Project Increment 1 Steps

Overview

To give you some help in approaching your work for Increment 1, I've provided the steps I implemented when building my Increment 1 solution. You'll notice that these steps are pretty detailed. I'm doing this for the first couple increments so you can get up to speed on Unity and C#, but the steps I give you for later project increments won't be nearly as detailed. That's because you're maturing (at least as programmers <grin>), so you should be able to figure out more of the details on your own.

The required functionality for this increment is "basic gameplay with two paddles and a single ball. The ball should NOT expire after a given period of time and you don't need to spawn a new ball when the current ball leaves the playfield. The ball should start in the center of the screen and start moving in a random direction, limited to angles between -45 and 45 degrees and 135 and 225 degrees. The ball should also bounce off the front of each paddle properly, bounce around the playfield properly, and be destroyed when it leaves the playfield."

Step 0: Downloading Starting Materials

To start your work, download the Project Increment 1 Materials.zip file from Canvas (or the burningteddy.com/boulder) and extract the zip file somewhere. The zip file contains your starting Wacky Pong Unity project and the text of a method you'll use in Step 8 below.

Step 1: Open and Set Up the Unity project

For this step, you're opening and setting up the provided Unity project.

1. Open the WackyPong project with Unity.
2. Select the Main Camera object and click the Background color picker in the Camera component in the Inspector. Change the color to whatever you want your background color to be in the game.
3. Select Edit > Project Settings > Physics 2D from the top menu bar and set the Y component of Gravity to 0.
4. Save and exit Unity.

When you run your game, the Game view should just be the background color you selected above.

Step 2: Add Edge Colliders to the Camera

For this step, you're adding edge colliders for the top and bottom sides of the camera view. You're doing this so the balls will bounce around the playfield properly.

1. Select the Main Camera object and click the Add Component button at the bottom of the Inspector. Select Physics 2D > Edge Collider 2D. Click the Edit Collider button in the

new component in the Inspector and drag the end points of the edge collider to be at the top edge of the camera view. Click the Edit Collider button in the Inspector again to stop editing the collider.
2. Add an edge collider for the bottom edge of the camera view.
3. Next, we'll add Physics Material 2D to the edge colliders to make them "bouncy". Create a new materials folder in the Project window. Add a new Physics Material 2D to that folder, named EdgeMaterial. Select the EdgeMaterial and change Friction to 0 and Bounciness to 1 in the Inspector. Select the Main Camera object and drag the EdgeMaterial onto the Material field of each of the Edge Collider 2D components in the Inspector.

Step 3: Add the Paddles

For this step, you're adding the paddles to the game.

1. Add sprites for the paddles to the sprites folder in the Project window. I used the same sprite for both paddles, but you can use different sprites for the left and right paddles if you prefer. I haven't provided any sprites for you, so you'll have to draw or find them yourself.
2. Drag the paddle sprite from the sprites folder in the Project window onto the Hierarchy window.
3. Rename the resulting game object LeftPaddle. Change the position of the paddle to be centered vertically and slightly to the right of the left side of the scene. Attach the Paddle script to the LeftPaddle game object.
4. Follow similar steps to add the right paddle to the scene.

When you run your game, you should see the paddles placed properly in the game.

Step 4: Move the Paddles

For this step, you're letting the left and right players move their paddles. The steps given below work for both paddles.

1. Add Rigidbody 2D components to your LeftPaddle and RightPaddle game objects and change the Body Type to Kinematic. This means we'll be changing the position of the rigidbody ourselves rather than adding forces to move it. Constrain the component so it can't rotate in z.
2. Open the ConfigurationUtils script in your IDE.
3. You'll notice that the class declaration says

   ```
   public static class ConfigurationUtils
   ```

   instead of

   ```
   public class ConfigurationUtils : MonoBehaviour
   ```

This class doesn't inherit from `MonoBehaviour` (we'll discuss inheritance later in the course). We're making the class static because we don't want to attach the class to game objects to instantiate it, we just want consumers to access the class directly. The C# `Console` and `Math` classes are static classes for the same reason.

4. Open the Paddle script in your IDE.
5. Fill in the documentation comment at the top of the script.
6. Add a `Rigidbody2D` field to the class and add code to the `Start` method to get the Rigidbody 2D component attached to the game object and save it in the field. We do this for efficiency so we don't have to retrieve the component every time we want to move the paddle.
7. Add a `FixedUpdate` method to the script; read the `MonoBehaviour` documentation for this method to learn more about it. This method is called 50 times per second, and it's the appropriate place to move the game object (which we'll move by moving the rigidbody).
8. In the Unity editor, add LeftPaddle and RightPaddle input axes to the input manager. The left paddle should be moved up and down using W and S and the right paddle should be moved up and down using the up and down arrow keys.
9. Add a serialized field to hold a `ScreenSide` value and populate the field properly for the LeftPaddle and RightPaddle game objects in the Unity inspector.
10. Declare a new `Vector2` field to hold the new position when you're moving the paddle. This saves memory because we don't have to create a new `Vector2` object every time we move, we can just reuse the field.
11. Add code to the body of the `FixedUpdate` method to move the rigidbody for the paddle based on input on the appropriate input axis based on what side of the screen the paddle is on. You should use the `ConfigurationUtils PaddleMoveUnitsPerSecond` property when you calculate how far to move the paddle. Read about the `Rigidbody2D MovePosition` method to learn how to use it.

When you run your game, you should be able to move both paddles up and down using the appropriate keys.

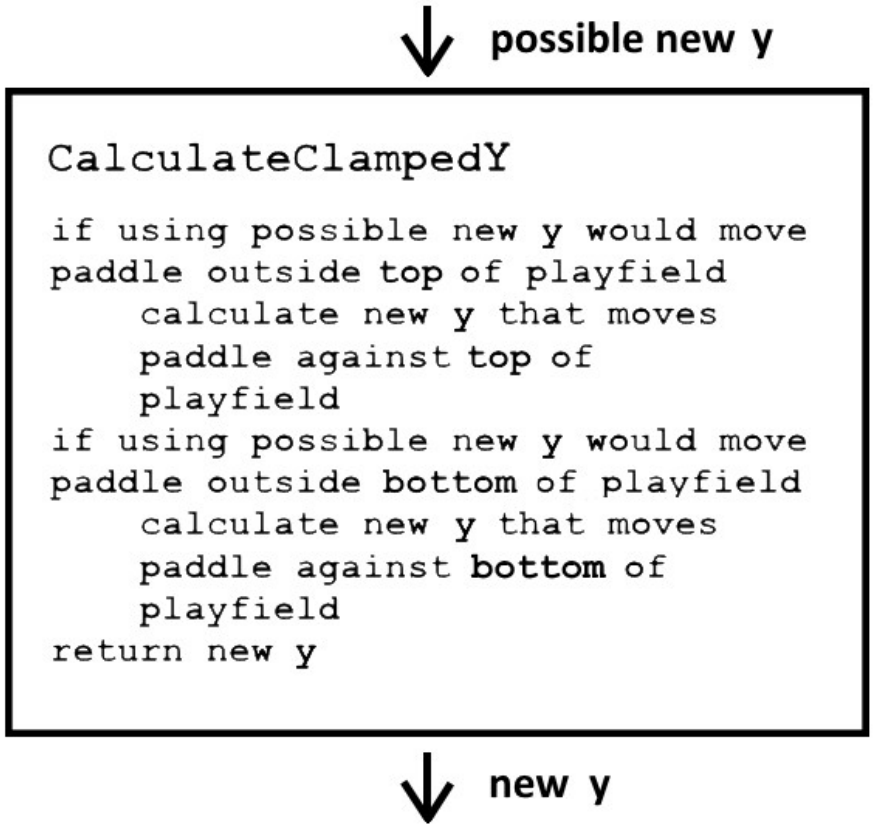
Step 5: Keep the Paddles in the Playfield

For this step, you're keeping the paddles in the playfield.

1. Add Box Collider 2D components to your LeftPaddle and RightPaddle game objects. This will let us detect when we move outside the edges of the screen and will also be helpful later when we need to detect collisions with balls in the game.
2. Open the Paddle script in your IDE and add a field to hold half the height of the collider. Add code to the `Start` method to retrieve the Box Collider 2D component, calculate half the height, and store it in your field. We do this so we don't have to retrieve the component and do the calculation whenever we need to clamp the paddle in the screen.
3. Observe that there's a GameInitializer script attached to the Main Camera. The ScreenUtils script exposes the world coordinates of the left, right, top and bottom edges of the screen. The GameInitializer script initializes the ScreenUtils script when the game starts.

4. Write a new `CalculateClampedY` method that returns a y value that takes a possible new y position for the rigidbody, shifts it if necessary to clamp the paddle to stay in the screen vertically, and returns a clamped new y position (which could just be the original new y position that was passed in if no clamping was required). Call this method if you move the game object in the `FixedUpdate` method BEFORE you call the `MovePosition` method. The "big idea" is that you have to calculate a valid new y position, including the clamping, before you call the method to move the rigidbody.

   The picture below shows what's happening with the `CalculateClampedY` method. We're passing in a possible new y position for the paddle and the method returns a new y position that guarantees the paddle will be in the playfield after it's moved. If the possible new y position would lead to the paddle being outside the playfield on the top or bottom, the method modifies the y position that was passed into the method to keep the paddle in the playfield and returns the modified y as the new y position the paddle should be moved to. If the possible new y position results in the paddle staying completely in the playfield, the method simply returns the y position that was passed in.



When you run your game, you should be able to move both paddles up and down and they should stay clamped in the screen.

You might be wondering why we need to bother with the `CalculateClampedY` method instead of just checking if the paddle is outside the playfield and moving it back into the playfield if

necessary -- in other words, correcting a movement outside the playfield instead of using the `CalculateClampedY` method to make sure we never leave the playfield. The problem is that the physics engine doesn't actually move the rigidbody until after the `FixedUpdate` method completes, so our correction would come on the frame after the paddle leaves the playfield. Unfortunately, if you try the "correct it after it happens" approach, you'll probably end up with players being able to move the paddle partially out of the playfield, especially if they keep holding the movement key down.

You also might be wondering why we didn't just collide the paddle with the edge colliders attached to the Main Camera to keep the paddle in the screen. We don't actually detect those collisions because our paddle Rigidbody 2D component is kinematic and the Main Camera doesn't have a Rigidbody 2D component attached to it. We could attach a Rigidbody 2D component to the Main Camera to solve this, but that adds processing load to the Physics 2D engine and it's not a very intuitive approach for how things really work.

Step 6: Add a ball

For this step, you're adding a ball to the scene.

1. Add a sprite for the ball to the sprites folder. I haven't provided a sprite for you, so you'll have to draw or find one yourself.
2. Drag the ball sprite from the sprites folder in the Project window onto the Hierarchy window.
3. Rename the resulting game object Ball.
4. Attach the Ball script to the Ball game object.

When you run your game, you should see the ball centered horizontally and vertically in the screen.

Step 7: Move the ball

For this step, you'll get the ball moving. Because the only thing the ball will bounce off is the top and bottom edges and the paddles, we need to start the ball moving to the left (angle between 135 degrees and 225 degrees) or to the right (angle between -45 and 45 degrees).

1. Add Rigidbody 2D (freeze rotation in z) and Box Collider 2D components to the Ball game object. I used a Box Collider 2D for my ball because it's square; if you use a different shape ball, use the appropriate Collider 2D shape for it.
2. Add a new Physics Material 2D to the materials folder, name it BallMaterial, set friction to 0 and bounciness to 1, and add the material to the Collider 2D component for the Ball.
3. Open the Ball script in your IDE.
4. Fill in the documentation comment at the top of the script.
5. Add a public static property to the `ConfigurationUtils` class to provide a `BallImpulseForce` value. For now, just return 5 from that property. If you need to change this value to something other than 5 to get the ball moving at a reasonable speed you can.

6. Add code to the `Start` method of the `Ball` class to get the ball moving. You can solve this problem however you see fit. I set min and max angles for going to the right, changed those min and max angles to go to the left instead half the time (`Random.value` helped me decide when to do that), then built and applied the force vector to get the ball moving. You have to use the `ConfigurationUtils` `BallImpulseForce` when you calculate the force vector to apply. Don't forget to convert degrees to radians at some point so sin and cos work properly.

When you run your game, the ball should move to the left and to the right approximately the same number of times each, in the appropriate range of angles for the side the ball is moving toward. Remember, random numbers can be funny (that's why we can flip a coin and get heads 10 times in a row), so if your game doesn't seem to be working, keep trying as a "sanity check".

You should also hit the ball with both paddles to make sure that's working properly.

Step 8: Curve the paddles

For this step, you're making it so you can control the angle of the ball's bounce when you hit it with a paddle.

1. Temporarily change the `Ball Start` method to always move the ball to the right.
2. Bounce the ball off the right paddle. As you can see the ball always bounces as though it's "reflecting" off the paddle. This is of course reasonable because the paddle is flat on the side, but it leads to somewhat boring gameplay. What we'd really like is to be able to control the angle of the bounce, where hitting the ball further from the center of the paddle bounces the ball off the paddle at a sharper angle. Essentially, we're treating the paddle as though it's convex even though it's not.
3. Although we could add an Edge Collider 2D component to the Paddle game object to handle this, it's hard to get that perfect and even harder if we decide to tune the curvature of the paddle later. That means we'll do it through scripting instead. Start by adding a Ball tag to the Ball game object.
4. Open the Paddle script in your IDE and add a `BounceAngleHalfRange` constant field. Set the constant value to be 60 degrees, converted to radians.
5. Copy the text from the Paddle OnCollisionEnter2D Method.txt file from the zip file into the Paddle script. This will give you a compilation error because the `Ball` class doesn't have a `SetDirection` method yet. The code I've provided to you calculates a new unit vector (a vector with magnitude 1) for the new direction the ball should move in based on where the ball hit the paddle.
6. Add a Rigidbody2D field in your Ball script and save a reference to that comment in the `Ball Start` method for efficiency. That way, we don't have to retrieve that component every time we bounce the ball off a paddle.
7. Add a `SetDirection` method to your Ball script. The method should change the direction the ball is moving in while keeping the speed the same as it was. Although we don't usually directly change the velocity of our `Rigidbody2D` components (we typically add forces to change their velocity instead) it will be WAY easier if you just change the velocity of the rigidbody attached to the ball directly in this method. Hint: the current ball

speed is just the magnitude of the rigidbody's velocity, so setting the velocity to the current speed * the new direction is exactly what you need.
8. Change the `Ball Start` method to randomly pick between moving to the left and moving to the right again.

When you run your game, you should be able to aim the ball based on where you hit it on both paddles.

Step 9: Destroy the ball

For this step, you're destroying the ball when it leaves the playfield.

1. Add an `OnBecameInvisible` method to the Ball script that destroys the Ball game object the script is attached to.
2. **Caution: Even if the ball destruction is working properly, it may not seem to be working in the Unity Editor. The best thing to do is to build the game and play the built game. If, however, you want to just stay in the editor, double click the Main Camera in the Hierarchy window, then use Middle Mouse Wheel to zoom in on the Scene view until the box that shows the edges of the camera view just disappears from view.**

When you run your game, the ball should be destroyed when it leaves the scene. You can confirm this by watching the Ball game object in the Hierarchy window get removed from the scene when the ball leaves the scene.

Turning In Your Assignment

You're required to turn in ALL of the following by the beginning of the scheduled class time on the due date:

Electronic Copy

1. Zip up your entire assignment folder into a file named <your last name and first initial>.zip. Log into Canvas and submit the file into the appropriate assignment.

Use the default Windows compression utility for this; the grader may not own WinZip, 7Zip, or whatever other program you're using to build your zip file. If you use something other than the Windows compression utility and the grader can't unzip your submission, you'll get a 0 on the assignment.

Late Turn-ins

Turn-ins are due at the beginning of the scheduled class time on the specified due date. No late turn-ins will be accepted.