Wacky Breakout Increment 5
Detailed Instructions

Overview

In this project (the fifth increment of our Wacky Breakout game development), you're refactoring the game to use event handling, leading to a much better object-oriented design. You're also adding the freezer and speedup effects 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 ball death processing

For this step, you're refactoring how we handle a ball being destroyed because its death timer has finished to use an event invoker and an event listener instead.

Create a new Events folder in the Scripts folder in the Project window. Create a new EventManager script in your new folder. Open the EventManager script in Visual Studio and change it to a static class that doesn't inherit from `MonoBehaviour`. Add a documentation comment for the class and add a using directive for the `UnityEngine.Events` namespace. Delete the `Start` and `Update` methods from the class.

Create a BallDiedEvent script in the Scripts/Events folder in the Project window and make it a UnityEvent with no parameters.

We know the `Ball` class is the class that will invoke this event, so add the appropriate field to the `Ball` class, making sure you assign a new `BallDiedEvent` object to the field to initialize it. Add an `AddBallDiedListener` method to the class that adds a listener to the event.

Add support to the EventManager for lists of listeners and invokers for the new event. In addition to the fields you need to add, be sure to include the following methods: `AddBallDiedInvoker`, `AddBallDiedListener`, and `RemoveBallDiedInvoker`.

We want to make sure we remove balls as invokers of this event when they're destroyed so we don't have extraneous invokers in our list of invokers; that's why we're including the `RemoveBallDiedInvoker` method.

Go back to the `Ball` class and add code to add it to the EventManager as an invoker of the event. Change the code that calls the `BallSpawner SpawnBall` method to invoke the event instead.

Also in the `Ball` class, add code to remove it from the EventManager as an invoker of the event when the ball is destroyed. Since we're now doing multiple things whenever we destroy a ball, and we destroy a ball in multiple places, I pulled that logic out into a private method.

Add code to the `BallSpawner` to add the `SpawnBall` method to the EventManager as a listener for the BallDiedEvent.

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 `Ball` class doesn't have to know about the `BallSpawner` class and its `SpawnBall` method when a ball dies any more, which is a better object-oriented design.

Step 2: Refactor ball lost processing

For this step, you're making it so the `Ball` class never calls the `BallSpawner SpawnBall` method directly.

Follow a similar process to the one you followed in the previous step to refactor the ball lost processing in the game. Don't use the BallDiedEvent for this processing, create a new BallLostEvent and add the appropriate support for this new event in the `EventManager`, `Ball`, and `BallSpawner` classes. Run your code to make sure everything a new ball is spawned when the player loses a ball.

You might be wondering why we're using two different events that both result in the ball spawner spawning a new ball. We're doing that because now we need to refactor the HUD functionality that tells the player how many balls they have left. Because we won't reduce that count when a ball dies but we will reduce it when a ball is lost, we need two different events. The BallSpawner needs to listen for both those events, but our HUD will only listen for the BallLostEvent.

Refactor the `HUD` to listen for the BallLostEvent and remove the call to the `HUD LoseBall` method from the `Ball` class.

Run the game to make sure everything works properly when you lose a ball.

Step 3: Make BallSpawner SpawnBall and HUD LoseBall methods private

Now that no other objects call the `BallSpawner SpawnBall` or `HUD LoseBall` methods directly, we should make those methods private. You should also remove `static` from the `HUD LoseBall` method. Do that now.

Step 4: 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 the `Start` method 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. Also, add code to remove the block from the EventMaanager as an invoker of the event just before you destroy the block.

Make the `HUD AddPoints` method private instead of public and remove `static` from that method. Add code to the `HUD Start` method to add the `AddPoints` 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 5: 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 checks if the `Finished` property is true 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.

Using an event for when the timer finishes should feel much more intuitive than the way we’ve been using timers up to this point. In real life, you don’t start a timer and then look at it on every “frame” to check if it’s finished yet (in Unity, checking the `Finished` property in an `Update` method). Instead, you start a timer and then you listen for the timer to notify you that it’s finished (in Unity, when the event is invoked it calls all the methods that have been added as listeners).

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 6: Add the freezer effect

For this step, you're implementing the freezer effect.

Add a value in the configuration data CSV file for the freezer effect duration (in seconds, I used 2) and add the appropriate fields and properties to the `ConfigurationData` and `ConfigurationUtils` class so the `EffectBlock` class will be able to access that value.

Add a field to the `EffectBlock` class to hold the effect duration. Add code to the `Effect` property to set the duration of the freezer effect.

In your Scripts/Events folder, create a new event called FreezerEffectActivatedEvent that has one float parameter (for the duration of the effect). Your event should be a child class of the appropriate form (one float argument) of UnityEvent. Remember that UnityEvent is in the UnityEngine.Events namespace.

Add a field to the `EffectBlock` class to hold a `FreezerEffectActivatedEvent` object. Add code to the `Effect` property to instantiate a new object in that field. Add a new public `AddFreezerEffectListener` method to let consumers of the class add a listener for the FreezerEffectActivatedEvent; remember that UnityAction is in the UnityEngine.Events namespace.

Add the appropriate fields and methods to the EventManager to add invokers and listeners and remove invokers for the FreezerEffectActivatedEvent.

Add code to the `EffectBlock Effect` property to add the script to the event manager as a freezer effect activated invoker.

Now we need to have the EffectBlock script invoke the event when the ball collides with the block. Because we need specialized behavior in this script, add the keywords `virtual protected` before the `OnCollisionEnter2D` method in the `Block` class. Override that method in the `EffectBlock` class, invoking the FreezerEffectActivated event (with the duration of the freezer effect as the argument) and removing the script from the EventManager as an invoker of the FreezerEffectActivatedEvent if this is a freezer block. You should also call the `OnCollisionEnter2D` method in the parent class (using the `base` keyword) to handle the scoring and block destruction.

Add fields to the `Paddle` class to keep track of whether or not the paddle is frozen and to hold a timer for when the paddle is frozen.

Write a method in the `Paddle` class to handle when the FreezerEffectActivatedEvent is invoked. You'll need to freeze the paddle at that point, of course, but you'll also have to either start the

freeze timer or add time to it if it's already running. You'll have to make the appropriate changes to the Timer script to support that.

Add code to the `Paddle Start` method to add a Timer component and to add your new method to the event manager as a listener for the FreezerEffectActivatedEvent.

Change the `Paddle FixedUpdate` method to only move the paddle if the paddle isn't frozen.

Add a method to the `Paddle` class to handle when the freeze timer finishes; this method should unfreeze the paddle and stop the freeze timer. Add this method as a listener for the TimerFinishedEvent for the freeze timer in the `Paddle Start` method.

To test my code, I changed the LevelBuilder script to only add Freezer blocks.

Step 7: Add the speedup effect

Add the speedup effect to the game using the ideas you used in the previous step. I used the two parameter forms of UnityEvent and UnityAction here so the invoker could pass the effect duration and speedup factor to the listeners.

In this case, it's certainly reasonable to add the speeding up when the effect is activated and returning to normal speed when the effect finishes functionality to the `Ball` class, much like we added the freezer effect starting and stopping to the `Paddle` class. That's the approach I took in my solution.

Remember, if a Speedup block is destroyed while a speedup is already in effect, the effect duration should be added to the duration of the effect but the speed of the balls shouldn’t be increased.

Don't forget to slow the balls down again when the effect is finished.

Finally, this is the first time we’ve had a listener for an event destroyed during gameplay. In addition to the other methods you added to the `EventManager` class to provide speedup effect support, add a `RemoveSpeedupEffectActivatedListener` method to the `EventManager` class. You’ll also need to add a `RemoveSpeedupEffectActivatedListener` method to the `EffectBlock` class for the `EventManager` method to call. Before destroying the ball (either because the death timer finished or the ball left the screen), call the `RemoveSpeedupEffectActivatedListener` method to remove it as a listener for that event.

To test my code, I changed the LevelBuilder script to only add Speedup blocks.

Step 8: Add speedup effect for newly-spawned balls

Even though balls that are currently active will speed up when the speedup effect is activated, balls that are spawned while the effect is active will move at the original speed. This is actually

an interesting problem to solve, because it means we need some class to keep track of the effect's status for the whole game.

Add a SpeedupEffectMonitor script to the Scripts/Gameplay folder in the Project window and attach the script to the main camera. Add the required code so this class listens for the SpeedupEffectActivatedEvent and uses a timer to keep track of whether or not the speedup event is currently active. The class should also expose properties that tell whether or not the speedup effect is currently active, what the speedup factor is, and how much time is left in the speedup effect. The last property will require a change to the Timer script to get how much time is left on a timer.

For a good object-oriented design, we don't want other classes in our solution to know about the main camera and the scripts attached to it, so add an EffectUtils script in the Scripts/Util folder in the Project window. This should be a public static class that provides static properties that simply wrap the properties exposed by the `SpeedupEffectMonitor` class. Other classes in our solution can then directly access this utility class to get that information just as they access other utility classes like `ConfigurationUtils` and `ScreenUtils`.

Next, a ball that's going to start moving needs to know about the speedup effect so it can start moving at the appropriate speed. Change the `StartMoving` method in the `Ball` class to use the properties in the `EffectUtils` class to start moving at the correct speed (and set the speedup fields as appropriate).

CAUTION: You can't just multiply the velocity by the speedup factor here because the physics engine hasn't actually started moving the ball yet. Change the force you apply to the ball as necessary to make it start moving at the correct speed.

You're done with this increment.