Wacky Breakout Increment 3
Detailed Instructions

Overview

In this project (the third 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 project in the materials zip file, which you can use as a starting point for this project if you'd like.

Step 1: 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 `PickupBlock` class will be able to access that value.

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

Create a new Events folder in the Scripts folder in the Project window. In your new folder, create a new event called FreezerEffectActivated 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 `PickupBlock` class to hold a FreezerEffectActivated event 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 FreezerEffectActivated event; remember that UnityAction is in the UnityEngine.Events namespace.

Create a new EventManager script in the Events folder in the Project window. Add the appropriate fields and methods to the class (remember, this should be a static utility class) to add invokers and listeners for the FreezerEffectActivated event. This is very similar to the event manager we implemented in the Refactoring Teddy Bear Destruction lecture.

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

Now we need to have the PickupBlock 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 `PickupBlock` class, invoking the FreezerEffectActivated event (with the duration of the freezer effect as the argument) if this is a freezer block and calling the method in the parent class 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 FreezerEffectActivated event 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 FreezerEffectActivated event.

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

Add code to the `Paddle Update` method to unfreeze the paddle and stop the freeze timer once the freeze timer is finished.

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

Step 2: Add the speedup effect

Add the speedup effect to game using the ideas you used in Step 1. 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.

Step 3: 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 SpeedupEffectActivated event 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/Utils 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.

Step 4: Starting on the menu system

Add a `MenuName` enumeration to list all the menu names in the game and add a public static `MenuManager` class with a public static `GoToMenu` method. Add a switch statement to that method with cases for each of the menu names and breaks in each of those cases to make sure you can compile.

NOTE: There's a lecture called Menus describing a menu system very similar to this one for a different game in the next module of the course. The ideas are the same as for your menu system here, so feel free to use the lecture (and the lecture code) if you get stuck building your menu system for this project. In the book (which I provided free to you in a Week 1 reading), developing a menu system very similar to this one is covered in Sections 20.2 through 20.5.

Step 5: Add a quit button to the main menu

Add a new scene called MainMenu to your game and rename scene0 to Gameplay instead. Don't forget to add new scenes you add to the game to the build using File > Build Settings so you can navigate between the scenes.

Add a quit button that exits the game to the main menu (in the MainMenu scene you just added). The Adding a Simple Menu System lecture shows how to do this. All menu buttons in your game, including this one, should swap sprites when they highlight and unhighlight.

Make sure you modify the UI Scale Mode value of the Canvas Scaler component of your Canvas game object to Scale With Screen Size. I also set my Reference Resolution in that component to 1280 by 720.

Build the game to test this functionality (quitting the application only works in a built game, not in the Unity editor).

Step 6: Add a help button and a help menu to the game

Add a help button that goes to a help menu from the main menu. Add a help menu that's a single page that displays brief game instructions. Include a Back button on the help menu that returns the player to the main menu.

Step 7: Add a play button to the game

Add a play button that goes to the Gameplay scene from the main menu.

Step 8: Add a pause menu to the game

Add a pause menu that pauses the game and displays resume and quit buttons when the player presses the Escape key during gameplay. Clicking the resume button resumes gameplay and clicking the quit button returns the player to the main menu. The Adding a Menu Manager lecture shows how I added a pause menu to the Fish Revenge game.

Remember, the pause menu needs to be a prefab in a Resources folder in the Project window for the approach I demonstrate in the lecture to work properly. Make sure you add the PauseMenu script to the game object you're turning into a prefab, not to the main camera.

That’s it for this project.