Wacky Breakout Increment 4
Detailed Instructions

Overview

In this project (the final increment of our Wacky Breakout game development), you're adding the remaining 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 project in the materials zip file, which you can use as a starting point for this project if you'd like.

Step 1: Refactor HUD AddPoints method

For this step, you're refactoring the `HUD` `AddPoints` method to use an event invoker and an event listener instead.

Define a new event for a points added event as a UnityEvent with one int argument.

The easiest way to find out what classes should be invokers for the event is to find out who calls the `HUD` `AddPoints` method. Right click the method name and select Find All References in Visual Studio. Add the appropriate field, field initialization, and an `AddPointsAddedListener` method to the class you find (which should be the `Block` class).

Caution: One reasonable place to initialize the field is in the `Start` method. Because the `StandardBlock`, `BonusBlock`, and `PickupBlock` child classes implement `Start` methods to set their points properly, the `Block` `Start` method never gets called! Add `virtual protected` in front of the `Block` `Start` method and add `override protected` in front of the 3 child class `Start` methods. Add code to the 3 child class `Start` methods to call the parent class `Start` method using the `base` keyword.

Add support to the EventManager for lists of listeners and invokers for the new event.

Go back to the `Block` class and add code to add it to the EventManager as an invoker of the event. Change the code that calls the `HUD` `AddPoints` method to invoke the event instead.

Remove the `public static` keywords in front of the `HUD` `AddPoints` method. Add code to add that method to the EventManager as a listener for the event.

When you run the code it should run just like it did before. This is a classic example of refactoring, where we change the structure of our code to improve it without changing the game functionality. With this change, the `Block` class doesn't have to know about the `HUD` class and its methods any more, which is a better object-oriented design.

Step 2: Refactor HUD ReduceBallsLeft method

For this step, you're refactoring the `HUD ReduceBallsLeft` method to use an event invoker and an event listener instead.

Follow a similar approach to the one you used in Step 1 to accomplish this refactoring.

Step 3: Refactor Timer to invoke a TimerFinished event

For this step, you're refactoring the `Timer` to invoke an event when the timer finishes. Change all the code that accesses the `Finished` property to use a listener for the event instead. I didn't use the EventManager to hook the invoker and listener together because both the timer and the method to handle the timer finished event were always in the same class in my implementation.

Note: There's actually one place in my code where I check if a timer is not finished; accessing the property for that check is the appropriate thing to do.

Step 4: Change BallSpawner SpawnBall method to private

For this step, you're making it so the `Ball` class never calls the `BallSpawner SpawnBall` method directly. You can take care of one of those calls by making the `SpawnBall` method a listener for the event you implemented in Step 2 above. You'll need a new event for when a ball dies, though, because the HUD is listening for the event from Step 2 but we don't reduce the number of balls left when a ball dies, we only do that when a ball is lost.

Step 5: Create game over message prefab

For this step, you're building the game over message prefab.

Create a prefab for the game over message and put it in the Resources folder. Add a script that handles pausing the game and unpausing the game, destroying itself, and going to the main menu when the Quit button is clicked. The script also needs to expose a way to set the score that's displayed.

This is similar to, but simpler than, the PauseMenu prefab.

Step 6: Show game over message when last ball is lost

For this step, you're showing the game over message when the player loses the last ball in the game.

I added a LastBallLost event, used the HUD as the invoker for that event (since the HUD keeps track of how many balls are left in the game), listened for the event in the WackyBreakout script, and had the WackyBreakout script instantiate the game over message and set the score in the

message when the event was invoked. I added a `Score` property to the HUD and tagged the HUD to make it easier for the WackyBreakout script to find so it could access the `Score` property.

Step 7: Show game over message when last block is destroyed

For this step, you're showing the game over message when the player destroys the last block in the level.

I added a BlockDestroyed event, used the Block as the invoker for that event (since the block knows when it's being destroyed), listened for the event in the WackyBreakout script, and had the WackyBreakout script instantiate the game over message and set the score in the message when the last block in the level was destroyed. I tagged the block prefabs so the WackyBreakout script could retrieve the tagged objects and see if the array had 1 block in it (the block about to be destroyed).

Step 8: Add sound effects

I used the approach from the Feed the Teddies game to implement the required sound effects in the game. Make sure you put all your audio clips in the Resources folder so the AudioManager can load them.

You should find the Sound lecture (and the code accompanying that lecture) from the More C# Programming and Unity course (the previous course in the specialization) useful as a reference as you complete this step.

That's it for this project -- and for the game!