the paddle's shape, position, and velocity. The `BoundaryBundle` has components for the boundary's shape and position. Each bundle has a `new` method that creates a new instance of the bundle with initial values for its components.

```
use bevy::prelude::*;
use bevy::math::Vec2;
use crate::components::*;
use crate::constants::*;

#[derive(Bundle)]
pub struct BallBundle{
    pub ball:Ball,
    pub shape: Shape,
    pub velocity: Velocity,
    pub position: Position,
}

impl BallBundle{
    pub fn new(x:f32, y:f32) -> Self{
        Self{
            ball: Ball,
            shape: Shape(Vec2::new(BALL_SIZE, BALL_SIZE)),
            velocity: Velocity(Vec2::new(x,y)),
            position: Position(Vec2::new(0.,0.)),
        }
    }
}

#[derive(Bundle)]
pub struct PaddleBundle{
    pub paddle: Paddle,
    pub shape: Shape,
    pub position: Position,
    pub velocity: Velocity,
}

impl PaddleBundle{
    pub fn new(x:f32, y:f32) -> Self{
        Self{
            paddle: Paddle,
            shape: Shape(Vec2::new(PADDLE_WIDTH, PADDLE_HEIGHT)),
```



```
            position: Position(Vec2::new(x,y)),
            velocity: Velocity(Vec2::new(0.,0.)),
        }
    }
}

#[derive(Bundle)]
pub struct BoundaryBundle{
    pub boundary: Boundary,
    pub shape: Shape,
    pub position: Position,
}

impl BoundaryBundle{
    pub fn new(x:f32, y:f32, width:f32) -> Self{
        Self{
            boundary: Boundary,
            shape: Shape(Vec2::new(width, BOUNDARY_HEIGHT)),
            position: Position(Vec2::new(x,y)),
        }
    }
}
```

components.rs

Found in folder: **/src**

This code defines various components and resources for a game. The components include `Player1` and `Player2` to identify the players, `Player1Score` and `Player2Score` to track their scores, `Ball`, `Paddle`, and `Boundary` to represent game objects, and `Position`, `Velocity`, and `Shape` to store their physical properties. The code also defines an `Scored` event to notify when a player scores, and a `Score` resource to keep track of the current score.

```
use bevy::prelude::*;
use bevy::math::Vec2;

#[derive(Component)]
pub struct Player1;

#[derive(Component)]
pub struct Player2;

#[derive(Component)]
pub struct Player1Score;

#[derive(Component)]
pub struct Player2Score;

pub enum Scorer{
    Player1, Player2
}

#[derive(Component)]
```

```
pub struct Ball;

#[derive(Component)]
pub struct Paddle;

#[derive(Component)]
pub struct Boundary;

#[derive(Component)]
pub struct Position(pub Vec2);

#[derive(Component)]
pub struct Velocity(pub Vec2);

#[derive(Component)]
pub struct Shape(pub Vec2);

#[derive(Event)]
pub struct Scored(pub Scorer);

#[derive(Resource, Default)]
pub struct Score{
    pub player1: u32,
    pub player2: u32,
}
```

constants.rs

Found in folder: **/src**

This code defines a set of constants that represent various dimensions and speeds in a game. The constants include the size of a ball, the width and height of a paddle, the speed of the ball and paddle, and the height of a boundary. The constants also include the height of the game window to configure the game's layout and physics. By defining these values as constants, they can be easily referenced and used throughout the game code, making it easier to modify and balance the game's behavior.

```
pub const BALL_SIZE: f32 = 5.;
pub const PADDLE_WIDTH: f32 = 10.;
pub const PADDLE_HEIGHT: f32 = 50.;

pub const BALL_SPEED: f32 = 5.;
pub const PADDLE_SPEED: f32 = 5.;

pub const BOUNDARY_HEIGHT: f32 = 20.;

pub const WINDOW_HEIGHT: f32 = 600.;
```

main.rs

Found in folder: **/src**

This code sets up the main application for a game using the Bevy game engine. It initializes the game window and sets up various systems, events, and resources that will be used throughout the game. The `main` function creates a new Bevy app and adds several plugins, including default plugins with a custom window configuration.

The app also sets up two stages of systems: `Startup` and `Update`. The `Startup` stage runs once at the beginning of the game and is responsible for spawning various game entities, such as the ball, paddles, camera, scoreboard, and boundary. The `Update` stage runs repeatedly during the game and handles tasks such as moving the ball and paddles, detecting scoring, updating the score, and handling collisions. The systems in the `Update` stage are ordered to ensure that certain tasks are performed after others, using the `after` method.

```
use bevy::prelude::*;
mod systems;
mod constants;
mod components;
mod bundles;

use systems::*;
use components::*;
use bundles::*;

fn main() {
    App::new()
    .add_plugins(DefaultPlugins.set(create_window()))
    .init_resource::<Score>()
    .add_event::<Scored>()
    .add_systems(Startup, (spawn_dotted_line, spawn_ball, spawn_paddles, spawn_camera
, spawn_scoreboard, spawn_boundary))
    .add_systems(Update, (
        move_ball,
        move_player1_paddle,
        detect_scoring,
        move_player2_paddle,
        respawn_ball.after(detect_scoring),
        update_score.after(detect_scoring),
        update_scoreboard.after(update_score),
        update_entity_positions.after(move_ball),
        move_paddles.after(move_player1_paddle),
        handle_collisions.after(move_ball),
    ))
    .run();
}
```

ball.rs

Found in folder: **/src/systems**

This code is responsible for managing the behavior of a ball in a game. It includes functions for spawning a new ball, moving the ball, updating its position, detecting when a score is made, and respawning the ball after a score.

```
use bevy::prelude::*;
```



```
use crate::components::*;
use crate::constants::*;
use crate::BallBundle;
use bevy::sprite::MaterialMesh2dBundle;
use rand::Rng;

pub fn spawn_ball(
    mut commands: Commands,
    mut meshes: ResMut<Assets<Mesh>>,
    mut materials: ResMut<Assets<ColorMaterial>>,
){
    let shape = Mesh::from(Circle::new(BALL_SIZE));
    let color = ColorMaterial::from(Color::srgb(1.0, 0.0, 0.0));

    let mesh_handle = meshes.add(shape);
    let material_handle = materials.add(color);

    commands.spawn((
        BallBundle::new(1.0, 0.0),
        MaterialMesh2dBundle{
            mesh:mesh_handle.into(),
            material:material_handle,
            ..default()
        }
    ));
}

pub fn move_ball(
    mut ball: Query<(&mut Position, &Velocity), With<Ball>>
){
    if let Ok((mut position, velocity)) = ball.get_single_mut(){
        position.0 += velocity.0 * BALL_SPEED;
    }
}

pub fn update_entity_positions(
    mut ball: Query <(&mut Transform, &Position)>
){
    for (mut transform, position) in ball.iter_mut(){
        transform.translation = position.0.extend(0.);
    }
}


pub fn detect_scoring(
    mut ball_query: Query<&mut Position, With<Ball>>,
    window_query: Query<&Window>,
    mut score_event_write: EventWriter<Scored>,
){
    let window = match window_query.get_single(){
    Ok(w) => w,
    Err(_) => return,
    };

    let half_window_width = window.resolution.width()/ 2.0;
```

```
    let ball_position = match ball_query.get_single_mut(){
        Ok(bp) => bp,
        Err(_) => return,
    };

    if ball_position.0.x > half_window_width{
        score_event_write.send(Scored(Scorer::Player2));
    } else if ball_position.0.x < -half_window_width{
        score_event_write.send(Scored(Scorer::Player1));
    }
}

pub fn respawn_ball(
    mut ball_query: Query<(&mut Position, &mut Velocity), With<Ball>>,
    mut events: EventReader<Scored>,
){
    if let Ok((mut position, mut velocity)) = ball_query.get_single_mut(){
        for event in events.read(){
            position.0 = Vec2::new(0.0, 0.0);

            let mut rng = rand::thread_rng();
            let angle: f32 = match event.0{
                Scorer::Player1 => rng.gen_range(-std::f32::consts::PI / 4.0..std::f3
2::consts::PI/4.0),
                Scorer::Player2 => rng.gen_range(3.0 * std::f32::consts::PI/4.0..5.0
* std::f32::consts::PI/4.0),
            };
            let speed: f32 = 1.0;
            velocity.0 = Vec2::new(angle.cos()*speed, angle.sin()*speed);
        }
    }
}
```

camera.rs

Found in folder: **/src/systems**

This code defines a function called `spawn_camera` that creates a new camera entity in the game world. When called, it uses the `Commands` system to spawn a `Camera2dBundle`, which is a pre-configured bundle of components that represents a 2D camera. The `Camera2dBundle` is created with its default settings, which means it will have a standard set of properties and behaviors for a 2D camera. By spawning this bundle, the function effectively adds a new camera to the game world, which can be used to render and display the game's 2D graphics.

```
use bevy::prelude::*;

pub fn spawn_camera(mut commands: Commands){
    commands.spawn(Camera2dBundle::default());
}
```

collision.rs

Found in folder: **/src/systems**

This code manages collisions in a 2D game by using the `handle_collisions` function, which detects collisions between a ball and other objects and calls `respond_to_collision` to adjust the ball's velocity. Collisions are detected by the `detect_collision` function, which uses bounding shapes to determine if and where an intersection occurs, then informs which direction to reverse the ball's velocity. The `spawn_boundary` function creates boundaries at the top and bottom of the game window for collision detection with the ball.

```
use bevy::prelude::*;
use crate::components::*;
use bevy::math::{ bounding:: {Aabb2d, BoundingCircle, BoundingVolume, IntersectsVolum
e}};
use crate::constants::BOUNDARY_HEIGHT;
use bevy::sprite::MaterialMesh2dBundle;
use crate::BoundaryBundle;

#[derive(Debug, PartialEq, Eq, Copy, Clone)]
pub enum Collision{
    Left, Right, Top, Bottom,
}

pub fn handle_collisions(
    mut ball_query: Query<(&mut Velocity, &Position, &Shape), With<Ball>>,
    other_things: Query<(&Position, &Shape), Without<Ball>>,
){
    if let Ok((mut ball_velocity, ball_position, ball_shape)) = ball_query.get_single
_mut(){
        for (position, shape) in &other_things{
            if let Some(collision) =  detect_collision(ball_position, ball_shape, pos
ition, shape){
                respond_to_collision(&mut ball_velocity, collision);
            }
        }
    }
}

fn detect_collision(
    ball_position: &Position,
    ball_shape: &Shape,
    other_position: &Position,
    other_shape: &Shape,
) -> Option<Collision>{
    let ball = BoundingCircle::new(ball_position.0, ball_shape.0.x);
    let wall = Aabb2d::new(other_position.0, other_shape.0/2.);

    if !ball.intersects(&wall){
        return None;
    }

    let closest_point = wall.closest_point(ball.center());
    let offset = ball.center() - closest_point;

    if offset.x.abs() > offset.y.abs(){
        if offset.x < 0. {
```



```
            Some(Collision::Left)
        } else {
            Some(Collision::Right)
        }
    } else {
        if offset.y < 0. {
            Some(Collision::Top)
        } else {
            Some(Collision::Bottom)
        }
    }
}

fn respond_to_collision(velocity: &mut Velocity, collision: Collision){
    match collision{
        Collision::Left | Collision::Right => velocity.0.x *= -1.,
        Collision::Top | Collision::Bottom => velocity.0.y *= -1.,

    }
}

pub fn spawn_boundary(
    mut commands: Commands,
    mut meshes: ResMut<Assets<Mesh>>,
    mut materials: ResMut<Assets<ColorMaterial>>,
    window: Query<&Window>,
){
    if let Ok(window) = window.get_single(){
        let window_width = window.resolution.width();
        let window_height = window.resolution.height();

        let top_boundary_y = window_height/2. - BOUNDARY_HEIGHT/2.;
        let bottom_boundary_y = -window_height/2. + BOUNDARY_HEIGHT/2.;

        let top_boundary = BoundaryBundle::new(0., top_boundary_y, window_width);
        let bottom_boundary = BoundaryBundle::new(0., bottom_boundary_y, window_width
);

        let mesh = Mesh::from(Rectangle::from_size(top_boundary.shape.0));
        let material = ColorMaterial::from(Color::srgb(0., 0., 0.));

        let mesh_handle = meshes.add(mesh);
        let material_handle = materials.add(material);

        commands.spawn((
            top_boundary,
            MaterialMesh2dBundle{
                mesh: mesh_handle.clone().into(),
                material: material_handle.clone(),
                ..default()
            },
        ));

        commands.spawn((
            bottom_boundary,
```

```
            MaterialMesh2dBundle{
                mesh: mesh_handle.clone().into(),
                material: material_handle.clone(),
                ..default()
            },
        ));

    }
}
```

mod.rs

Found in folder: **/src/systems**

This code defines and exports several modules in a Rust program. The modules are named `ball`, `paddle`, `camera`, `window`, `collision`, and `scoreboard`. Each module is declared with the `pub mod` keyword, making them publicly accessible. The code also re-exports all public items from each module using the `pub use` keyword. This allows other parts of the program to access the contents of these modules without having to navigate through the module hierarchy.

```
pub mod ball;
pub mod paddle;
pub mod camera;
pub mod window;
pub mod collision;
pub mod scoreboard;

pub use ball::*;
pub use paddle::*;
pub use camera::*;
pub use window::*;
pub use collision::*;
pub use scoreboard::*;
```

paddle.rs

Found in folder: **/src/systems**

This code is responsible for spawning and moving paddles in a game. The `spawn_paddles` function creates two paddles, one for each player, and positions them on either side of the game window. Each paddle is given a mesh, material, and initial position. The `move_player1_paddle` and `move_player2_paddle` functions update the velocity of each paddle based on keyboard input. Player 1’s paddle is controlled by the up and down arrow keys, while Player 2’s paddle is controlled by the W and S keys. The `move_paddles` function then updates the position of each paddle based on its velocity, ensuring that the paddles do not move off the screen.

```
use bevy::prelude::*;
use crate::components::*;
use crate::constants::*;
use crate::PaddleBundle;
use bevy::sprite::MaterialMesh2dBundle;
```



```
pub fn spawn_paddles(
    mut commands: Commands,
    mut meshes: ResMut<Assets<Mesh>>,
    mut materials: ResMut<Assets<ColorMaterial>>,
    window: Query<&Window>,
){
    if let Ok(window) = window.get_single(){
        let window_width = window.resolution.width();
        let padding = 50.;
        let right_paddle_x = window_width/2. - padding;
        let left_paddle_x = -window_width/2. + padding;

        let mesh = Mesh::from(Rectangle::new(PADDLE_WIDTH, PADDLE_HEIGHT));
        let mesh_handle = meshes.add(mesh);

        commands.spawn((
            Player1,
            PaddleBundle::new(right_paddle_x, 0.),
            MaterialMesh2dBundle{
                mesh: mesh_handle.clone().into(),
                material: materials.add(
                    ColorMaterial::from(Color::srgb(0.0, 1.0, 0.0))
                ),
                ..default()
            }
        ));

        commands.spawn((
            Player2,
            PaddleBundle::new(left_paddle_x, 0.),
            MaterialMesh2dBundle{
                mesh: mesh_handle.into(),
                material: materials.add(
                    ColorMaterial::from(Color::srgb(0.0, 0.0, 1.0))
                ),
                ..default()
            }
        ));

    }
}


pub fn move_player1_paddle(
    keyboard_input: Res<ButtonInput<KeyCode>>,
    mut paddle: Query<&mut Velocity, With<Player1>>,
){
    if let Ok(mut velocity)=
    paddle.get_single_mut(){
        if keyboard_input.pressed(KeyCode::ArrowUp){
            velocity.0.y = 1.;
        } else if keyboard_input.pressed(KeyCode::ArrowDown){
            velocity.0.y = -1.0;
        } else {
```

```
            velocity.0.y = 0.0;
        }
    }
}

pub fn move_player2_paddle(
    keyboard_input: Res<ButtonInput<KeyCode>>,
    mut paddle: Query<&mut Velocity, With<Player2>>,
){
    if let Ok(mut velocity)=
    paddle.get_single_mut(){
        if keyboard_input.pressed(KeyCode::KeyW){
            velocity.0.y = 1.;
        } else if keyboard_input.pressed(KeyCode::KeyS){
            velocity.0.y = -1.0;
        } else {
            velocity.0.y = 0.0;
        }
    }
}

pub fn move_paddles(
    mut paddle: Query<(&mut Position, &Velocity), With<Paddle>>,
    window: Query<&Window>,
){
    if let Ok(window) = window.get_single(){
        let window_height = window.resolution.height();

        for (mut position, velocity) in &mut paddle{
            let new_position = position.0 + velocity.0 * PADDLE_SPEED;
            if new_position.y.abs() < window_height / 2.0 - PADDLE_HEIGHT / 2.0 {
                position.0 = new_position;
            }
        }
    }
}
```

scoreboard.rs

Found in folder: **/src/systems**

This code is responsible for managing the scoreboard in a game. The `spawn_scoreboard` function creates two text elements, one for each player's score, and positions them at the top of the screen. The text elements are initialized with a score of "0" and are styled with a large font size and white color. The `update_scoreboard` function updates the text elements with the current score whenever the score changes. It retrieves the current score from a resource and updates the text elements accordingly. The `update_score` function increments the score for the player who scored, based on events received from the game. It also prints the current score to the console.

```
use bevy::prelude::*;
use crate:: components::*;

pub fn spawn_scoreboard(
```

```
    mut commands: Commands,
){
    commands.spawn((
        TextBundle::from_section(
            "0",
            TextStyle{
                font_size: 64.0,
                color: Color:: WHITE,
                ..default()
            },
        )
        .with_text_justify(JustifyText::Center)
        .with_style(Style{
            position_type: PositionType::Absolute,
            top: Val::Px(5.0),
            right: Val::Percent(45.0),
            ..default()
        }),
        Player1Score
    ));

    commands.spawn((
        TextBundle::from_section(
            "0",
            TextStyle{
                font_size: 64.0,
                color: Color:: WHITE,
                ..default()
            },
        )
        .with_text_justify(JustifyText::Center)
        .with_style(Style{
            position_type: PositionType::Absolute,
            top: Val::Px(5.0),
            left: Val::Percent(45.0),
            ..default()
        }),
        Player2Score
    ));
}

pub fn update_scoreboard(
    mut player1_score: Query<&mut Text, With<Player1Score>>,
    mut player2_score: Query<&mut Text, (With<Player2Score>, Without<Player1Score>)>,
    score: Res<Score>,
){
    if score.is_changed(){
        if let Ok(mut player1_score) = player1_score.get_single_mut(){
            player1_score.sections[0].value = score.player1.to_string();
        }
        if let Ok(mut player2_score) = player2_score.get_single_mut(){
            player2_score.sections[0].value = score.player2.to_string();
        }

    }
```

```
}

pub fn update_score(
    mut score: ResMut<Score>,
    mut events: EventReader<Scored>,
){
    for event in events.read(){
        match event.0{
            Scorer::Player1 => score.player1 +=1,
            Scorer::Player2 => score.player2 +=1,
        }
    }
    println!("Score: {} - {}", score.player1, score.player2);

}
```

window.rs

Found in folder: **/src/systems**

This code creates a window for a game and sets up a visual element within it. The `create_window` function returns a `WindowPlugin` that configures the primary window with a title of “My Pong Game”. The `spawn_dotted_line` function creates a series of white dots, spaced evenly apart, that run vertically down the center of the window. The dots are created using a loop that calculates the position and size of each dot based on the window’s height and a set gap size between dots. Each dot is then added to the game world using the `Commands` system.

```
use bevy::prelude::*;
use crate::constants;


pub fn create_window() -> WindowPlugin{
    WindowPlugin{
        primary_window: Some(Window{
            title: "My Pong Game".to_string(),
            ..default()
        }),
        ..default()
    }
}

pub fn spawn_dotted_line(mut commands: Commands){
    let dot_color = Color::srgb(1.0, 1.0, 1.0);
    let dot_size = Vec3::new(5.0, 20.0, 1.0);
    let gap_size = 10.0;
    let num_dots = (constants::WINDOW_HEIGHT/ (dot_size.y + gap_size)) as i32;

    for i in 0..num_dots{
        commands.spawn(SpriteBundle{
            sprite: Sprite{
                color:dot_color,
                ..default()
            },
```

```
            transform: Transform {
                translation: Vec3::new(0.0, i as f32 * (dot_size.y + gap_size) - cons
tants::WINDOW_HEIGHT/2.0, 0.0),
                scale: dot_size,
                ..default()
            },
            ..default()
        });
    }
}
```

Cargo.toml

Found in folder: **/**

This code defines the metadata for a Rust package. It specifies the name of the package as “my_bevy_game” and its version as “0.1.0”. The code also declares the dependencies required by the package. It depends on two external crates: “bevy” version “0.14” and “rand” version “0.8”. This means that when the package is built, these crates will be automatically downloaded and included in the build process.

```
[package]
name = "my_bevy_game"
version = "0.1.0"
edition = "2021"

[dependencies]
bevy = "0.14"
rand = "0.8"
```