Wacky Breakout Increment 1
Detailed Instructions

Overview

In this Unity project (the first increment of our Wacky Breakout game development), you're building the Unity project and adding basic gameplay to the game.

Step 1: Create the Unity project

For this step, you're creating the Unity project and getting it set up.

1. Create a new 2D Unity project called WackyBreakout.
2. Rename the SampleScene scene as scene0 and save it.
3. 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.
4. Select Edit > Project Settings > Physics 2D from the top menu bar and set the Y component of Gravity to 0.
5. 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 left, right, and top 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 and bottom of the left edge of the camera view. Click the Edit Collider button in the Inspector again to stop editing the collider.
2. Add edge colliders for the right edge and the top 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 Paddle

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

1. Create a new Sprites folder in the Project window and add a sprite for the paddle to that folder. I haven't provided a sprite for you, so you'll have to draw or find one yourself.
2. Drag the paddle sprite from the Sprites folder in the Project window onto the Hierarchy window.
3. Rename the resulting game object Paddle.
4. Change the position of the paddle to be centered horizontally and slightly above the bottom of the playfield.

When you run your game, you should see the paddle centered horizontally and slightly above the bottom of the playfield.

Step 4: Move the Paddle

For this step, you're letting the player move the paddle.

1. Add a Rigidbody 2D component to your Paddle game object 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. Create a new Scripts folder in the Project window and create a new Configuration folder in the Scripts folder. Copy the ConfigurationUtils script from the zip file into that folder. Open the script in Visual Studio.
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 a lot in the next module). We're making the class static because we don't want to attach the class to game objects of 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. Create a new Gameplay folder in the Scripts folder in the Project window. Create a C# script named Paddle in that folder. Attach the script to the Paddle game object. Open the script in Visual Studio.
5. Add a 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. Add code to the body of the `FixedUpdate` method to move the rigidbody for the paddle based on input on the Horizontal input axis (already provided in the default Unity project). 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.

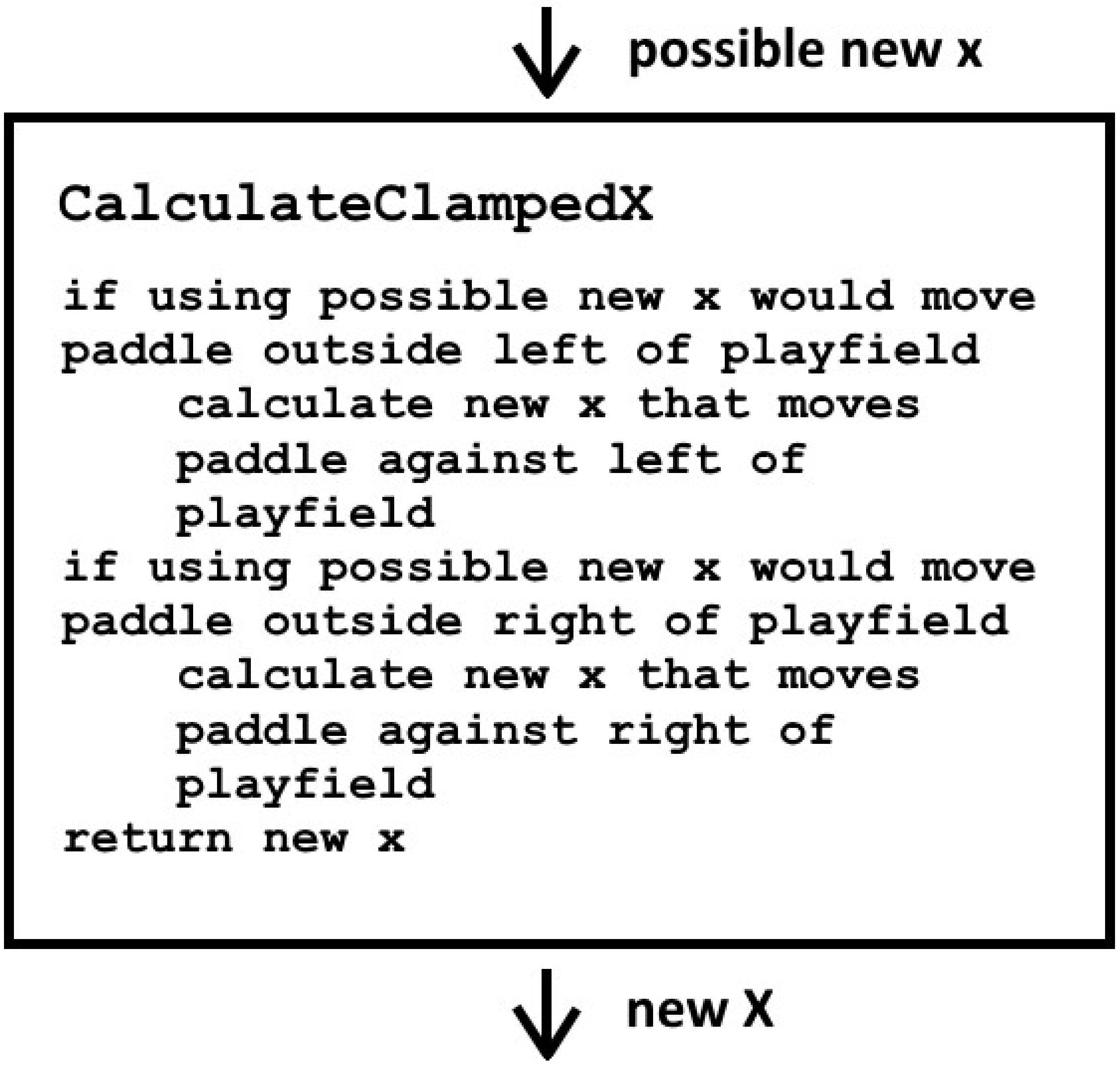
When you run your game, you should be able to move the paddle left and right.

Step 5: Keep the Paddle in the Playfield

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

1. Add a Box Collider 2D component to your Paddle game object. 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. Create a new Util folder in the Scripts folder in the Project window. Copy the GameInitializer and ScreenUtils scripts from the zip file into that folder. Attach the GameInitializer script 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.
3. Open the Paddle script in Visual Studio and add a field to hold half the width of the collider. Add code to the `Start` method to retrieve the Box Collider 2D component, calculate half the width, 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.
4. Write a new `CalculateClampedX` method that returns an x value that takes a possible new x position for the rigidbody, shifts it if necessary to clamp the paddle to stay in the screen horizontally, and returns a clamped new x position (which could just be the original new x 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 x position, including the clamping, before you call the method to move the rigidbody.

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



```
CalculateClampedX

if using possible new x would move
paddle outside left of playfield
    calculate new x that moves
    paddle against left of
    playfield
if using possible new x would move
paddle outside right of playfield
    calculate new x that moves
    paddle against right of
    playfield
return new x
```

When you run your game, you should be able to move the paddle left and right.

You might be wondering why we need to bother with the `CalculateClampedX` 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 `CalculateClampedX` 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 the player 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. Change the position of the ball to be centered horizontally and about 1/3 of the way up from the bottom of the screen.

When you run your game, you should see the ball centered horizontally and about 1/3 of the way up from the bottom of 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 at this point is the left, right, and top edges, you'll temporarily start it by send it up and to the right. In the next step we'll change the ball to make it start by moving down.

1. Add Rigidbody 2D and Box Collider 2D components to the Ball game object.
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 Box Collider 2D component for the Ball.
3. Add a new Ball script to the Gameplay folder and attach the script to the Ball game object. Open the script in Visual Studio.
4. Add a documentation comment to the script.
5. Add a public static property to the `ConfigurationUtils` class to provide a `BallImpulseForce` value. For now, just return 200 from that property.
6. Add code to the `Start` method of the `Ball` class to get the ball moving at a 20 degree angle. Use the `ConfigurationUtils BallImpulseForce` when you calculate the force vector to apply.

When you run your game, the ball should move up and to the right so it bounces off the right, top, and left edges of the scene before leaving the bottom of the screen (it might hit the right edge again before leaving the screen).

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

Step 8: Curve the paddle

For this step, you're making it so you can hit the ball with the paddle.

1. Change the `Start` method in the Ball script to start the ball moving straight down.
2. Bounce the ball off the paddle. As you can see, the ball always bounces straight back up. This is of course reasonable because the paddle is flat on the top, 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 Visual Studio 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 `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.

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

Step 9: Hit the blocks

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

1. Add some sprites for several different blocks to the Sprites folder. I haven't provided sprites for you, so you'll have to draw or find them yourself.
2. Drag one of the sprites into the Hierarchy window, rename it to StandardBlock, and add a Box Collider 2D component to it.
3. Create a new Prefabs folder in the Project window and create a prefab from the StandardBlock game object.
4. Build a single row of the prefab blocks about 1/3 of the way down from the top of the screen. Don't worry if there are small gaps on the left and right sides of the screen. Change some of the block instances in the scene so they use a variety of your block sprites. Don't worry, we'll automate this process later!

When you run your game, you should be able to bounce the ball off the blocks.

Step 10: Break the blocks

For this step, you're removing blocks that are hit by the ball from the game.

1. Add a Block script to the Gameplay folder and attach the script to your StandardBlock prefab. Open the script in Visual Studio.
2. Add a documentation comment to the script.
3. Add an `OnCollisionEnter2D` method to the script. The method should destroy the block if the collision is with a Ball game object.
4. If you hit a block just right you can get the Ball spinning, which we definitely don't want! Constrain the Rigidbody 2D component attached to the Ball to Freeze Rotation in Z

When you run your game, you should be able to destroy blocks by hitting them with the ball.

Step 11: Fix the side collision bug

For this step, you're fixing the side collision bug (which you might not have noticed!). At this point, you can actually get strange behavior from the ball if you hit it with the side of the paddle rather than the top. That's because the `OnCollisionEnter2D` method I gave you for the Paddle script doesn't make sure the collision occurred on the top of the paddle, though it should.

1. Write a new method in the Paddle script that returns true for a top collision and false otherwise. Think about the relative locations of the ball and the paddle to figure out the required logic for this method. You can even draw pictures if you need to! The `Collider2D` object passed in as a parameter for the `OnCollisionEnter2D` method provides some useful information, especially the contacts for the collision. When comparing floats, you don't really want to use == for your comparison, you'll want to check if the two floats are within some tolerance of each other (I used 0.05f for my tolerance). Note: I added a `halfColliderHeight` field to help with this.
2. Change the Boolean expression for the if statement in the `OnCollisionEnter2D` method to check for both the ball tag and a top collision (using the method you just wrote).

When you run your game, bouncing the ball off the side of the paddle should work intuitively, with the ball continuing to move down instead of bouncing up from the side. You might need to temporarily slow the ball down so you can check this.

Step 12: Add configuration data file processing

For this step, you're making the `ConfigurationData` class read from a configuration data file and use the values from that file for its properties.

1. Copy the ConfigurationData script I provided in the zip file into your Configuration folder. The `ConfigurationData` class acts as a container for all the configuration data. That data may come from a configuration data file or, if reading that file fails, from default values hard-coded into the class. The good news is that none of the other classes have to care where the data came from, they just use it.

2. Open the ConfigurationUtils script in Visual Studio and add a static field to hold a `ConfigurationData` object.
3. Add code to the `ConfigurationUtils Initialize` method to call the `ConfigurationData` constructor to populate your new field. Change all the `ConfigurationUtils` properties to return the appropriate properties from your new field instead of the hard-coded values they currently return.
4. Add code to the `GameInitializer Awake` method to call the `ConfigurationUtils Initialize` method.
5. Run your game. It should work just like it did before you started working on this step in the assignment.
6. Create a csv (comma-separated value) file containing the configuration data; you can name it anything you want. The first line in the file should be a comma-separated list of the value names and the second line in the file should be a comma-separated list of the values. This is the format you might expect to see exported from a spreadsheet that game designers were using to tune the game. The `ConfigurationData` class only has two properties at this point, so you only need to include names and values for those two properties in the file. Don't worry, this file will have many more values before we're done with the game!
7. Create a new StreamingAssets folder (capitalize it exactly as shown) in your Project window and move your csv file into that folder. Putting the file there makes it automatically get included in our build and also makes it so we can use `Application.streamingAssetsPath` to get the file location from our script, which will work both in the editor and when you distribute your game.
8. Add code to the `ConfigurationData` constructor to open the configuration data file using `Application.streamingAssetsPath` and your file name as the full file name. Be sure to include this code in an exception handler. You should have a catch block that catches all exceptions (but doesn't do anything in the catch block) and you should have a finally block that closes the file if it's not null. You'll need to add a `System.IO` using directive to get access to the `StreamReader` class and a `System` using directive to get access to the `Exception` class.
9. Add code to extract the values from the second line in the file and populate the fields with them. I did this in a separate method, knowing that code will get fairly large by the time we're done with the game, but you can add the code to the constructor instead if you prefer that approach.

When you run your, game it should work just like it did before this step.

You should make sure this is working by changing values in your csv file and making sure those changes have the expected effect in the game (be sure to close the csv file before running your game). This should work fine in the editor and for standalone Windows and Mac builds, but WebGL builds don't seem to read from the csv file even though it's included in the build. Numerous people have posted this problem on the web, but from what I can tell no one has ever answered those posts. That's why we've made sure the `ConfigurationData` class works with default values if the file read fails.

That's it, you're finally done with this project!