Wacky Breakout Increment 6
Detailed Instructions

Overview

In this project (the final increment of our Wacky Breakout game development), you're adding the remaining functionality to the game; specifically, you're adding the menus and sound 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: Starting on the menu system

Add a Scripts/Menus folder in the Project window.

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 the Feed the Teddies game. 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 increment. 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 2: Add a quit button to the main menu

Change the name of the Scene0 scene to Gameplay.

Add a new scene called MainMenu to your game. When you add new scenes to the game you also need to add those scenes to the build using File > Build Settings so you can navigate between the scenes. The order of the scenes matters, because when the game starts running it will start on the scene at the top of the list. Make sure you add both the MainMenu and Gameplay scenes to the Scenes in Build, with the MainMenu scene first.

Remove the GameInitializer script from the main camera in the Gameplay scene and attach that script to the main camera in the new MainMenu scene. Because the main menu scene will be the starting scene for the game, we want to initialize the game in that scene.

Add a Canvas to the MainMenu scene and change the name of the Canvas to Main Menu. To add a Canvas, right click in the Hierarchy window and select UI > Canvas.

Add a quit button that exits the game to the main menu canvas. The Adding a Simple Menu System video in the previous module shows how to do this. All menu buttons in your game, including this one, should swap sprites when they highlight and unhighlight.

Note: In Unity 2021.1.26f1, I got an error message when I clicked the Sprite Editor for my quit menu button sprite; the error message told me I needed to install the 2D Sprite package. To do that, I selected Window > Package Manager from the top menu bar, clicked the Packages dropdown near the top left of the Package Manager window, selected Unity Registry, then selected 2D Sprite from the options on the left and installed that package.

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 1920 by 1080.

Create a MainMenu script in the Scripts\Menus folder in the Project window and attach that script to the canvas for your main menu. Add an `ExitGame` method that quits the application to the `MainMenu` class and add that method as a listener to the Button component of the quit button.

Select Edit > Project Settings… from the top menu bar and select Player on the left. On the right, expand the Resolution and Presentation section. Click the dropdown next to Fullscreen Mode and select Windowed. Set the Default Screen Width and Default Screen Height to reasonable values (I used 1920 and 1080 for those). Close the Project Settings window.

The `Application Quit` method doesn't work in the editor; you'll have to build for PC, Mac & Linux Standalone to test this functionality. Build and run the game to make sure the quit button exits the game.

Step 3: 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.

Hint: Add code to the switch statement in the `MenuManager GoToMenu` method to help you move between the menu scenes.

If you run into trouble this, remember – the debugger is your friend!

Step 4: Add difficulty differences to configuration data

Add values in the configuration data CSV file for easy, medium, and hard ball impulse forces and easy, medium, and hard min and max spawn seconds. Add the appropriate fields and properties for those values to the `ConfigurationData` class.

For now, change the `ConfigurationUtils` impulse force and min and max spawn seconds properties to return those properties for the easy difficulty properties from the `ConfigurationData` class.

Step 5: Add a difficulty menu scene

Add a new scene to your Unity project for the Difficulty Menu. Add Easy, Medium, and Hard buttons that highlight when you hover the mouse over them to the menu (they won't do anything when you click them yet).

Add the new scene to the Scenes In Build in the Build Settings.

Add a play button to the main menu that will go to the difficulty menu when clicked. Add code to the `MenuManager` `GoToMenu` method to go to the difficulty menu scene. Add a `MainMenu` `GoToDifficultyMenu` method that calls the `MenuManager` `GoToMenu` method with the appropriate argument. Add the `GoToDifficultyMenu` method as a listener for the On Click event for the play menu button on the main menu.

When you run your game, clicking the play button on the main menu should bring you to the difficulty menu. All the difficulty menu buttons should highlight when you hover over them, but they won't do anything yet when you click them.

Step 6: Start easy game

Add a `Difficulty` enum with `Easy`, `Medium`, and `Hard` values to the Scripts folder in the Project window.

Go to Section 20.4 of the book to the place where we're adding the `GameStartedEvent` (around 88% complete in the Kindle book or page 435 in the pdf). Add a `GameStartedEvent` (with a `Difficulty` parameter rather than an `int` parameter) to the project.

Add a DifficultyMenu script to the Scripts/Menus folder and attach the script to the canvas for the difficulty menu.

Add support for adding invokers and listeners for the `GameStartedEvent` to the `EventManager` in the usual way. The `DifficultyMenu` class will be the invoker for the event, so you'll need to add a field for a game started event and an `AddGameStartedListener` method to that class. At this point, everything should compile.

Next, in the `DifficultyMenu` `Start` method, add the difficulty menu as a game started event invoker in the event manager.

Based on our work with menus, you've probably realized that we need separate methods to respond to clicks on each of the menu buttons. Add a `StartEasyGame` method that invokes the game started event with easy difficulty. Go to the Unity editor and add this method as a listener for the On Click event for the easy menu button.

Great, now the event will be invoked at the appropriate time, but nobody is listening for it. Go to Section 20.4 of the book to the place where we're adding the `DifficultyUtils` class (around 89% complete in the Kindle book or page 437 in the pdf). Add the `DifficultyUtils` class to the project. You'll have to modify the code in the book a little bit to get it to work in your project, but if you understand how the class in the book works you should be able to make those minor modifications easily. Don't forget to call the `DifficultyUtils Initialize` method from the `GameInitializer Awake` method; be sure you do that after the call to the `ConfigurationUtils Initialize` method.

Run the game to make sure an easy game starts and works properly.

Step 7: Start medium and hard games

You probably noticed that an easy game starts properly, using the appropriate ball speed and spawn rates for an easy game, but this is "coincidental correctness". It happens because the `ConfigurationUtils` properties for those values return the easy values, not because you've implemented the difficulty system.

To see this, add `DifficultyMenu StartMediumGame` and `StartHardGame` methods to the `DifficultyMenu` class and add those methods as listeners for the On Click events for the Medium and Hard menu buttons on the difficulty menu. You can now click the Medium or Hard button to start a game, but it still starts an easy game.

The best way to proceed may seem a little complicated, but it keeps it so all the objects in the game continue always using the `ConfigurationUtils` class instead of having to know which values are difficulty-dependent.

Start by adding public static properties to the `DifficultyUtils` class for consumers to get `BallImpulseForce`, `MinSpawnDelay`, and `MaxSpawnDelay`. For now, just return 0 from the get accessor for each of those properties.

In the `ConfigurationUtils` class, leave the existing `BallImpulseForce`, `MinSpawnDelay`, and `MaxSpawnDelay` properties as they are. Add properties for `EasyBallImpulseForce`, `MediumBallImpulseForce`, `HardBallImpulseForce`, `EasyMinSpawnDelay`, `MediumMinSpawnDelay`, `HardMinSpawnDelay`, `EasyMaxSpawnDelay`, `MediumMaxSpawnDelay`, and `HardMaxSpawnDelay`, returning the appropriate `ConfigurationData` property from each of those new properties. These properties will only be accessed by `DifficultyUtils`, not by any other object or class.

Go back to the `DifficultyUtils BallImpulseForce`, `MinSpawnDelay`, and `MaxSpawnDelay` properties and add if or switch statements to the get accessors for those properties so they return the appropriate easy, medium, or hard `ConfigurationUtils` property based on the current values of the `DifficultyUtils difficulty` field.

One more thing to do and you'll be done. Go to the `ConfigurationUtils` class and change the `BallImpulseForce`, `MinSpawnDelay`, and `MaxSpawnDelay` properties to return the corresponding properties from the `DifficultyUtils` class (instead of the easy properties from the `configurationData` field).

Run the game to make sure all 3 difficulties work properly.

I know this seems like a lot of work to do, but it keeps all the other classes in our game working properly without any change at all, and that's a big win.

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.

Caution 1: The pause menu won't respond to button clicks unless there's an Event System in the scene. We should only have one Event System in a scene, and the Gameplay scene already has an Event System from when we added the HUD canvas, but the pause menu doesn't respond properly if we try to use that Event System. You should include an Event System in the Pause Menu prefab and delete the Event System from the Gameplay scene.

Caution 2: If you implement the pausing functionality in a `GameplayManager` class like I did, don't forget to attach that script to the main camera in the Gameplay scene!

Step 9: 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.

Caution: Make sure you include an Event System in the Game Over Message prefab.

Step 10: 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 LastBallLostEvent, 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 my GameplayManager script, and had the GameplayManager 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 GameplayManager script to find so it could access the `Score` property.

Step 11: 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 BlockDestroyedEvent, used the Block as the invoker for that event (since the block knows when it's being destroyed), listened for the event in the GameplayManager script, and had the GameplayManager 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 GameplayManager script could retrieve the tagged objects and see if the array had 1 block in it (the block about to be destroyed) to decide when to display the game over message.

Step 12: 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 Adding an Audio Manager lecture (and the code accompanying that lecture) from the C# Class Development course (the previous course in the Specialization) useful as a reference as you complete this step.

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