CAMOSUN COLLEGE

COMPUTER SCIENCE DEPARTMENT

ICS 123 - GAMING AND GRAPHICS CONCEPTS

LAB 7 – RETAINED MODE GUI AND ADDING AN EVENT SYSTEM

DUE: DEMO BY END OF LAB MON MARCH 20
You must work individually for this lab.
Submission Instructions are on the last page of the lab.

Objective:

To have the students add a retained mode GUI and Event System to their Game.

Preparation:

Supplemental Video Tutorials:

https://unity3d.com/learn/tutorials/topics/2d-game-creation/sprite-editor
https://unity3d.com/learn/tutorials/topics/user-interface-ui/ui-canvas

Background and Theory:

The Retained Mode GUI System and Advanced HUD:

Recall that in Lab 4 you created a VERY simple HUD which contained a simple aiming spot. In that case, we used Unity's *immediate mode* GUI system which calls the `OnGUI()` function every game frame to place UI elements. The *immediate mode* GUI system works well for very simple use cases but for more control over layout and functionality, you want to use the newer *retained mode* GUI System.

In the *retained mode* GUI system, UI elements are placed once and automatically re-rendered every frame. In addition, you place UI elements right in Unity's visual editor, allowing you to fine-tune your UI as you build it.

In this lab, we'll use the *retained mode* GUI system to polish the HUD in the game. Unlike 2D images used as textures, 2D images used for a HUD or a pure 2D Scene are called **Sprites**. In our case, we are overlaying a 2D GUI (which will include Sprites) over our 3D Scene. When using retained mode, the UI objects must be attached to a ***Canvas*** object. This is a special object that renders a 2D GUI in a 3D game.

Event-Driven Programming:

Games make extensive use of the Event-Driven programming paradigm. In this paradigm, instead of code being executed sequentially, code is executed in response to events. For example, the laser beam that the Enemies fire generate an Event when they collide with an Object. In *LaserShooter.cs*, the `OnTriggerEnter()` method is executed only when this event happens. Besides games, event-driven programming is used in GUIs. Every time a menu item is selected or a button in the GUI is pressed, an event is generated which causes its associated code to run. That

means as a developer, you don't know in what order your event-driven code will be executed. You must keep this in mind when using this paradigm.

Event System:

This lab is all about coding the 2D HUD (UI or GUI) that will interface with the Scene you have been building. When you think about it, the GUI and the Scene are completely separate modules and to practise sound Software Design, we should keep them ***loosely coupled***. By doing so, any changes we make with the Scene won't break the UI and vice versa. We'll have to employ a bit of overhead to accomplish this by adopting an *Event-based Messaging System*.

The messaging system will be based on custom Events you create. When something happens in the Scene, an Event will be broadcast and a UI Listener will get the Event and update the UI. Going the other way, when a UI Control gets changed, an Event will be broadcast and a Scene Listener will get the Event and update the Scene.

A script called *Messenger.cs* will handle this System and is already written for you. The script comes from wiki.unity3d.com, a wiki site which contains a huge number of Unity tips, tutorials, scripts, and other resources. Documentation for the Messenger can be found here. The script itself can be downloaded from D2L. The figure below outlines how the Event System will work:

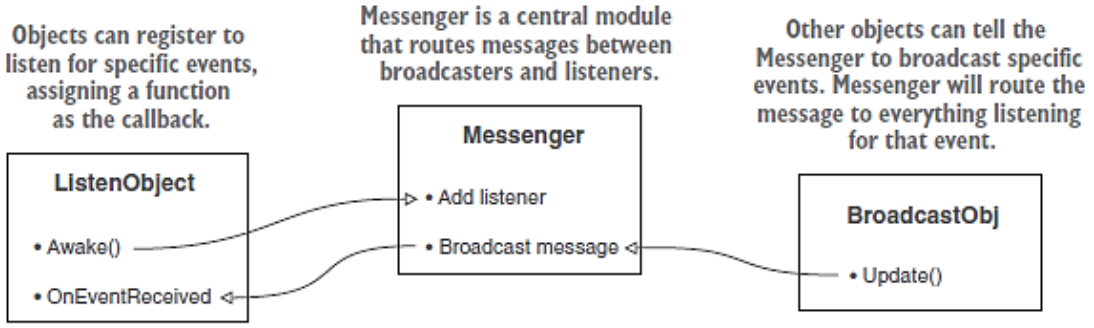

Figure 1: The Event System
(Credit: figure 6.17 from *Unity in Action*, used with permission)

An Object (or several) that wants to listen for a specific Event registers with the Messenger (Listeners are also called *Subscribers*). A different Object, tells the Messenger when this Event happens (called a *Broadcaster*). The Messenger, in turn, broadcasts this fact to all the Subscribers. Once a Subscriber knows the Event has occurred, a method (called the callback) is executed in response to the Event.

You will implement the Event System in Task 5.

Unity API References for this Lab:

https://docs.unity3d.com/ScriptReference/EventSystems.EventSystem.IsPointerOverGameObject.html
https://docs.unity3d.com/ScriptReference/UI.Text.html
https://docs.unity3d.com/ScriptReference/UI.Image.html
https://docs.unity3d.com/ScriptReference/UI.Button.html
https://docs.unity3d.com/ScriptReference/UI.Slider.html
https://docs.unity3d.com/ScriptReference/Time-timeScale.html
https://docs.unity3d.com/ScriptReference/PlayerPrefs.html

Tasks:

Tasks Part 1 – Import Lab 6's Assets into a New Project

1. You will start a new Project for Lab 7 but you will want to import the Scene and all the other Assets from Lab 6. First, open your Lab 6 project in Unity. Go up to the menu bar and select *Assets -> Export Package*. This should open the Exporting Package window:

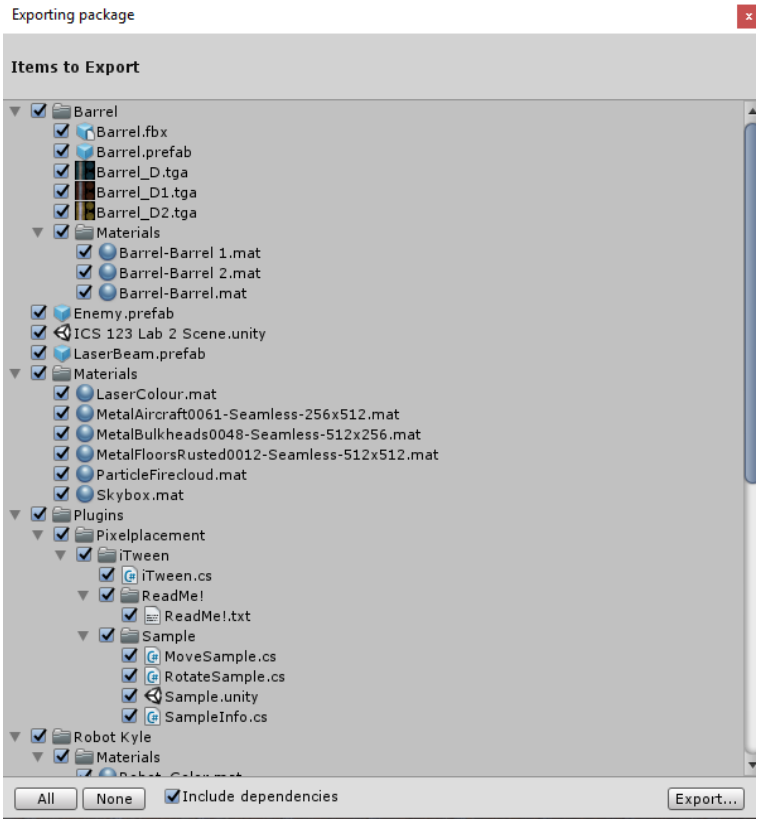

Figure 2: The Exporting package window

   Ensure that all your Assets are checked and that *Include dependencies* is checked. Hit the *Export…* button.

2. Save this package under your Lab 7 folder (create one if you haven't yet!). Save this new package as 'ICS123Lab6Assets'. A *.unitypackage* extension will automatically be added.

3. Now go up the menu bar and select *File -> New Project*. Name this new project 'ICS 123 Lab 7'. Hit the 'Create project' button. This will close your current Lab 6 Project and relaunch Unity with a new empty Lab 7 Project.

4. Once the new empty project is loaded in Unity, go up to *Assets* and select *Import Package -> Custom Package*. Browse for the package you created in Step 2. An Import Unity Package window will pop-up. Ensure everything is checked and hit the Import button.

5. In the Project Tab under Assets, you can double click on your Scene to load it up into the Scene Tab. You're done!

Tasks Part 2 – Paper Prototyping the GUI

1. First, you need to decide what information and functionality will be needed for the GUI. Assuming that killing an Enemy scores the Player 1 point, a score indicator would be a good idea. The Player's health, which up to now has been displayed in the Console Tab, should instead be included in the GUI. A 'Settings' button to allow the Player to change settings or exit the game is also needed.

2. Next, layout the GUI on a piece of paper (the paper prototype!). You could also use something like Visio or Dia (http://dia-installer.de/). The following figure is an example:

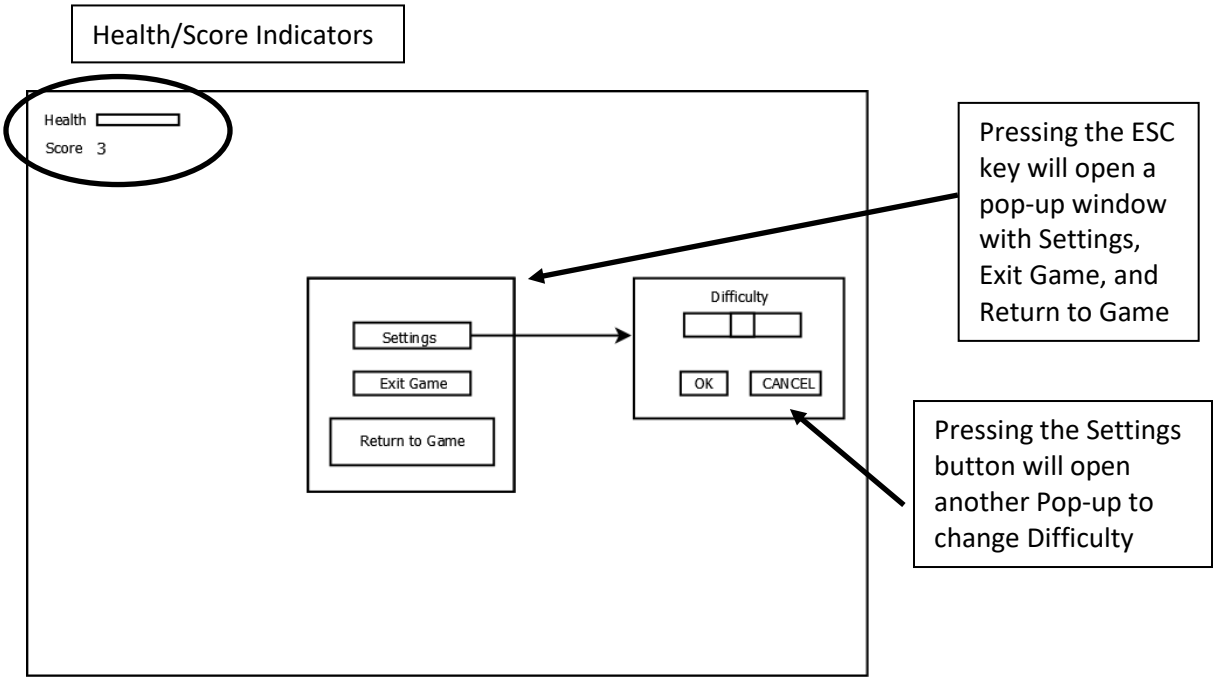

Figure 3: A 'Paper' Prototype of the GUI

Your GUI doesn't have to match that of the one above but you will need to show your paper prototype as part of your demo mark (see *Submission* at the end of the lab)

Tasks Part 3 – Building the UI in Unity

Tasks Part 3a – Adding the Canvas and Working in 2D Mode

1. Before you can start placing the 2D UI Elements in your GUI, you must add a Canvas object to your project. Go up to *GameObject -> UI -> Canvas*. Rename this to *HUD Canvas*. Note that when you add a Canvas object to your Scene, another object gets added automatically called `EventSystem`. This object is required in order for the UI to work properly and you can ignore it (but don't delete it!).

2. You are now going to work with the Unity Visual Editor in **2D mode**, instead of the default **3D mode**. To change to 2D mode, press the 2D button in the upper left of the Scene Tab window. When in 2D mode, the 2D Rect tool (to the left of the 3D Scale tool) will become active.

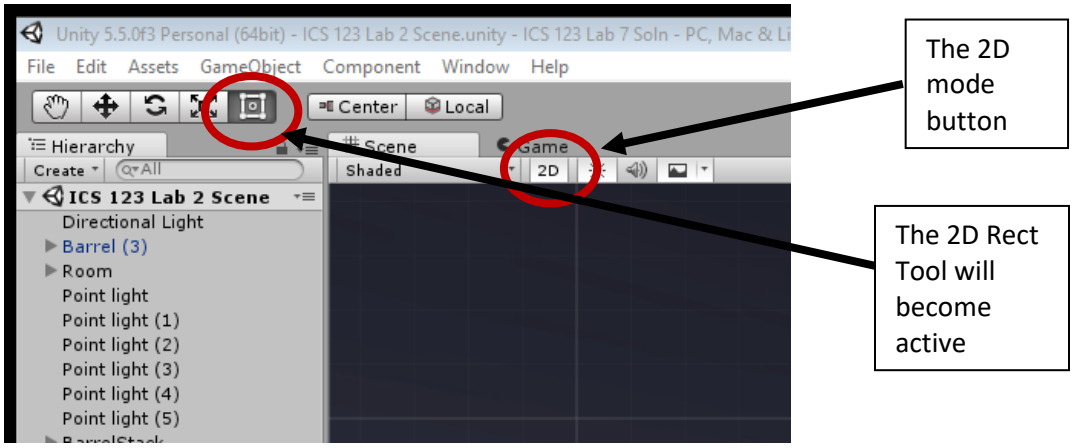

Figure 4: The 2D mode button

Double-click on the HUD Canvas in the Hierarchy Tab to zoom out and view the Canvas fully. You should change one setting in the Inspector Tab. Check where it says *Pixel Perfect* in the Canvas Component. This setting slightly adjusts positions of icons so as to make them as sharp as possible (pixel aligns them).

3. As you are working on the HUD, you may find the Skybox in the background distracting. To temporarily turn it off, hit the Effects button in the upper left, shown in the figure below.

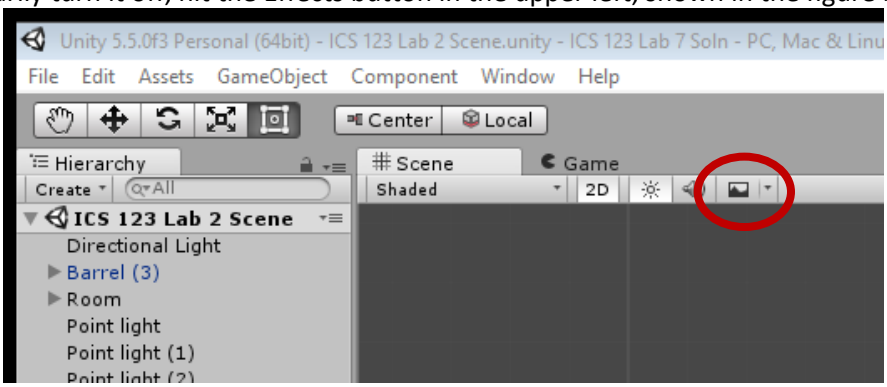

Figure 5: The Effects button. Press it to toggle Effects on and off.

Now you can start adding text, buttons, and images to the HUD. Again, the steps outlined here will be based on building a HUD like that of the Paper Prototype GUI mock-up shown in this lab. Your GUI may differ.

Tasks Part 3b – Adding UI Objects to the Canvas

1. Start by placing some Text. Select *GameObject->UI->Text*. In the Inspector Tab, in the Text box, type *Health*. You can change the Font Size, Style, and Color. As seen in the figure on the next page, the Font Color was changed to white, the Font Size was changed to 18 and the Font Style was set at Bold. In the Hierarchy Tab, rename the Text object to *HealthLabel*. (Note: the only font that you will have available to you is Arial. You can import other fonts into your project by putting the '.ttf' file in your Assets folder in File Explorer. Unity will then automatically import it). Place this Text object in the upper left corner.

2. Place two more Text objects. One will be the Score label (call it *ScoreLabel*) and the other

will actually hold the Score value (call it *ScoreValue* and set it to a slightly larger size of 24). Set the text for the Score value to some number so you see how it will look. Note that for all the Text objects, you can play around with the Text Alignment under the Paragraph section of the Text component. Place these two Text objects below the *HealthLabel* in the top left corner.

3. Next, place the Image that will hold the Health Bar. Select *GameObject->UI->Image* and place it in the upper left next to the *HealthLabel* object. In the Hierarchy Tab, rename the image to *HealthBar*. At this point, the new HUD's upper left corner should look something like the figure below.

Figure 6: Upper Left Corner of HUD

4. Let's make the Health Bar a little more exciting. A Sprite will replace the current boring rectangle. Activate the Asset Store (CTRL + 9 if the tab isn't open) tab. In the menu on the right, select *Textures & Materials -> Icons & UI*. Activate the Filters and select *Free Only*. In here you should see all sorts of Sprite (also called Icons) packages. You can of course pick whichever you like but for the example in this lab, the *Simple UI Elements* package was chosen.

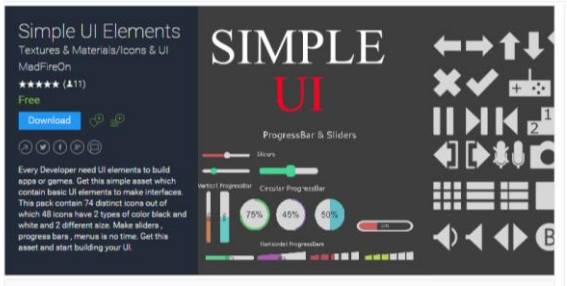

Figure 7: The Simple UI Elements Icons Package

5. Click on the Download button. The Import Unity Package window should pop-up. In this window, you have a few options depending on which Sprite package you chose. The *Simple UI Elements* package has each icon stored in separate PNGs in both black and white and in two different sizes. You could select individual icons you plan to use to import or, as in previous imports, select them all. Alternatively, you can select one or more of the Sprite sheets. A Sprite sheet contains all the icons in one PNG. The advantage of using a Sprite sheet is all the sprites are there in a single image file, de-cluttering your project. You can go back and pick and choose Sprites to use from this single sheet by slicing them. For this example, the *Extras.png* Sprite sheet will be the only image imported from this package.

6. You may need to re-organize your Project folders after it is imported. In this example, it created a *Raw and SpriteSheets* folder under and an *Assets* folder under the Project's

Assets folder. The *Raw and SpriteSheets* folder was moved under the Project's *Assets* folder, re-named to *Sprites* and the duplicate Asset's folder deleted. Remember to do this using the Visual Editor's Project Tab. Never do this in Windows File Explorer!! (NOTE: Re-naming a folder is a little tricky. You left click to select a folder then left click again on its name. Then wait about half a second for it to become editable).

7. If you imported the Sprite Sheet, you will need to 'slice' the sprite(s) you need (akin to a cookie cutter). Click on *Extras.png* in the Assets window and in the Inspector Tab change the Sprite Mode from *Single* to *Multiple*. Hit the *Apply* button. Next, press the *Sprite Editor* button. You should see the Sprite Editor Window come up with your Sprite Sheet loaded.

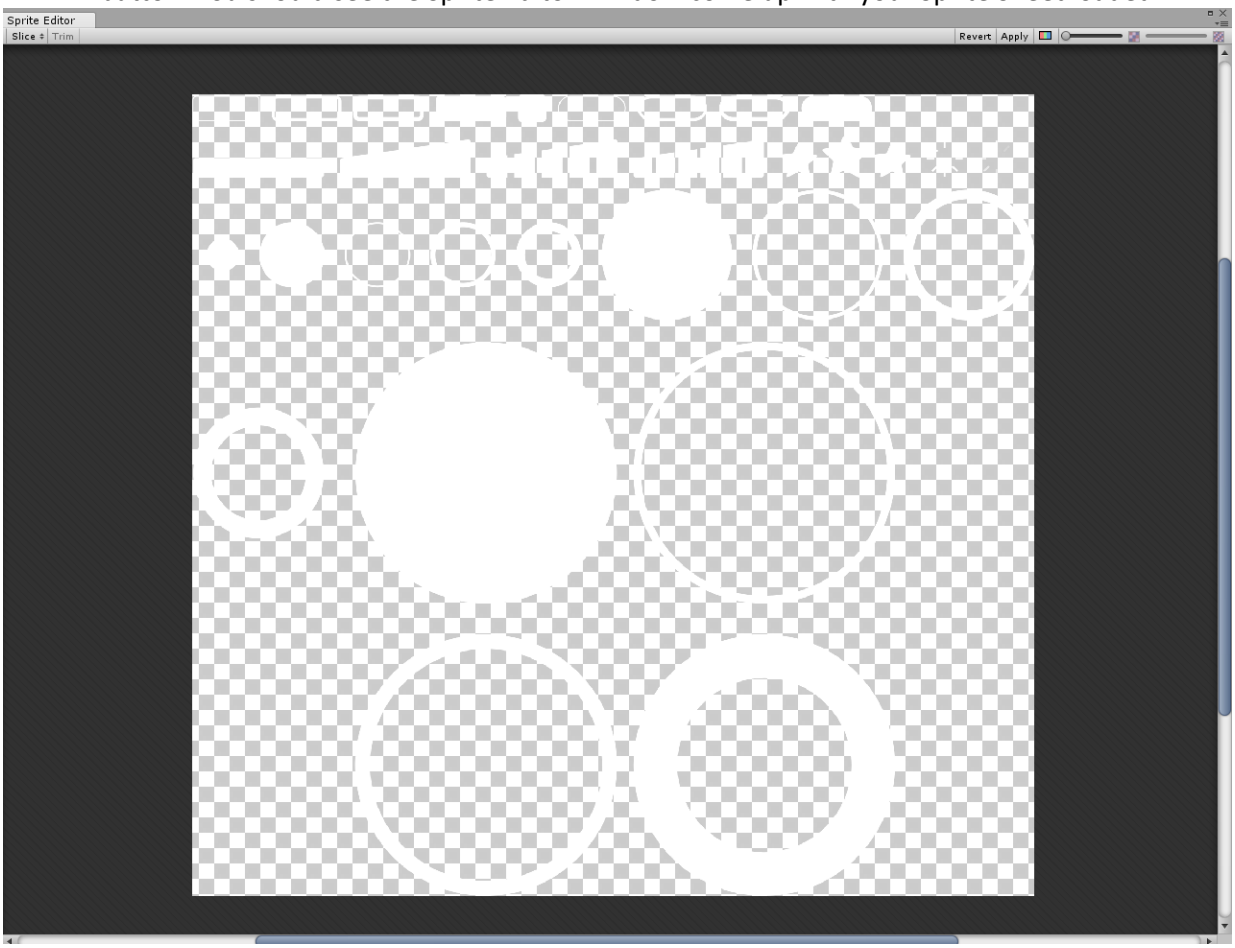

Figure 8: Sprite Editor Window with a Sprite Sheet Loaded

8. The Sprite sheet may look a little funky; that's due to the white sprites on a transparent background. To make this easier for slicing, hit the RGB/Alpha toggle button, circled in the figure below.

Figure 9: RGB/Alpha Channel Toggle Button

9. In this example, the desired Health Bar is the one in the second row third from left (you could choose differently). That sprite needs to be sliced. Simply drag a box around it using your left mouse button. It doesn't have to be exact, but ensure your box includes the

whole icon and none of the other surrounding ones. Next, hit the *Trim* button, located in the upper left of the Sprite Editor window. This will trim the box tight around the sprite. Then hit the *Apply* button in the upper right. Close the Sprite Editor Window. Back in the Assets Tab, you should see your new Sprite as child of the Sprite Sheet, ready for use.

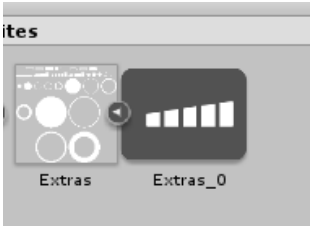

Figure 10: Extras_0 is the Spliced Sprite from the Extras Sprite Sheet

Now you can attach this Sprite to the *HealthBar* object. Select the *HealthBar* in the Hierarchy Tab. If you look over at the Inspector Tab and the Image Component, there's a *Source Image* slot. Drag the sliced sprite onto this slot and you should see it as the new Health Bar image in the UI. Then press the *'Set Native Size*' button. This re-sizes the UI object to match the image used. It'll be too big, so change the X and Y Scale to 0.5 (Note: You'll see the Z-axis here which may seem a bit odd since this is 2D. In 2D, the value for Z has to do with stacking. Sprites with lower Z values stack on top of ones with higher values. We won't be using this feature for the HUD so just leave it at 1). The color of the bar was changed to green (you don't have to do this; it will be done in code later) and as the health drops, the color will change to yellow and then red (handled in code).

10. Next, place a Panel in the center of your Canvas (*GameObject->UI->Panel*). Call it *OptionsPopup*. Add three buttons to it (you'll have to re-size the Panel to fit the buttons): one for *Settings*, one for *Exit Game*, and one for *Return to Game*. In the Hierarchy Tab, rename them to *SettingsButton*, *ExitButton*, and *ReturnButton* respectively. Finally, give the buttons linkage with the *OptionsPopup* panel by making them all children of the Panel. At this point, your UI should look similar to the figure below:

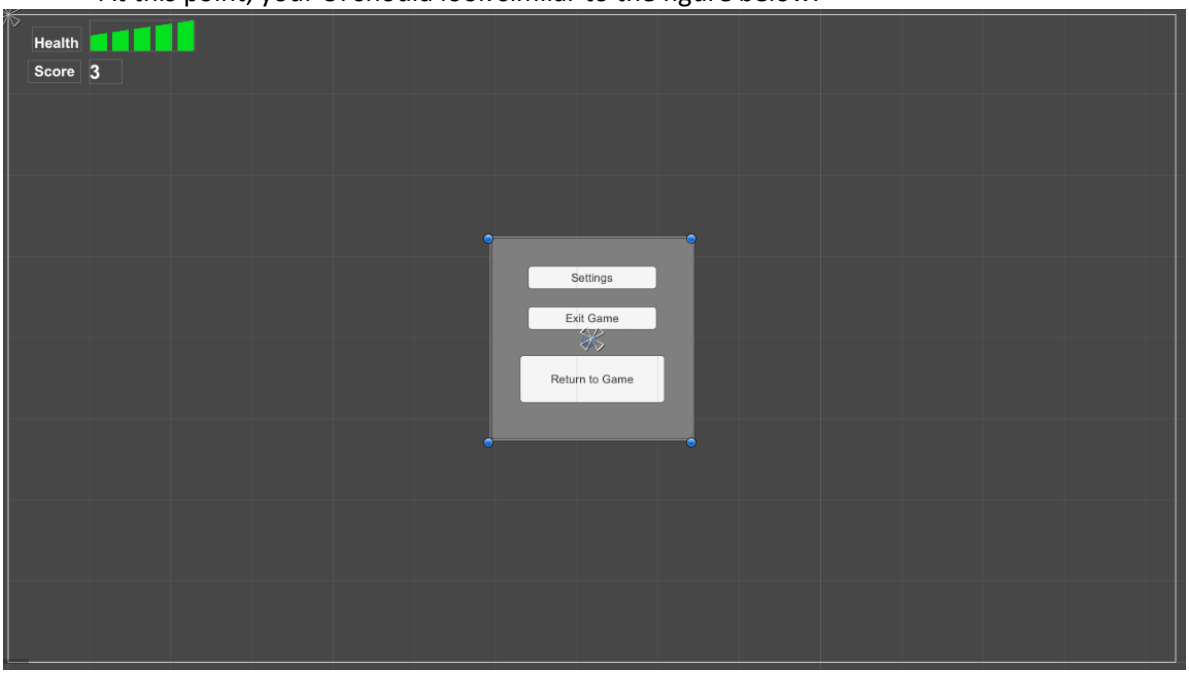

Figure 11: The HUD up to this point

11. Try playing your Scene now. A couple of things will probably stick out – The Options Popup is always on and you still have the aiming spot from the old HUD. The Options Popup you'll fix in the next Task but the aiming spot can be fixed now - time to get rid of it and upgrade the crosshair. Import a PNG image of a crosshair. You can find many by doing a Google image search. You may need to resize it – 128x128 is good. Make sure it has a trasparent background (this is a good reference https://docs.gimp.org/en/gimp-using-web-transparency.html). You can import it by either dragging and dropping the image into the *Sprites* folder of your Assets window or by selecting *Assets->Import New Asset…* in the menu.

12. You need to tell Unity that the crosshair is a Sprite not a Texture. By default, in 3D Projects, Unity assumes 2D images are for Textures. To fix this, select the image in the Assets window and in the Inspector Tab change the *Texture Type* to *Sprite (2D and UI)*. Hit *Apply* at the bottom.

13. Place a new UI image in the centre of the Canvas. Select *GameObject->UI->Image*. Rename it to *CrossHair*. Attach your crosshair image that you downloaded to the CrossHair UI Element by dragging it onto the *Source Image* box of the CrossHair Image Component (just like you did for the Health Bar). You may want to play with the Scale. Once you are satisfied, you can delete the old *immediate mode* GUI code in *RayShooter.cs*.

Tasks Part 3c – Anchoring and Fine-tuning the Position of the UI Objects

1. When you select any of the UI Objects you have placed, you'll notice a sort of four-bladed fan in the center of the Canvas. This is the object's anchor. Anchor's are important because the UI object's position will remain relative to it, even when the Screen size changes (Player's playing the Scene at different resolutions on different screens). To move a UI object's anchor, hit the *Anchor Presets* button on the right hand side of the *Rect Transform* component (this is used for UI objects instead of the regular Transform one) in the Inspector Tab (circled in the next figure).

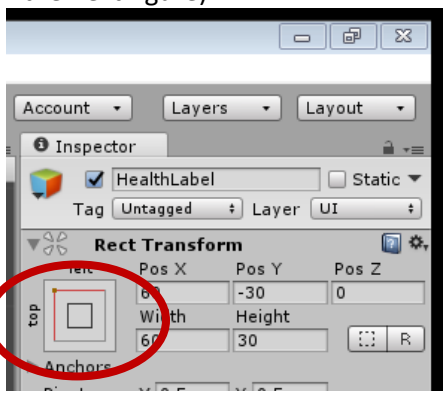

Figure 12: The Anchor Presets Button

2. This will bring up the *Anchors Presets* window. The best way to understand what you need to do here is in an example. For the *HealthLabel* in the top left corner, you want to set the anchor to the left top corner of the canvas. To do so, click on the top left corner preset (circled in the figure below).

Figure 13: The top right corner Anchor Preset

You should see the anchor for the gear icon move to the top left corner of the canvas. Now, the *HealthLabel's* position is relative to this. If the screen size changes, it will be anchored to that top left corner. Now, you can fine-tune its position. If you refer to the figure previously, in this example, the label has been placed 60 pixels away from the left edge of the canvas and 30 pixels from the top edge.

3. Now follow a similar process for the other UI objects. All the objects in the top left corner should be anchored to top left corner of the canvas. For the centre pop-up and crosshair, have the anchor set to the centre. The buttons' anchors will be with respect to the Panel.

4. At this point, the visual layout of your HUD should be complete (except for the Settings Pop-up). Next, you need to add the code to programmatically make the link with the UI objects.

Tasks Part 4 – Coding the UI

Tasks Part 4a – Preparation

1. Back in Lab 4 you locked the mouse to the centre of the view and turned it off in *RayShooter.cs*. The locking mechanism we'll keep but it will be handled by the UI now. **Remove** the following statements from *RayShooter.cs* that was added in Lab 4:

```
Cursor.lockState = CursorLockMode.Locked;
Cursor.visible = false;
```

You can also remove the `aimSize` variable as it is no longer needed.

2. Now you need a Script that will handle the UI events. You already added an Empty Controller Game Object to your Scene with a Scene Controller Script as a Component. That Script handles the 3D Scene. You'll use a different Script to handle the 2D UI. You could combine both Scripts but that is poor practise. Having separate Scripts does two things: a) enforces *separation of concerns* – one script handles the 3D Scene, the other the 2D GUI and, b) reduces the complexity of having all the code in a single Script. With this in mind, create a new C# Script and call it *UIController*. Attach it to the Controller Game Object.

3. Before you can start coding your UI Elements, you need to add the UI namespace to *UIController.cs*. Add it just below the other namespace declarations:

```
using UnityEngine.UI;
```

Tasks Part 4b – Coding the ScoreValue Text Object

1. The first element you will tackle is the *ScoreValue* Text UI Object. To start, make a private class scope variable to hold an integer that will store the score:

```
private int score;
```

2. You want to make a connection between your program code and the UI object. Add a private variable with class scope. We need access to this variable in Unity, so we need to use the `[SerializeField]` attribute:

```
[SerializeField] private Text scoreValue;
```

   This will hold a reference to our *ScoreValue* UI Object.

3. Now, in the `Start()` method:

```
// init score
score = 0;
scoreValue.text = score.ToString ();
```

4. Save your Script. Back in Unity's Visual Editor, when you select the Controller Game Object, you should see a *Score Value* box in the UI Controller of the Inspector Tab. To make the link with your *ScoreValue* UI Object, drag it from the Hierarchy Tab onto the *Score Value* box of the UI Controller component in the Inspector Tab (this is the same sort of thing you did to link Prefabs with Scripts).

5. Play the Scene. The Score should change to 0!

Tasks Part 4c – The Health Bar

1. First, you need to make some changes in the Image Component of the Health Bar UI Element in order for it to work properly. Ensure *Image Type* is **Filled**, *Fill Method* is **Horizontal**, and *Fill Origin* is **Left**. *Fill Amount* can be any value. You can play with that slider to see the effect it will have on the Health Bar.

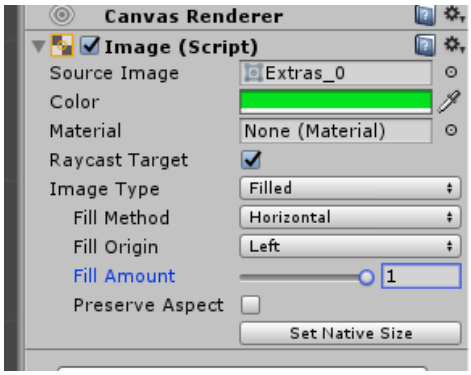

Figure 14: Image Component of the Health Bar

2. In *UIController*, create a private `int` variable with class scope and call it *_health*.

3. Just like the *ScoreValue* UI Object, you need to create a variable to hold a reference to the *HealthBar* UI Object. Add the following declaration below the `scoreValue` one:

```
[SerializeField] private Image healthBar;
```

4. Now, add the following statements in `Start()`:

```
healthBar.fillAmount = 1;
healthBar.color = Color.green;
```

The `fillAmount` variable is a percentage so this statement sets the image fill to 100% (all bars full). Finally, the fill color is set to green. You may have already done this in Unity but this is how you do it programmatically. You'll change it later to `Color.yellow` and `Color.red` as the Player loses health!

Tasks Part 4d – The Options Pop-Up Panel

1. First, create a new C# Script and call it *OptionsPopup*. Add the `using UnityEngine.UI;` namespace below all the other ones at the top of the file then code the class like the figure on the next page:

```
    public class OptionsPopup : MonoBehaviour {
1       [SerializeField] private Image crossHair;

2       public void Open() {
            // pause game & turn off crosshair
3           Time.timeScale = 0;
4           crossHair.gameObject.SetActive (false);
            // turn on popup
5           this.gameObject.SetActive (true);
           // activate mouse
6           Cursor.lockState = CursorLockMode.None;
        }

7       public void Close() {
            // turn off popup & turn on crosshair
8           this.gameObject.SetActive (false);
9           crossHair.gameObject.SetActive (true);
           // lock the mouse cursor to centre of view
10          Cursor.lockState = CursorLockMode.Locked;
        }

11      public void OnSettingsButton() {
        }

12      public void OnExitGameButton() {
13          Application.Quit ();
        }

14      public void OnReturnToGameButton() {
            //unpause game & close popup
15          Time.timeScale = 1;
16          Close ();
        }

17      public bool IsActive () {
18          return this.gameObject.activeSelf;
        }
    }
```

Figure 15: The OptionsPopup Class

Discussion of code:

Line 1:

Exactly like you did in *UIController* for other UI Objects, you need to declare a variable to hold a reference to the *CrossHair* UI Object here because you want to turn it off when the Popup menu is active.

Line 2:

This method will get called when the ESC key is pressed. This will be handled in *UIController*. You'll do that shortly.

Line 3:

`Time.timeScale` is a handy static variable that controls the time in the game. Normally, this is set to 1 (real-time). By setting this variable to 0, you're essentially

stopping time (in the game, of course!!) which causes the game to pause – handy, while the popup menu is active.

Line 4: De-activates the crosshair. The `crossHair.gameObject` is how you get a reference to the *CrossHair* game object in the HUD. In this case, you de-activate it. With the popup active, it would be annoying!

Line 5: Activates the Popup window. `this.gameObject` is the Game Object that this Script is a component of (in this case, the *OptionsPopup* Game Object in the HUD). You can think of it as the reverse of using `GetComponent`. In this case, you want' to activate the popup.

Line 6:
This makes the mouse cursor visible and frees it so that it can be used with the menus.

Line 7:
This method will get called once when the UI is initialized and then again when the *Return to Game* button is pressed.

Lines 8 & 9:
This basically reverses what was done in the `Open()` method. The Options popup is de-activated and the crosshair re-activated.

Line 10:
Like in previous labs, we'll lock the cursor to the centre of the view and make it invisible.

Line 11:
This will be called when the *Settings* button is pressed. You'll code this later in this lab.

Line 12 & 13:
Pressing this button will cause the game to exit completely. It won't work right now because we are running our labs within Unity. Later, when you make the game a stand-alone executable, this should work.

Line 14:
This will be called when the *Return to Game* button is pressed.

Lines 15 & 16:
Since the Player is going to return to the game, we need to get the time running again. Then, we call `Close()` to de-activate the popup and re-activate the crosshair.

Lines 17 & 18:
We can use this method to check whether the *OptionsPopup* is currently open or not. This will be useful in *UIController*.

2. With the code now complete, make *OptionsPopup.cs* a component of the *OptionsPopup* UI Object. Drag the script from the Assets window onto the *OptionsPopup* Game Object in the Hierarchy Tab.

3. With the *OptionsPopup* Game Object selected in the Hierarchy Tab, create linkage between it and the *CrossHair* Game Object by dragging the *CrossHair* onto the *Cross Hair* box in the *Options Popup* Component of the Inspector Tab.

4. Now you need to create a link between your *UIController* Script and the *OptionsPopup* Game Object much like you already have for the *ScoreValue* and *HealthBar* UI Objects. In *UIController.cs*, add the following instance variable below where you declared *healthBar*:

```
[SerializeField] private OptionsPopup optionsPopup;
```

5. Recall that back in Task 3, when you play-tested your HUD, the Options Pop-up was always activated. Let's fix that! In the `Start()` method add the following statement:

```
// close options pop-up on start of game
optionsPopup.Close ();
```

6. So how does it get opened? In the `Update()` method, we'll check every frame whether the ESC key has been pressed. As long as it hasn't already been opened, we'll open it:

```
if (Input.GetKeyDown ("escape") && !optionsPopup.IsActive()) {
    optionsPopup.Open ();
}
```

7. Play the Scene. No Popup! Hit ESC. The crosshair should de-activate and the popup activate!

Tasks Part 4e – The Return to Game Button

1. You need some way to cause the `OnReturnGameButton()` method in *OptionsPopup* to be called when the Button is pressed in the UI. This link is done in the Visual Editor. Select the *ReturnButton* UI Object in the Hierarchy Tab. Near the bottom of the Button Component in the Inspector Tab, you'll see the *On Click ()* Panel. Hit the '+' at the bottom (circled in the figure below).

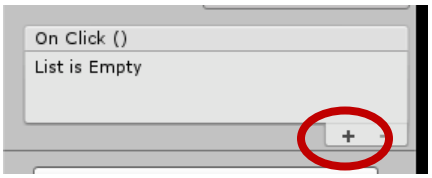

Figure 16: The On Click () Panel of the ReturnButton

2. A small box will appear. Drag the *OptionsPopup* UI Object (NOT the Script) from the Hierarchy Tab into this box. Once you do this, a function drop-down will be enabled (circled in the figure on the next page).

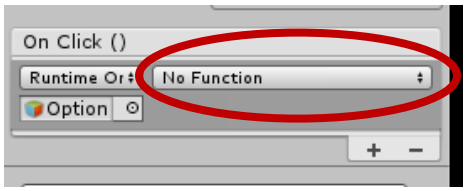

Figure 17: The Function drop-down bar

Select *OptionsPopup->OnReturnToGameButton ()*. That will create the link! Save your Scene and play it. Test that the button works!

Tasks Part 4f – The Settings Button and the Settings Pop-up

1. Your turn! From previous tasks, you have all the knowledge you need to make the Settings Pop-up. Your Settings Pop-up should resemble the following figure:

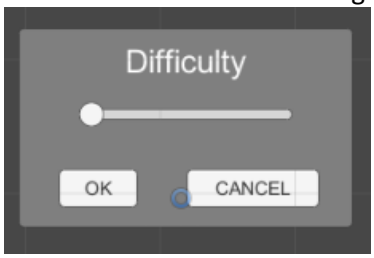

Figure 18: The Settings Popup

Implement this popup. Follow the steps below:

- You can temporarily disable the *OptionsPopup* and *CrossHair* Objects in the Visual Editor to make it easier to design the Settings Pop-up. In the Inspector Tab, beside the Object's title, is a checkbox. Uncheck this to turn the Object off.

- Add a new Panel anchored to the centre of the Canvas. Call it *SettingsPopup*. Add a Text, Slider, and Two Button UI Objects. Name them *DifficultyLabel*, *DifficultySlider*, *OKButton*, and *CancelButton* respectively. Position and anchor them appropriately.

- Create a new C# Script called *SettingsPopup.cs*. Include `Start()`, `Open()`, `Close()`, `OnOKButton()`, and `OnCancelButton()` methods. You will also need two `[SerializeField]` variables: one for the Options Popup and another for the Slider. Don't forget to add `using UnityEngine.UI;` at the top of the file.
    - In the `Open()` method, you need to activate the popup. In the `Close()`, de-activate it.
    - In both the `OnOKButton()` and `OnCancelButton()`, you need to de-activate the Settings Popup and re-activate the Options one. Use the `Open()` and `Close()` methods from OptionsPopup.cs as examples on how to do this.

- In *UIController.cs*, you will need to add a `[SerializeField]` variable to reference your Settings Popup. In `Start()` you'll have to make sure that Settings Popup is closed.

- Now you can code the Settings Button in *OptionsPopup.cs*. You'll need to add a

`[SerializeField]` variable to reference your Settings Popup. In `OnSettingsButton()` you need to close the Options Popup and open the Settings Popup.

- Finally, you need to link all the game objects to those `[SerializeField]` variables you declared using the Inspector Tab of the Visual Editor.
    - Link your *SettingsPopup* to the UI Controller Script Component of the Controller Game Object.
    - Link your *SettingsPopup* to the Options Popup Script Component of the *OptionsPopup* Game Object.
    - Link the *OptionsPopup* and *DifficultySlider* (Note: the variable you declare for *DifficultySlider* should be of type `Slider`) to the Settings Popup Script Component of the *SettingsPopup* Game Object.

- Lastly, link the *OK* and *Cancel* Buttons with their appropriate method in *SettingsPopup*. Follow the same kind of procedure as Task *4e*.

2. There's a few more things that need to be done with this Popup. First, the Difficulty Slider. Exactly what will this control? You have three options: the easy way, the hard way, or the uber hard way.
    a. **The easy way** – have the difficulty slider control the speed of the Enemy, the speed of the LaserBeam, and/or the speed of the Player. You can have the single Slider control all three or just your choice of one or two of the speeds.
    b. **The hard way** – have the difficulty slider control the number of Enemies. This is hard because it involves using a dynamic array (a List in C#). The number of Enemies will change at run-time which means you will probably have to make some changes to your *SceneController.cs* Script.
    c. **The uber hard way** – combine both a. and b. using multiple sliders. This will be a bit of work so be prepared!

3. Whichever you choose, you will need to change the Slider range. Select the Slider in the Hierarchy Tab and set Min Value to 1 and Max Value to 10. You may have to playtest these numbers and change accordingly depending on which option you chose from step 2. Also, check the **Whole Numbers** option.

4. Add two more Text UI Objects to your Settings Pop up. One will be label that describes what the Slider does. The other will hold the current value of the Slider. The below figure gives an example of the Difficulty Slider controlling the number of Enemies:

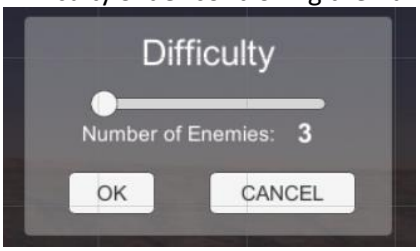

Figure 19: EnemiesLabel and NumEnemiesValue Text Objects Added to Popup

5. Add the following method to *SettingsPopup.cs*. This method assumes the Slider will control the number of Enemies. You will want to change the variable names if your Slider

does something else:

```
public void OnDifficultyValue(float numEnemies) {
        numEnemiesValue.text = numEnemies.ToString ();
    }
```

This will take the value from the Slider and set the Text of the *NumEnemiesValue* Game Object to that value. In order for this to work, a reference to the Text UI Object is required. Add a `[SerializeField]` variable of type `Text` (in this case the variable name is *numEnemiesValue*) and link the Text UI Object with the Script.

6. Getting the `OnDifficultyValue` method to be called is exactly like what you have done with the Buttons except that in the Inspector Tab instead of *On Click()* it's *On Value Changed (Single)*. Make the link with your Slider method in *SettingsPopup.cs*.

7. The Difficulty setting should probably be stored on the Player's computer so that it doesn't get reset every game. To do so, add the following statement in the `OnOKButton()` method in *SettingsPopup.cs*:

```
PlayerPrefs.SetInt ("numEnemies", (int)difficultySlider.value);
```

This will store the setting of the Slider (see the API reference for `PlayerPrefs` to see where). The cast to `int` is necessary because the Slider stores its setting as a `float`. To retrieve the setting, in the `Start()` method, add the following statement:

```
difficultySlider.value = PlayerPrefs.GetInt ("numEnemies", 1);
```

The `1` in the above is a default value to use in case the setting hasn't been stored yet. Typically, this should match your *Min Value* that you set for the Slider in the Inspector Tab.

8. Play your Scene. The GUI should completely work at this point except for the Exit Game Button. Move your slider to see that the value changes. Save your setting, stop and re-start the Scene to see that it is being saved. You'll probably quickly notice that the UI and the Scene aren't communicating with each other (Health Bar and Score aren't changing and the Difficulty setting doesn't do anything). This will be fixed in the next task!

Tasks Part 5 – Implementing the Event System

Tasks Part 5a – Scene to UI Event Handling

1. We haven't yet made the connection between the GUI code and the Scene. As explained in the Preparation section of this lab, we'll implement an event-based messaging system to make this connection. Download the ***Messenger.zip*** file from D2L. Unzip it and then add it to your Project.

2. Create a new C# Script and name it *GameEvent*. This will be a static class that will simply

store your Events. Get rid of all the boilerplate code in this class and change it so that the Script's contents match the code below:

```
public static class GameEvent {
    public const string ENEMY_DEAD = "ENEMY_DEAD";
    public const string DIFFICULTY_CHANGED = "DIFFICULTY_CHANGED";
}
```

The static keyword means that this class can't be instantiated. Our events are just strings and are declared constants.

3. In UI Controller, add the following method:

```
void Awake() {
        Messenger.AddListener (GameEvent.ENEMY_DEAD, OnEnemyDead);
}
```

`Awake()`, like `Start()`, is called automatically once at the start of the lifetime of the Script, however, `Awake()` is always called before every objects' `Start()`. By registering the Listener in `Awake()`, we ensure that the Listener is active before anything 'starts'. The last argument, `OnEnemyDead`, is the callback method, i.e. the method called when this listener gets the event.

4. Implement the `OnEnemyDead` method. This method should have a private access modifier, update the `score` variable, and update the *scoreValue* UI Text.

5. You also need to add an `OnDestroy()` method. This gets automatically called when the Game Object this Script is attached to gets destroyed. In this method, you need to remove the Listener:

```
void OnDestroy() {
        Messenger.RemoveListener (GameEvent.ENEMY_DEAD, OnEnemyDead);
}
```

For every Listener that you add, you must always call `RemoveListener` in the `OnDestroy()` method.

6. Finally, in *ReactiveTarget.cs*, in the `DeadEvent()` method just before the statement where the Enemy Game Object is destroyed, broadcast the Event that the enemy is dead.:

```
Messenger.Broadcast (GameEvent.ENEMY_DEAD);
```

7. Play your Scene! The Score should be updating in your HUD!

Tasks Part 5b – UI to Scene Event Handling

1. You can go the other way, of course, AND you can pass a value with your Event. For example, if your Difficulty Slider will control the speed of the Enemy, you'll have to have *WanderingAI.cs* register as a Listener:

```
void Awake() {
        Messenger<float>.AddListener (GameEvent.DIFFICULTY_CHANGED,
            OnDifficultyChanged);
}
```

2. Then, implement the `OnDifficultyChanged` method. This method would have a single float parameter. It would update the `enemySpeed` variable already declared in this Script.

3. Don't forget to remove the Listener in the `OnDestroy()` method:

```
void OnDestroy() {
        Messenger<float>.RemoveListener (GameEvent.DIFFICULTY_CHANGED,
            OnDifficultyChanged);
    }
```

4. The UI would be the broadcaster this time, specifically, *SettingsPopup*. In *SettingsPopup.cs*, in the `OnOKButton()` method:

```
Messenger<float>.Broadcast(GameEvent.DIFFICULTY_CHANGED,
            difficultySlider.value);
```

5. Now that you've seen how to do Scene to UI events and UI to Scene events, **finish implementing your Event System**. You'll definitely need to add an Event for the Health Bar (broadcast with the health value from *PlayerCharacter.cs* and add a listener in *UIController.cs*), one to stop the Camera's view from changing when the Options Popup is activated (broadcast from *OptionsPopup.cs* add a listener and boolean variable to *MouseLook.cs*), and one to stop the left mouse button shooting when the Options Popup is active (add a listener and boolean variable to *RayShooter.cs*). Good luck!

Submission:

Demo your Scene to the Lab Instructor:

- **Show your Paper Prototype for your GUI (either on Paper or Digitally made in Dia or Visio).**
- **Show that your GUI has at least a Health Bar, a Score Value, an Options Popup with three buttons, and a Settings Popup with a Slider and two buttons.**
- **Show that you have removed the immediate mode GUI and have a new crosshair.**
- **Show that your GUI's functionality works except for the Exit Game Button:**
  - **Hitting ESC opens the Options Popup and the Game Pauses.**
  - **Hitting Return to Game closes the Options Popup and un-pauses the game.**
  - **Hitting the Settings Button opens the Settings Popup and closes the Options Popup.**
  - **You can change the Slider and hitting OK stores the value.**
- **Show that your Event System has been fully implemented:**
  - **The Score increments by one for every Enemy killed.**
  - **The Health Bar gets reduced and changes color from Yellow to Red.**
  - **When GUI popups are open, the camera view also pauses.**
  - **Changing the Difficulty affects the Scene: speeds change or the number of Enemies change or both**

- A successful demo scores you 20 marks.