Wacky Breakout Increment 2
Detailed Instructions

Overview

In this project (the second increment of our Wacky Breakout game development), you're adding functionality to the game.

To give you some help in approaching your work for this project, I've provided the steps I implemented when building my project solution. I've also provided my solution to the previous in the materials zip file, which you can use as a starting point for this project if you'd like. As you can see, the steps for this project are much less detailed than those for Wacky Breakout Increment 1; that's because you're maturing as a game programmer and should be able to figure out how to do more without detailed instructions.

Step 1: Add a death timer to the ball

For this step, you're destroying the ball after a set period of time.

Add a value in the configuration data CSV file for the ball lifetime (in seconds, I used 10) and add the appropriate fields and properties to the `ConfigurationData` and `ConfigurationUtils` class so the `Ball` class will be able to access that value.

Copy the Timer script from the zip file you downloaded into your Scripts folder.

Add a Timer component to the Ball and run the timer with the ball lifetime as the timer duration when the Ball is added to the scene.

Have the ball destroy itself when the timer is finished.

Step 2: Spawn a new ball when ball dies

For this step, you're spawning a new ball when a ball dies.

Create a new BallSpawner script in your Scripts folder and attach the script to the main camera. Add a public method that spawns a new ball in the game to the script (you'll need to create a Ball prefab and add a new field to the script for the Ball prefab as well).

Have the Ball script call the new method just before it destroys itself. You'll need to get access to the BallSpawner script to call a method on that script. That's not as hard as it sounds, though, because `Camera.main` gives you a reference to the main camera and because scripts are just components you can use the `GetComponent` method to get a reference to the script.

Step 3: Fix ball spawning unfairness

For this step, you're making balls wait for a second before they start moving. This actually gives the player a chance to "get set" at the start of the game, and also makes it more fair when a new ball is spawned into the scene.

There are a variety of reasonable ways to have the Ball script do this. I added another timer, and when the timer was finished I added the force to the ball to get it moving.

I actually had to add a `Stop` method to the Timer script because my Ball script was detecting that the move timer was finished every frame (after 1 second) and adding force to the ball every frame. That really got the ball moving!

Step 4: Spawn a new ball when a ball leaves the bottom of the screen

For this step, you're spawning a new ball when a ball leaves the bottom of the screen.

Have the Ball script call the `SpawnBall` (or whatever you called it) method in the BallSpawner script after it detects it became invisible (left the screen) and before it destroys itself (which it should do because it's no longer in the game). The logic is actually a little more complicated than that, because a ball that's destroying itself because its death timer expired also becomes invisible as it's removed from the scene. Be sure to include the appropriate logic in your `OnBecameInvisible` method to sure you don't spawn a ball in that method if the ball became invisible because the death timer finished.

When I run my game at this point, everything worked fine until I clicked the Play button in the Unity editor to stop the game. At that point, I got a Unity error that says "Some objects were not cleaned up when closing the scene. (Did you spawn new GameObjects from OnDestroy?)" This happens because as Unity shuts the game down, the ball becomes invisible, which tells the BallSpawner to spawn a new ball. This isn't really a problem in this game, but in a game where we were moving between scenes it would be.

To fix this, I added more logic to my `OnBecameInvisible` method to make sure the ball is actually below the bottom of the screen and only spawn a new ball if it is (we still need to spawn a new ball when the death timer finishes no matter where the ball is, but I already handle that case in the `Update` method).

Caution: Even if this is working properly, so a player could play the built game and see a new ball spawned immediately after a ball left the bottom of the screen, it may not seem to be working in the Unity Editor. The best thing to do is to actually build and play the game to test this functionality. 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 bottom and top of the camera view just disappears from view.

Step 5: Spawn a new ball every 5 to 10 seconds

For this step, in addition to replacing balls that expired or left the screen, you're spawning a new ball randomly every 5 to 10 seconds. The minimum and maximum spawn times need to be stored in and used from the configuration data file.

Add values in the configuration data CSV file for the min and max spawn seconds (in seconds, I used 5 and 10 for these) and add the appropriate fields and properties to the `ConfigurationData` and `ConfigurationUtils` class so the `BallSpawner` class will be able to access these values.

Add a Timer component to the BallSpawner and run the timer with a random duration between the min and max spawn seconds when the BallSpawner is added to the scene. I wrote a separate method to provide the spawn delay since I know I'll need that code every time I spawn a ball (to figure out how long to wait until spawning the next ball)

Add code to the `Update` method to spawn a new ball when the spawn timer finishes and restart the spawn timer with a new random time.

Step 6: Spawn into a collision-free location

For this step, you're making sure balls are spawned into a collision-free location. We need to do this now that we can have multiple balls in the game because we don't want to spawn one ball on top of another. The "big idea" behind the approach I used is that if I'd be spawning into a collision, I set a flag to say I need to retry the spawn and I try the spawn again on the next frame of the game if that flag is true.

You can check if there's a collision in a particular area of the game world by calling the `Physics2D` `OverlapArea` method with two diagonally opposite corners of the rectangle you want to check. Because balls always spawn in the same location, I declared fields for and saved the lower left and upper right corners of the ball collider at the spawn location in the `Start` method, then used those corners when I called the `OverlapArea` method.

You should of course feel free to try to figure this out on your own. If you get stuck (or tired!), though, I've provided some potentially useful chunks of code in the collision free spawning code.txt file in the zip file you downloaded.

Step 7: Add HUD with score and balls remaining

For this step, you're adding a HUD with the score and balls remaining. These values won't be updated yet, that will come in the following steps.

Add a new HUD game object to the scene and add balls left and score Text objects as children of that game object. Put reasonable default text for those Text objects. I placed the balls left text on the lower left and the score text on the lower right, but the placement is up to you.

Be sure to select the Canvas in the HUD game object and change the UI Scale Mode in the Canvas Scaler component to Scale With Screen Size. This makes it so everything is placed and sized reasonably no matter what resolution the player uses when they run the game.

Step 8: Add accurate scoring

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

Write a HUD script and attach it to the HUD game object. Add static fields to hold the score Text object and the actual score. Add a static public method that a block can call to add points to the score. In the body of that method update both the score and the score text that's displayed.

Add code to the `Start` method to populate the score Text field. To make this easier, I added a tag to the score Text object in the scene so I could just find the object by its tag in the `Start` method. This is a little trickier than you might think, though, because you need to actually get the Text component attached to the score Text game object.

Add a field to the Block script to hold how many points the block is worth. Make this field protected, because child classes will need to access this field to set it.

Add code to the Block script to call the new HUD method just before the block is destroyed. This isn't an optimal object-oriented solution because Block objects shouldn't really have to know about the existence of the HUD or the methods it exposes, but it's a reasonable solution given the C# knowledge we have at this point. Don't worry, we'll make this much better once we know about delegates and event handling.

Add a value in the configuration data CSV file for the Standard block points and add the appropriate fields and properties to the `ConfigurationData` and `ConfigurationUtils` class so the `StandardBlock` class (see next paragraph) will be able to access this value.

Create a new StandardBlock script as a child class of the `Block` class. Add fields for the standard block sprites, marking each field with SerializeField so you can populate it in the Inspector. Add code to the StandardBlock `Start` method to set the points the block is worth to the Standard block value. Also add code to this method to randomly select one of the standard block sprites for this block.

Remove the Block script from the StandardBlock prefab and add the StandardBlock script instead. Populate the three sprite fields in the Inspector.

Step 9: Automatically build 3 rows of standard blocks

For this step, you're adding automatic block building to the game. We'll actually implement this so we add the paddle to the scene as well, so this step will actually add all the required gameplay objects to the scene.

Turn the Paddle into a prefab and remove it from the scene.

Create a new LevelBuilder script and attach it to the main camera. Add a field to the script for the Paddle prefab, add code to the `Start` method to instantiate the prefab in the scene, and populate the field in the Inspector.

Remove the Ball game object from the scene and change the BallSpawner script to immediately spawn a new ball when it's added to the scene.

Remove all the StandardBlock game objects from the scene and add a StandardBlock prefab as a field in your script. Populate the field in the Inspector.

Add code to build the three rows of standard blocks. The rows should start around 1/5 of the screen height down from the top of the screen and the rows should be centered horizontally. You'll need to create a temporary block to retrieve the block width and height; don't forget to destroy that block when you've done that! After that, it's just some math with the screen dimensions and block size and a couple of nested for loops to instantiate all the blocks. Go ahead and draw a picture if it will help you figure out the math.

Step 10: Add Bonus, Freezer, and Speedup blocks

For this step, you're adding the other 3 kinds of blocks.

Add a value in the configuration data CSV file for the Bonus block points and add the appropriate fields and properties to the `ConfigurationData` and `ConfigurationUtils` class so the `BonusBlock` class (see next paragraph) will be able to access this value.

Add a sprite for the Bonus block and create a new BonusBlock script as a child class of the `Block` class. Add code to the `BonusBlock` `Start` method to set the points the block is worth to the Bonus block value.

Make a BonusBlock prefab using the Bonus block sprite and the BonusBlock script. Add a field for the BonusBlock prefab to the LevelBuilder script and populate that field in the Inspector. Change the LevelBuilder script to only use Bonus blocks so you can make sure the scoring works properly.

Add a value in the configuration data CSV file for the Pickup block points and add the appropriate fields and properties to the `ConfigurationData` and `ConfigurationUtils` class so the `PickupBlock` class (see next paragraph) will be able to access this value.

Add sprites for the Freezer and Speedup blocks and create a new PickupBlock script as a child class of the `Block` class. Add code to the `PickupBlock` `Start` method to set the points the block is worth to the Pickup block value. Add fields to the script to store the two pickup sprites and populate them in the Inspector.

Make a PickupBlock prefab using the Freezer block sprite and the PickupBlock script. Add a field for the PickupBlock prefab to the LevelBuilder script and populate that field in the Inspector.

Copy the PickupEffect.cs file from the zip file into your Scripts\Gameplay folder. That file defines a PickupEffect enum with two values: Freezer and Speedup. Add a field to the PickupBlock script to hold which kind of pickup effect the block is and add a public property for setting the field. Set the sprite for the block appropriately within the set accessor for the property.

Change the LevelBuilder script to only use Freezer blocks so you can make sure the scoring and sprite setting works properly. Do the same for Speedup blocks. Hint: You'll need to save the block game object you instantiate so you can get the PickupBlock component attached to that game object so you can access the property to set the effect for the block.

Note: You do NOT have to actually implement the freezer and speedup effects; that happens in the next project. The points should be correct when you hit a pickup block, though.

Step 11: Randomizing the blocks

For this step, you're randomly adding the four types of blocks to the scene based on probabilities.

Add values in the configuration data CSV file for the different block probabilities and add the appropriate fields and properties to the `ConfigurationData` and `ConfigurationUtils` class so the `LevelBuilder` class will be able to access those values.

Add code to the LevelBuilder script to randomly decide which block to place each time a block is added to the level based on the probabilities for each block. My `Start` method was getting somewhat large at this point, so I added a `PlaceBlock` method that randomly picks a block to place at a given location.

Step 12: Add accurate balls left

For this step, you're making the balls left display in the HUD accurate.

Add a value in the configuration data CSV file for the number of balls per game and add the appropriate fields and properties to the `ConfigurationData` and `ConfigurationUtils` class so the `HUD` class will be able to access this value.

In the HUD script, add static fields to hold the balls left Text object and the actual balls left. Add a static public method that a ball can call to tell the HUD it left the bottom of the screen. In the body of that method update both the balls left and the balls left text that's displayed.

Add code to the `Start` method to populate the balls left value and Text fields. The balls left value starts as the number of balls per game. To make the Text part easier, I added a tag to the balls left Text object in the scene so I could just find the object by its tag in the `Start` method. This is a little trickier than you might think, though, because you need to actually get the Text component attached to the balls left Text game object.

Add code to the `Ball OnBecameInvisible` method to call the new HUD method as appropriate. Don't worry that the balls left can go negative; we won't see that when we add functionality to end the game when that gets to 0.

That’s it for this project.