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: Hit the block

For this step, you're adding a block 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. Add a block about 1/3 of the way down from the top of the screen directly above the ball and paddle.

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

Step 2: Break the block

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

1. Add a Block script to the Scripts/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 add a Freeze Rotation Constraint on the Z axis

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

Step 3: 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 and ball 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.

Turn the Ball into a prefab and remove it from the scene. Create a new BallSpawner script that immediately spawns a new ball when it's added to the scene (you'll need to add a field for the Ball prefab and populate it in the Inspector). Attach the BallSpawner script to the main camera.

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 StandardBlock game object from the scene and add a StandardBlock prefab as a field in your LevelBuilder 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.

When you run your game, you should have 3 rows of blocks (with no spaces between the blocks) and you should be able to destroy the blocks by hitting them with the ball.

Note: At this point, all the blocks will be using the same sprite. Don't worry, we'll change that in a later increment!

Step 4: Add a death timer to the ball

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

Add a property to the `ConfigurationUtils` class for the ball lifetime (in seconds, I used 10), making the property `public` so the `Ball` class will be able to access that value.

Copy the Timer script from the zip file you downloaded into your Scripts/Gameplay 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 5: Spawn a new ball when ball dies

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

Add a `public SpawnBall` method to the BallSpawner script that spawns a new ball in the game to the script.

Change the `BallSpawner Start` method to call the `SpawnBall` method when it's added to the scene if you added that functionality directly into the `Start` method in Step 3.

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 6: 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 7: 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` 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 8: 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. Add properties to the `ConfigurationUtils` class for minimum and maximum spawn times. I used 5 seconds for the min spawn seconds and 10 seconds for the max spawn seconds. Make the properties `public` 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.

Note: It's possible for a collision between the balls to stop one or more of the balls. Don't worry about that; the stopped ball(s) will eventually die out of the game. You could of course write additional code to make sure each ball's speed stayed constant, but we won't bother doing that in this game.

Step 9: 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.

You're done with this increment.