Wacky Breakout Increment 3
Detailed Instructions

Overview

In this Unity project (the first increment of our Wacky Breakout game development), you're building the Unity project and adding basic gameplay to the game.

Step 1: Add configuration data file processing

For this step, you're adding a `ConfigurationData` class, having that class read from a configuration data file and use the values from that file for its properties, and changing the `ConfigurationUtils` class to use the `ConfigurationData` class.

1. Copy the ConfigurationData script I provided in the zip file into your Scripts/Configuration folder. The `ConfigurationData` class acts as a container for all the configuration data. That data may come from a configuration data file or, if reading that file fails, from default values hard-coded into the class. The good news is that none of the other classes have to care where the data came from, they just use it.
2. Open the ConfigurationUtils script in Visual Studio and add a static field to hold a `ConfigurationData` object.
3. Add code to the `ConfigurationUtils Initialize` method to call the `ConfigurationData` constructor to populate your new field. Change all the `ConfigurationUtils` properties to return the appropriate properties from your new field instead of the hard-coded values they currently return.
4. Add code to the `GameInitializer Awake` method to call the `ConfigurationUtils Initialize` method.
5. Run your game. It should work just like it did before you started working on this step in the assignment.
6. Create a new StreamingAssets folder (capitalize it exactly as shown) in the Assets folder in your Project window. Putting our csv file (which we'll create in the following step) there makes it automatically get included in our build and also makes it so we can use `Application.streamingAssetsPath` to get the file location from our script, which will work both in the editor and when you distribute your game.
7. Create a csv (comma-separated value) file containing the five pieces of configuration data that are currently exposed by the `ConfigurationData` class; you can name the file anything you want, but be sure to put it in your StreamingAssets folder. The first line in the file should be a comma-separated list of the value names and the second line in the file should be a comma-separated list of the values. This is the format you might expect to see exported from a spreadsheet that game designers were using to tune the game. The `ConfigurationData` class only has five properties at this point, so you only need to include names and values for those five properties in the file. Don't worry, this file will have many more values before we're done with the game!
8. Add code to the `ConfigurationData` constructor to open the configuration data file using `Application.streamingAssetsPath` and your file name as the full file name. Be

sure to include this code in an exception handler. You should have a catch block that catches all exceptions (but doesn't do anything in the catch block) and you should have a finally block that closes the file if it's not null. You'll need to add a `System.IO` using directive to get access to the `StreamReader` class and a `System` using directive to get access to the `Exception` class.

9. Add code to extract the values from the second line in the file and populate the fields with them. I did this in a separate method, knowing that code will get fairly large by the time we're done with the game, but you can add the code to the constructor instead if you prefer that approach.

When you run your, game it should work just like it did before this step.

You should make sure this is working by changing values in your csv file and making sure those changes have the expected effect in the game (be sure to close the csv file before running your game). This should work fine in the editor and for standalone Windows and Mac builds, but WebGL builds don't seem to read from the csv file even though it's included in the build. Numerous people have posted this problem on the web, but from what I can tell no one has ever answered those posts. That's why we've made sure the `ConfigurationData` class works with default values if the file read fails.

Step 2: Add HUD with score and balls remaining

For this step, you're adding a HUD with the score and balls remaining. These values won't be updated yet, that will come in the following steps.

Add a new Canvas to the scene and rename it to HUD. Add balls left and score Text objects as children of that game object. Put reasonable default text for those Text objects. I placed the balls left text on the lower left and the score text on the lower right, but the placement is up to you.

Be sure to select the Canvas in the HUD game object and change the UI Scale Mode in the Canvas Scaler component to Scale With Screen Size. This makes it so everything is placed and sized reasonably no matter what resolution the game runs at. You also have to provide a Reference Resolution; I used 1920 by 1080 for that.

Step 3: Add accurate balls remaining

For this step, you're making the balls remaining display in the HUD accurate.

Add a value in the configuration data CSV file for the number of balls per game and add the appropriate fields and properties to the `ConfigurationData` and `ConfigurationUtils` class so the `HUD` class will be able to access this value.

Create a HUD script and attach it as a component to the HUD canvas. In the HUD script, add a field (marked with [SerializeField]) to hold the balls left Text object (as a `GameObject`). Populate the field in the Inspector.

Add static fields to hold the balls left `Text` object and the actual balls left. Add a static public method that a ball can call to tell the HUD it was lost (left the bottom of the screen). In the body of that method, update both the balls left and the balls left text that's displayed.

Add code to the `Start` method to populate the balls left value and Text field. The balls left value starts as the number of balls per game. We can initialize the Text field by getting the Text component attached to the balls left Text game object field we populated in the Inspector. You should also set the text to reflect the correct number of balls left here.

Add code to the `Ball OnBecameInvisible` method to call the new HUD method as appropriate. Don't worry that the balls left can go negative; we won't see that when we add functionality to end the game when that value gets to 0.

Run the game. When you lose a ball off the bottom of the screen, the balls left text display should reduce the number of balls remaining by 1.

Step 4: Add accurate scoring

For this step, you're adding scoring functionality to the game.

In the HUD script, add a field (marked with [SerializeField]) to hold the score Text object (as a `GameObject`). Populate the field in the Inspector.

Add static fields to the HUD script to hold the score `Text` object and the actual score. Add a static public method that a block can call to tell the HUD to add points to the score. In the body of that method, update both the score and the score text that's displayed.

Add code to the `Start` method to populate the score value and Text field. The score should start at 0. We can initialize the Text field by getting the Text component attached to the score Text game object field we populated in the Inspector. You should also set the text to reflect the correct score here.

Add a private field to the Block script to hold how many points the block is worth. Add a protected property with a set accessor so child classes can set the value of the field.

Add code to the Block script to call the new HUD method just before the block is destroyed. This isn't an optimal object-oriented solution because Block objects shouldn't really have to know about the existence of the HUD or the methods it exposes, but it's a reasonable solution given the C# knowledge we have at this point. Don't worry, we'll make this much better once we know about delegates and event handling.

Add a value in the configuration data CSV file for the Standard block points (I used 1 for this value) and add the appropriate fields and properties to the `ConfigurationData` and `ConfigurationUtils` class so the `StandardBlock` class (see next paragraph) will be able to access this value.

Create a new StandardBlock script as a child class of the `Block` class. Add fields for the standard block sprites, marking each field with `[SerializeField]` so you can populate it in the Inspector. Add code to the StandardBlock `Start` method to set the points the block is worth to the Standard block value using the appropriate `ConfigurationUtils` property. Also add code to this method to randomly select one of the standard block sprites for this block.

Remove the Block script from the StandardBlock prefab and add the StandardBlock script instead. Populate the three sprite fields in the Inspector.

Run the game. You should have 3 rows of standard blocks with a random assortment of your standard block sprites, and your score should increase by the standard block points amount each time you destroy a block.

That's it, you're done with this increment.