Basic Scripting

Table of Contents

Lab Introduction

In this lab, you will gain a surface level introduction to programming by interpreting and editing elements of pre-written scripts. You will not be required to create any scripts for this lab. **Information only relevant to artists will appear in blue.** **Information only relevant to programmers will appear in red.**

You will be adding a new enemy type to a simple minigame, converting both enemy types into Unity Prefabs, and then spawning your enemy Prefabs into the minigame.

Upon completion of this lab, you should have a basic understanding of what variables are and of potential ways to modify them. You will also learn how to create and edit Prefabs in Unity. Programmers will learn about variable protection levels and gain a surface level introduction to how different types of scripts interact with each other. The final part of this lab will walk you through setting up Unity Collab, Unity's built-in version control tool that will allow for easier collaboration for future projects in this class.

Lab Instructions

Editing Scripts

1. Select the Main scene from the Assets folder in the *Project* tab. The image on your screen should match the one seen in Figure 1.

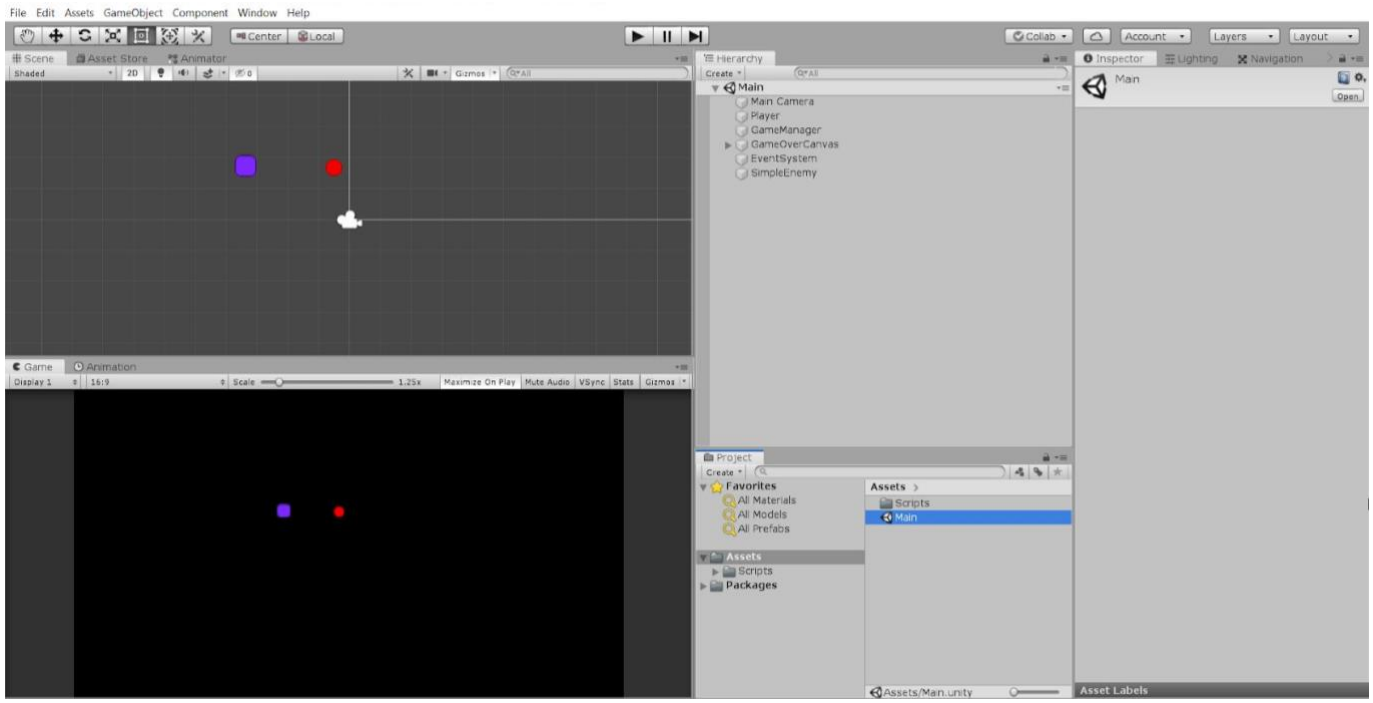

Figure 1

   a. Notice that the Player GameObject under Main in the *Hierarchy* tab corresponds to the purple square, while the SimpleEnemy GameObject corresponds to the red circle.
2. Press play and observe how the game currently plays. Use WASD or the arrow keys to move the Player around.
   a. If the SimpleEnemy object touches the Player, the game should end.

In order to make this rather dull minigame more interesting, we will add multiple enemies and ensure that the player can survive more than one hit before dying.

3. Open the Scripts folder, located under Assets in the Project tab. Navigate to the Player folder, find the script called **Health**, and double click to open it.
4. You should see several lines of code similar to the ones transcribed below. Read the section indicated below.

```
public class Health: MonoBehaviour {
      /// Non-programmers only need to look between >>>>> and <<<<<
      /// >>>>>

      // Most scripts will have a collection of Variables at the top.  Similar to their function
      // in mathematical equations, a variable is a name that is associated with a value.  In this
      // example, the variable startingHealth is set to a value of 1.  Following the variable is a
      // comment that explains its purpose.  Variables preceded by the keyword public will show up
      // in the Inspector tab when you attach this script to a GameObject in the Hierarchy tab.

      public int startingHealth = 1; // This is how much health you have before you die

      // <<<<< Non-programmers may stop reading here

      int currentHealth;
```

In addition to the **public** keyword used in the code block above, several other **variable protection levels** exist in programming:

- **Private** — The default setting, meaning that no other script may access this variable (including child scripts).
- **Public** — All scripts may access this variable. Additionally, it will show up as a field in the *Inspector* tab when this script is attached to a GameObject.
    - If you require a variable to be public but **not** show up in the Inspector, type **[HideInInspector]** before the variable declaration or on the line above it.
- **Protected** — Only child scripts may access this variable. No other scripts can.

Each GameObject with the Health script attached to it will have its own copy of the variables declared within the script (unless the variables are preceded by the keyword **static**). Public variables will remain set to their default values unless set to another one via the Inspector. **Inspector changes to public variables will override values set in scripts.**

We will now modify the **startingHealth** variable in the Inspector in order to allow our Player to survive more than one hit.

5. Exit out of this script and return to Unity, then select the Player GameObject in the Hierarchy. In the Inspector, displayed in Figure 2, locate the component titled **Health (Script)**. The component should have only one modifiable field, titled **Starting Health**. Change this value to any number greater than 1.

Checkoff Requirement: The Player should be able to survive more than one hit from an enemy.

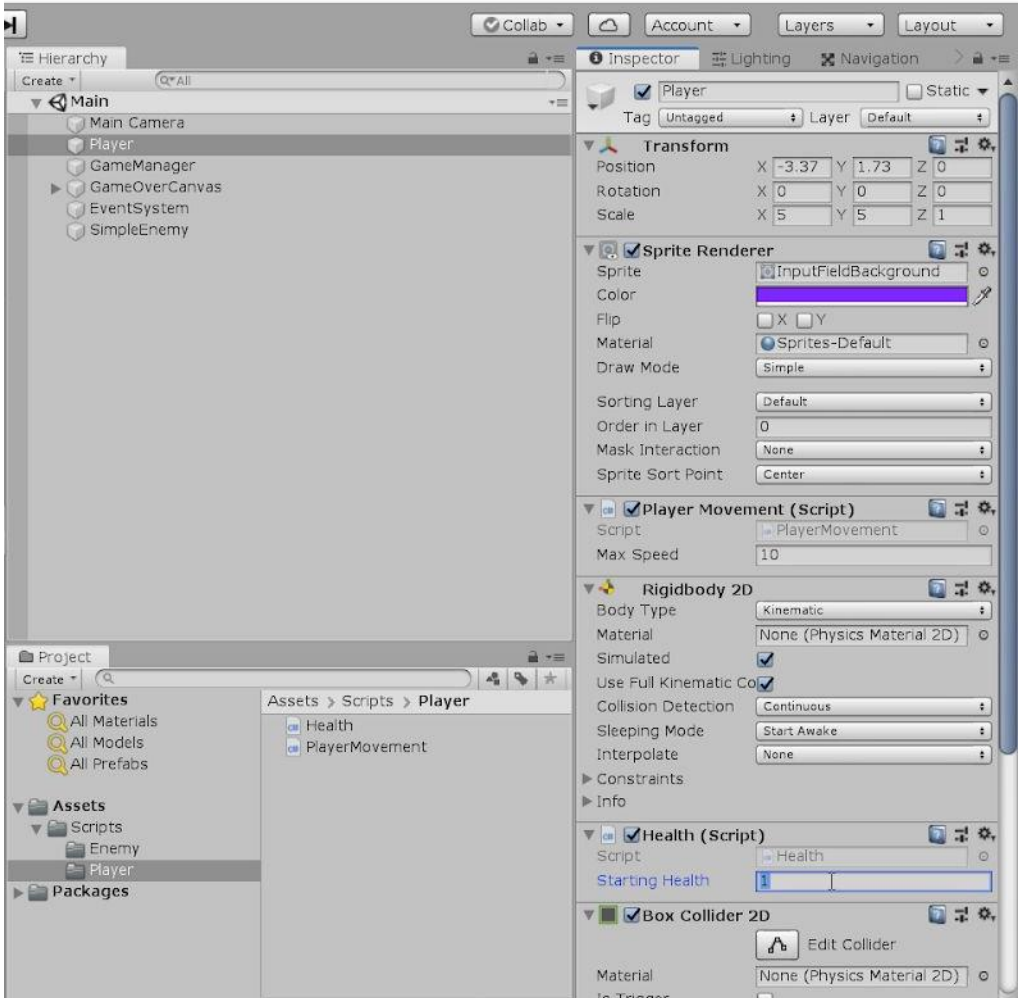

Figure 2

Prefabs

Creating Prefabs

A **Prefab** is a pre-defined GameObject that is saved as an **Asset**, in a manner similar to how one would save a script or an art asset. Prefabs are created by dragging an existing GameObject from the *Hierarchy* into the *Project* tab. Once a Prefab has been created, you can repeatedly drag it from the *Project* tab into the *Scene* in order to create multiple copies of the object.

6. Create a Prefabs folder within your Assets folder. Drag the **SimpleEnemy** GameObject from the Hierarchy into this new folder to create a new Prefab.
7. Drag the SimpleEnemy Prefab into the Scene to instantiate a new SimpleEnemy.

Notice the following:

- All instances of a Prefab have the same name and are numbered in order of creation.
- All instances of a Prefab are highlighted blue in the Hierarchy.
- When you click on a Prefab in the Hierarchy, the Inspector should display an additional tab labeled Prefab (Figure 3).

Figure 3

Editing Prefabs

You may frequently want to apply edits you make to a single Prefab across all Prefab instances (e.g. changing the color of all SimpleEnemies to be blue instead of red). Here are two ways to accomplish this:

1. Edit an instance of the Prefab in the Scene tab, then apply the change across all Prefab instances (Figure 4).
    a. Make a change to an instance of your Prefab.
    b. Select the drop-down menu labeled “Overrides”.
    c. Select “Apply All” to update the Prefab to match your current instance.

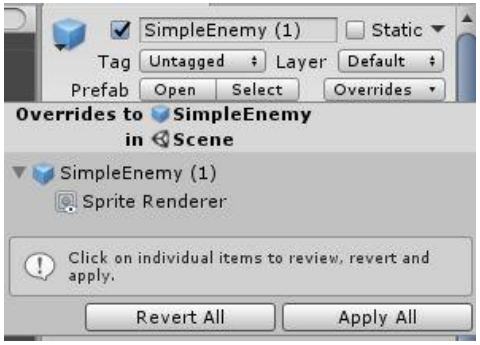

Figure 4

2. Edit the base Prefab in **Prefab Mode**, which will automatically apply your update across all currently instantiated Prefabs (Figure 5).
    a. Open the Prefab base in Prefab Mode.
    b. Make a change to the base Prefab.
    c. As long as the “Auto Save” toggle is on, your changes will automatically be applied across all instances of the Prefab. If not, press the “Save” button to apply your changes.

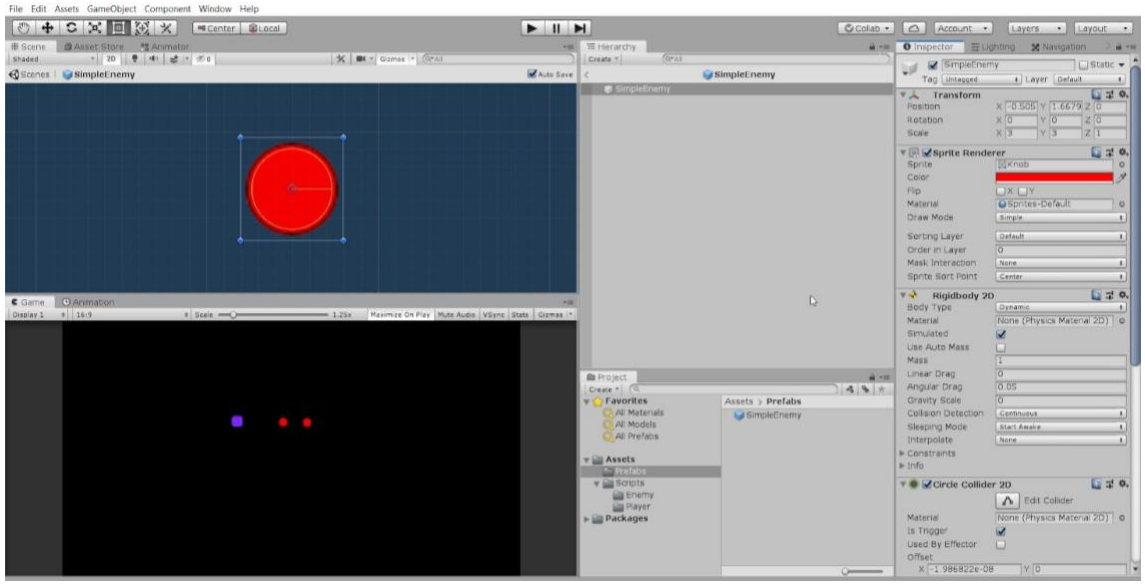

Figure 5

8. Make the default color of all SimpleEnemies blue using the first method (editing the **SpriteRenderer** component’s color field). Verify that your changes have been applied by instantiating a new SimpleEnemy afterwards.

Checkoff Requirement: All instances of SimpleEnemies should be blue.

Spawning Prefabs

9. Select the **GameManager** object from the Hierarchy tab and edit the **My Game Manager (Script)** component in the following ways:
    a. Expand the dropdown menu for **Enemy Spawns** by clicking the arrow on the left.
    b. Change the size from 0 to 2 (this is in order to make room for our next Enemy type).
    c. Drag the SimpleEnemy Prefab you created onto **Element 0** (or add it by clicking the circle located to the right of the word “Prefab”).

Verify that when you press play, SimpleEnemies continue to spawn indefinitely.

Creating Prefab Variants

10. Delete all SimpleEnemy instances (every SimpleEnemy object in the Scene/Hierarchy), but **do not delete the Prefab asset that you created in the Prefab folder**.

Checkoff Requirement: No Prefab instances should exist in the Scene.
We will now add a new enemy type to our minigame.

11. Right-click the Prefab asset in the Project tab and Navigate to **Create > Prefab Variant** (Figure 6). Rename this variant to “Sniper”.

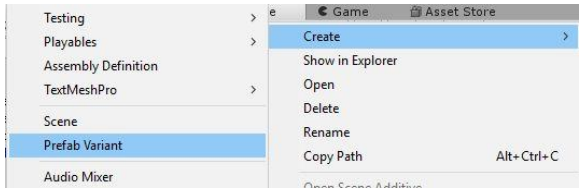

Figure 6

12. Make the following changes to the Sniper Prefab using Prefab Mode (the second method in the “Editing Prefabs” section):
    a. Change the color of the SpriteRenderer component to whatever color you’d like.
    b. **Do not remove the SimpleEnemyMovement Script**.
    c. Change the damage field on the **Attack** component to a value of 2.
    d. Select **Add Component** and search for **Sniper Movement**.

i. Set the speed field to a value of 16 and the spawn distance field to a value of 10 (you're welcome to experiment with different values if you would like).

e. Modify all variables in the **Enemy Data (Script)** component to match Figure 7. For a more detailed description of the purpose of each variable, read the comments located in the scripts.

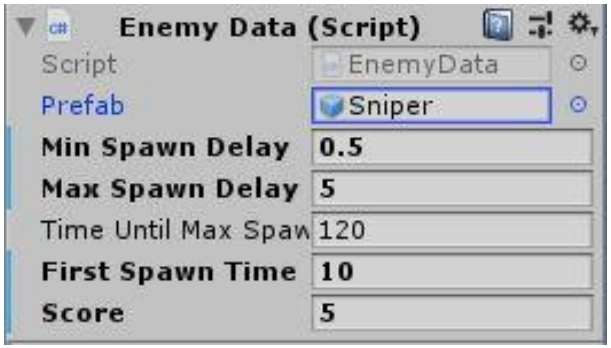

Figure 7

Checkoff Requirement: The Sniper enemy type should have a unique color.
Checkoff Requirement: The Sniper enemy type should begin spawning at 10 seconds.

13. Select the GameManager object from the Hierarchy tab and edit the **My Game Manager (Script)** component to add the Sniper to the GameManager:
    a. Drag your new Sniper Prefab onto Element 1 **0** (or add it by clicking the circle located to the right of the word "Prefab").

Verify that your My Game Manager component matches Figure 8.

Figure 8

Debugging

Press Play and notice the odd behavior of the Snipers. Their movement may be erratic (jittering and stuttering), or they may rush toward the Player at impossibly high speeds.

14. Go to your new Sniper prefab and remove the **Simple Enemy Movement** component.
15. Save the prefab and start the minigame. You should now observe that the Snipers spawn heading initially towards the Player, but proceed in a straight line rather than following the Player.

Checkoff Requirement: The Sniper enemy type should have a different movement pattern from the SimpleEnemy.
Artists may now proceed to the summary and checkoff below.

Open the scripts **MyGameManager.cs** and **EnemyData.cs**. These scripts combine a few different methods of making the Inspector for a script more useful:

- **[HideInInspector]** — Use this if you require a variable to be public (for example, in a struct or an array) but don't want it to show up in the Inspector
- Structs — Use these to create convenient groupings of variable names that will **remain** grouped in the Inspector
- Arrays/Lists — Use these to make adding more elements to your game more convenient. Rather than having to add a new public variable for each enemy type, this allows you to add a new enemy entirely in the Inspector.

Take a look at the relationship between **EnemyMovement.cs**, **SimpleEnemyMovement.cs**, and **SniperEnemyMovement.cs.** This is an example of how to utilize **Inheritance** (a concept you may be familiar with if you've taken CS61B).

- The **protected** keyword is important here.
- Functions that you intend to override must be visible to the child script (public or protected) as well as **meant** to be overridden (virtual or abstract).
- When a child class overrides a method, you must use the keyword **override**.
- When you override a method, it is good coding practice to call the parent method by using **base.methodName()**, wherein **base** refers to the parent class.
- Note that we separated Player and Enemy movement into different scripts; there are not many shared elements between them in the case of our minigame.

Notice how **Health.cs** and **Attack.cs** interact. This is an example of how to utilize the concept of **Composition.**

- When an object with the Attack script attached comes into contact with another object, it will check to see if that object has a Health script. If it does, it will call that Health script's takeDamage() function. It also checks to see if the health is less than or equal to zero, so that it can inform the GameManager that the game is over.
- **Composition** is the idea of building up behaviors through modular components. For instance, an enemy has a movement script and an attack script, and these scripts together make up the behavior of an enemy. In **Inheritance**, you build unique things up from the top down by adding new features to a parent class. In Composition, you build unique things from the bottom up by assembling different pieces together in order to achieve a desired behavior.
- Most games will utilize a combination of Composition and Inheritance.

Unity Collab

The final part of this lab will introduce you to Unity's built-in version control tool, **Unity Collab**. You may be familiar with the term **version control** if you've used Github or other similar software before; version control is a means of documenting your changes to a project as you're working, which allows you to track your project history for safety and debugging purposes. Unity Collab also allows for easy collaboration between team members working on a project. We will be covering the basics of Collab in order to prepare you for group projects that will take place later in the course.

16. Open the Collab tab in the upper right window. Sign into your Unity account if you haven't already done so, then verify that your screen matches Figure 9.
17. Click on **Start now** and allow Unity to complete its set-up procedure (this may take a few minutes). Once this step is completed, you should see a blue arrow next to the word Collab, as seen in Figure 10.

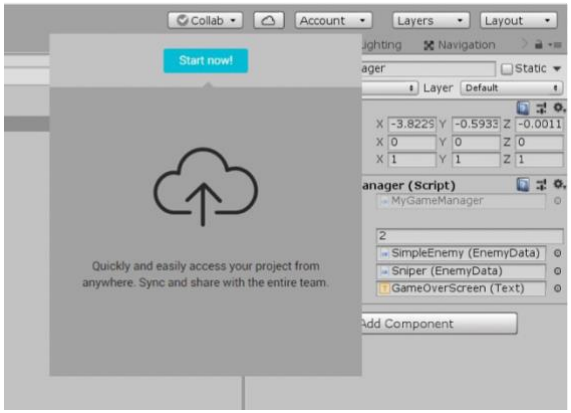

Figure 9

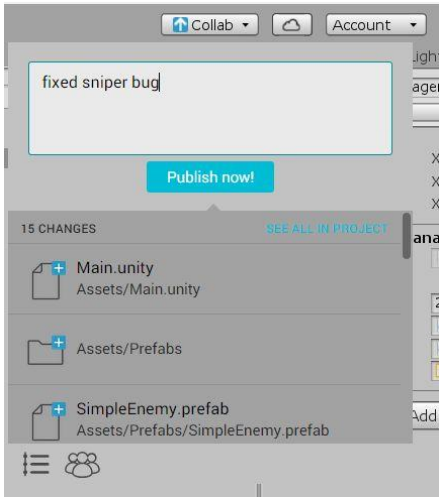

Figure 10

18. In the scrollable section, it will list all of the files you have made local changes to. In the provided text box, type in a descriptive message that summarizes what you've done with these file changes: for example, "fixed sniper bug" or "created new enemy type". This is equivalent to a Github commit message, which allows your teammates and yourself to know what was changed in each version of your project.
19. Once you've finalized all changes and typed your commit message, select **Publish now** and this version of your game will be pushed to the cloud.
    a. A **green** check mark next to the Collab label indicates that your local project is up to date with the latest version on the cloud, a **blue** arrow (as seen previously) indicates that your local project contains changes that have not yet been

pushed to the cloud, and a yellow arrow indicates that there are changes in the cloud that have not been applied to your local project (typically indicating that your teammates have made changes that you must pull in order to remain up to date).

We will now add the decal staff account to your project.

20. Select the icon of people at the bottom of the Collab window (Figure 11) and Unity will open a project manager webpage in your web browser. Click on the **Add a person or group** text box and enter Berkeley.gamedev@gmail.com, then click **Add**.

Figure 9

21. This will allow you to add collaborators to your project so that you can push and pull each other's changes.

Unity Collab also has functionality that allows you to review your version history or revert to previous versions. For more information, read the official manual here: https://docs.unity3d.com/Manual/UnityCollaborate.html

Checkoff Requirement: You should have set up Unity Collab for this project and invited the decal staff account.

Summary

Congratulations! You have now successfully created a minigame with two different enemy types that will spawn in and attack your Player. You have also gained a better understanding of scripts, variables, and Prefabs — things that will be invaluable in your further exploration of Unity. If you would like more practice with these concepts, please take a look at the optional challenge tasks on the last page of this lab.

Checkoff Requirements

1. The Player should be able to survive more than one hit from an enemy.
2. All instances of SimpleEnemies should be blue.
3. No prefab instances should exist in the Scene.
4. The Sniper enemy type should have a unique color.
5. The Sniper enemy type should begin spawning at 10 seconds.
6. The Sniper enemy type should have a different movement pattern from the SimpleEnemy.
7. You should have set up Unity Collab for this project and invited the decal staff account.

Challenges (Optional)

Here are some additional, more challenging tasks if you'd like extra practice with Unity. Keep in mind that these tasks are much more open-ended than the ones covered in this lab — please do not hesitate to ask for help if you choose to explore one or more of them.

1. Add a new enemy type to the minigame.
    a. Repeat the steps from the lab, but make a new movement or attacking script.
2. Polish the aesthetic of your minigame
    a. Add art assets to the SpriteRenderer, rather than the simple polygons we've been working with.
    b. Add sound effects.
    c. Look into particle effects and trail renderers.
3. Add a Player attack that can eliminate enemies.
    a. https://docs.unity3d.com/ScriptReference/Physics2D.CircleCast.html
    b. The enemies currently have hitboxes known as **Triggers**. Any colliders can pass through a Trigger collider without any physics taking place, but a user-specified event will be triggered (similar to something like a sensor on a sliding glass door at a grocery store).
        i. CircleCast will not detect Colliders that are Triggers; therefore, you will have to add another Collider to your enemies.
    c. If you use the Health script that we provided, you will have to explore ways to make sure that the enemies are unable to attack each other.
        i. Give the enemies a special tag or add them to a unique layer (found at the top of the Inspector).
        ii. Add a public variable to the Attack script that establishes what tag or layer the target must have in order to be attacked.
    d. To avoid the enemies experiencing physics interactions with their new colliders, go to **Edit>Project Settings>Physics 2D** and view the Layer Collision Matrix. You can then specify which layers can collider with which layers.