CAMOSUN COLLEGE

COMPUTER SCIENCE DEPARTMENT

ICS 123 - GAMING AND GRAPHICS CONCEPTS

PROJECT – FINISHING UP AND DEPLOYING YOUR PROJECT

Objective:

To have the students re-load their Scene when the Player's health has reached zero and to have them deploy their Game as a standalone executable.

Preparation:

Background and Theory:

There isn't anything new to learn in this lab other than how to re-load a Scene and make the Game stand-alone. Both are relatively easy.

Labs Summary:

Through the course of these labs, you have learned concepts and procedures for designing a modern 3D Game using Unity. Although we covered most concepts, there were a few concepts we didn't have time for:

- *2D Game Development* – Unity was initially designed for 3D Game Development however, it has since added 2D Game Development. Although not as mature as its 3D tools, the workflow for 2D development is fairly similar to 3D development.

- *Making the Player Third-Person* and how to work with the Camera in this case.

- *Managing Inventory and more robust Software Design* – I touch on Software Design in this lab but it is something that is out of scope for this course.

- *Adding Networking Capability to the Game* – Most games today allow you to play with other human opponents over the Internet. This is a pretty serious endeavour. In fact, in Game Studios, they typically have a team doing just the networking aspect of a game.

- *Background Music and Sound Effects* – As we discussed at the beginning of the course on Game Design, sound can really enhance your game and ignite an emotional response from the Player.

In addition, almost all the topics that we <u>did</u> cover could be expanded on. For example, adding a Splash Screen, cut-scene Animations, an outdoor Scene, different weapons, jumping and crouching, etc. There's a lot of topics!!! That being said, on completion of these labs and this course, you should have a fairly solid foundation in Game Design and Development on which to build a career.

Unity API References for this Lab:

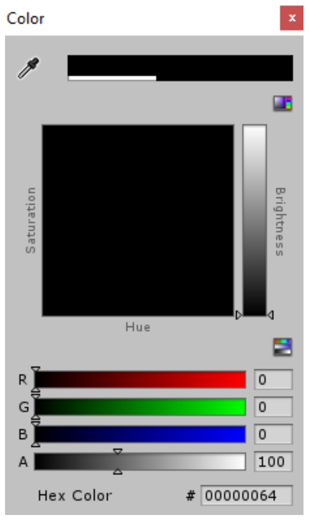
https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.LoadScene.html

Tasks:

Task 1 – Adding a 'Game Over' Screen

1. Put the Unity Visual Editor in 2D Editing Mode. Zoom so that your HUD Canvas fills the Screen. Add a new Panel – go up to the menu and select *GameObject -> UI -> Panel*. Name this Panel *GameOverScreen*. Ensure that the Panel is the exact same size as the HUD Canvas, i.e. that it covers it completely.

2. With the Panel selected, in its Image Component in the Inspector Tab, change its color to black and set the Alpha Channel to 100 so that the Color Picker looks like the Figure below:



R 0
G 0
B 0
A 100
Hex Color # 00000064

Figure 1: The Game Over Screen's Color Picker

   You can try Playing your Scene and see the effect that this will give. It should darken the view without completely obscuring it.

3. Next, add a Text UI Object to the Panel. Anchor it somewhere near the center of the Panel. Change the text to say 'Game Over'. Change the color and size of the Font.

4. Add two UI Buttons to the Panel. Call one *StartAgainButton* and the other *ExitGameButton*. Anchor them somewhere near the center of the Panel. Change the text of the buttons so that the *StartAgainButton* says 'Start Again' and the *ExitGameButton* says 'Exit Game'. At this point, your GameOverScreen should look something like the following figure:

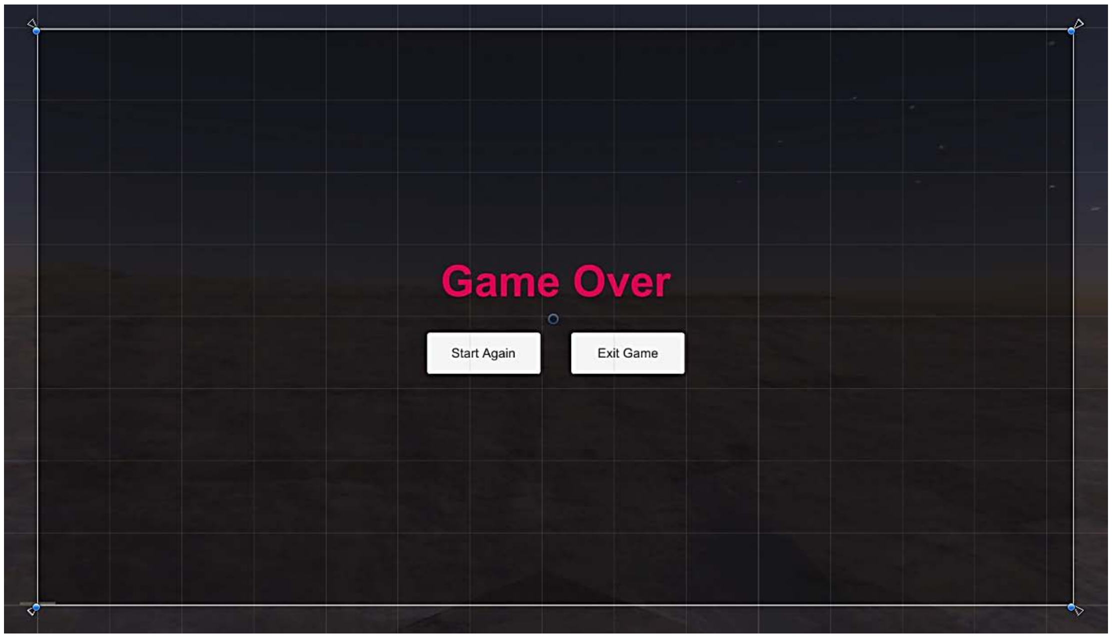


Figure 2: The Game Over Screen

Task 2 – Coding the Game Over Screen

1. Create a new C# Script and call it *GameOverScreen*. Attach it to the *GameOverScreen* Game Object. Open it **MonoDevelop**. At the top below the other `using` statements, add the following two namespaces:

```
using UnityEngine.UI;
using UnityEngine.SceneManagement;
```

   The `UnityEngine.UI` namespace you've seen before and which we need when working on UI stuff. The `UnityEngine.SceneManagement` namespace is exactly as it sounds – it contains the `SceneManager` class which manages which Scenes are currently loaded/active in the Game.

2. Add a `SerializeField` instance variable for the *CrossHair* Game Object:

```
[SerializeField] private Image crossHair;
```

3. Add an `Open()` method:

```
public void Open() {
    // pause game & turn off crosshair
    Time.timeScale = 0;
    crossHair.gameObject.SetActive (false);
    // turn on popup
    this.gameObject.SetActive (true);
    // broadcast Event
    Messenger.Broadcast(GameEvent.GAME_TIME_CHANGED);
    // Activate mouse
    Cursor.lockState = CursorLockMode.None;
}
```

If this code looks familiar, it should! It is the exact same code as the `Open()` method in the *OptionsPopup* Script. We pause the game, de-activate the *CrossHair*, activate the Game Object this Script is attached to (in this case, *GameOverScreen*), broadcast that the game has been paused, and free the mouse cursor. With this repeated code, we enter the realm of Software Design and I'm sure you have some questions:

Question: If this is the same code as the Open() method in OptionsPopup, why not just call that method?

Answer: That would be poor Software Design. We aren't opening the *OptionsPopup* so it would seem like a weird thing to do and would make your code hard to understand. It also makes the *OptionsPopup* and *GameOverScreen* code ***tightly coupled***. What if you want to make a change only to the OptonsPopup `Open()` method? Would you remember, or would another developer know, that you called it in *GameOverScreen*?

Question: I learned about Inheritance. Can't we just put this method in a parent class and make both this Script and the OptionsPopup Script children of the parent class and inherit the method?

Answer: Good Idea! But one big problem – C#, like Java, can only inherit from one parent class. If you look closely, we are already inheriting from `MonoBehaviour`.

Question: What about making a Popup class that inherits from MonoBehaviour and then have OptionsPopup and GameOverScreen inherit Popup?

Answer: This is getting more in the realm of possibility. This kind of structure gives birth to what are known as 'inheritance trees'. It can add a lot of complexity to your code and can introduce new problems. For example, let's say you have *OptionsPopup*, *GameOverScreen*, and *SettingsPopup* all inheriting *Popup*. What if you want to add a behaviour (method) that only *OptionsPopup* and *SettingsPopup* should be able to use? You can add it in *Popup*, which would save implementing it in both *OptionsPopup* and *SettingsPopup* but now it's available to *GameOverScreen* when it shouldn't be. So, now you'd have to create a new sub-class adding more complexity to your design.

Question: Well, I also learned about Composition. Could I make an OpenBehaviour *interface, which classes like PopupOpen and GameOverScreenOpen would implement, and then make these components of OptionsPopup, SettingsPopup, and GameOverScreen appropriately?*

Answer: Yes! In fact, this is called the ***Strategy Design Pattern***. *Design Patterns* are like blueprints for code that have already solved common software design problems. This isn't a course in Software Design and Engineering and you haven't learned Design Patterns yet, so I haven't mentioned it up to this point.

We are on the tipping point of it being worth it to re-factor our code to use the *Strategy Design Pattern* for our problem. If I needed just one more Popup, I'd probably do it. But, as this isn't a Software Design course, and we aren't going to add any more Popups, I wouldn't re-factor your code to use it. Leave the 'repeated' code in GameOverScreen's `Open()` and other methods. If you were to continue developing the GUI in the Game, implementing this Design Pattern would probably be a good idea.

4. Next, add a `Close()` method:

```
public void Close () {
    this.gameObject.SetActive (false);
}
```

5. Next, add an `IsActive ()` method:

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

This is the same method you added in *SettingsPopup* and *OptionsPopup*. It simply returns true or false whether the Game Object is active or not. This will be useful in the *UIController* Script.

6. Now, we need to code the methods for the Buttons. Start with the Exit Game Button:

```
public void OnExitGameButton() {
    Application.Quit ();
}
```

Same code as in *OptionsPopup*.

7. Next, the Start Again Button:

```
public void OnStartAgainButton () {
    // reload same scene
    SceneManager.LoadScene (0);
    Time.timeScale = 1;
    Messenger.Broadcast(GameEvent.GAME_TIME_CHANGED);

}
```

`SceneManager.LoadScene(0)` closes the current Scene, flushes all the Game Objects (and their Components, including Scripts) and then reloads the Scene at Build Index 0 (more on that later). We then un-pause the game and broadcast an event that we have done so.

8. This is everything we need in *GameOverScreen*. Save your Script. Back in the Unity Visual Editor, link your *CrossHair* Game Object with the *GameOverScreen* Object by dragging and dropping the *CrossHair* in the box in the Inspector Tab.

9. Next, define your Callbacks for the two buttons. With a button selected, drag and drop the *GameOverScreen* Game Object into the empty box in the Inspector Tab under *On Click ()* and choose the appropriate method.

10. Now we need to make some changes to a few other Scripts. Open the *UIController* Script and add a `SerializeField` instance variable for *GameOverScreen*:

```
[SerializeField] private GameOverScreen gameOverScreen;
```

11. In the `Start()` method, you need to make sure the *GameOverScreen* Game Object is not active when the game starts:

```
gameOverScreen.Close ();
```

12. In the `Update()` method, you need to make sure that the GameOverScreen (along with all other Popups) aren't active when the ESC key is pressed:

```
if (Input.GetKeyDown ("escape") &&
    !optionsPopup.IsActive() &&
    !settingsPopup.IsActive() &&
    !gameOverScreen.IsActive()) {
    // no popup is open, ok to open Options Popup
        optionsPopup.Open ();
}
```

13. So, how do we know when to open the *GameOverScreen*? That information will come from our Scene (backend) code when the Player's health reaches 0. We need to add a new Event. In *GameEvent*, add the following Event:

```
public const string PLAYER_DEAD = "PLAYER_DEAD";
```

14. Back in *UIController*, add a listener in the `Awake()` method:

```
Messenger.AddListener (GameEvent.PLAYER_DEAD, OnPlayerDead);
```

And remove it in `OnDestroy()`:

```
Messenger.RemoveListener (GameEvent.PLAYER_DEAD, OnPlayerDead);
```

15. Then, implement the callback:

```
private void OnPlayerDead() {
    gameOverScreen.Open ();
}
```

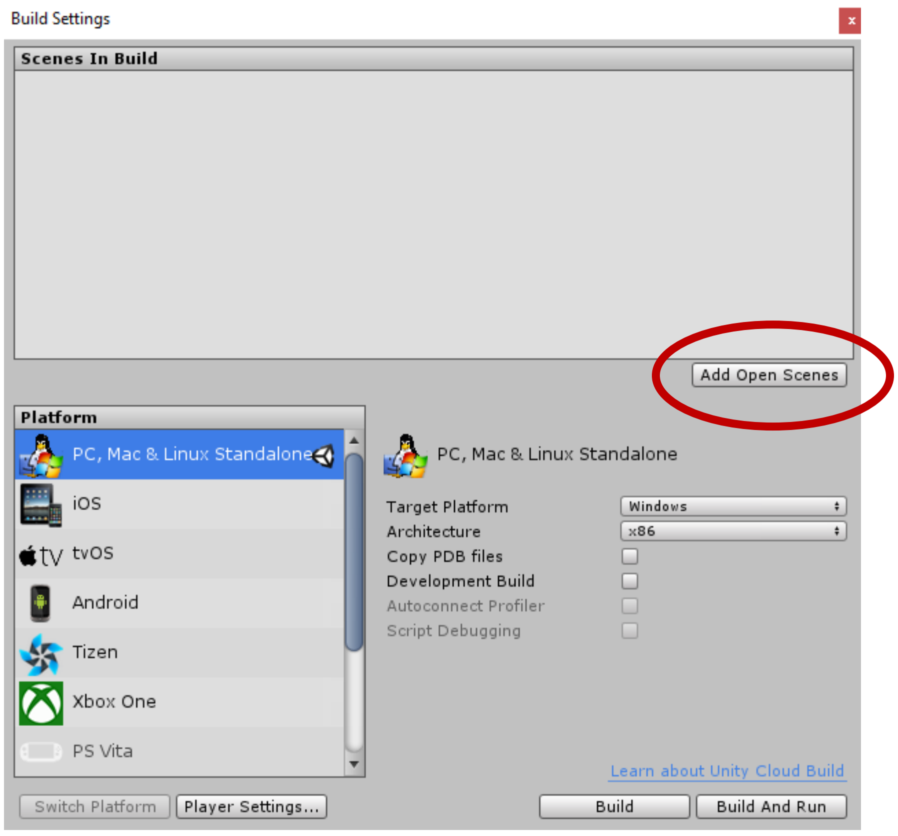
16. Now you need to Broadcast when the Player's health reaches 0. In the PlayerCharacter Script, comment out the Debug.Break() statement and add a Broadcast:

```
if (health == 0) {
    //Debug.Break();
    Messenger.Broadcast (GameEvent.PLAYER_DEAD);
}
```

17. Save all your Scripts. Play your Scene and ensure the Game Over Screen appears when the Player's Health is 0 (NOTE: The *Start Again* and *Exit Game* buttons won't work until the next Task is completed).

Task 3 – Deploying the Game

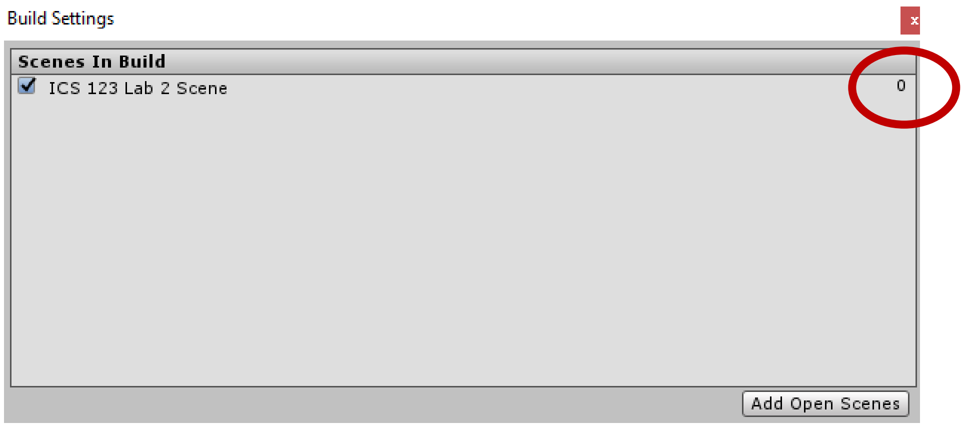
1. Now the exciting part! In the Unity Visual Editor, go up to the menu and select *File -> Build Settings….*



Target Platform: Windows
Architecture: x86
Copy PDB files
Development Build
Autoconnect Profiler
Script Debugging

Figure 3: The Build Settings Window

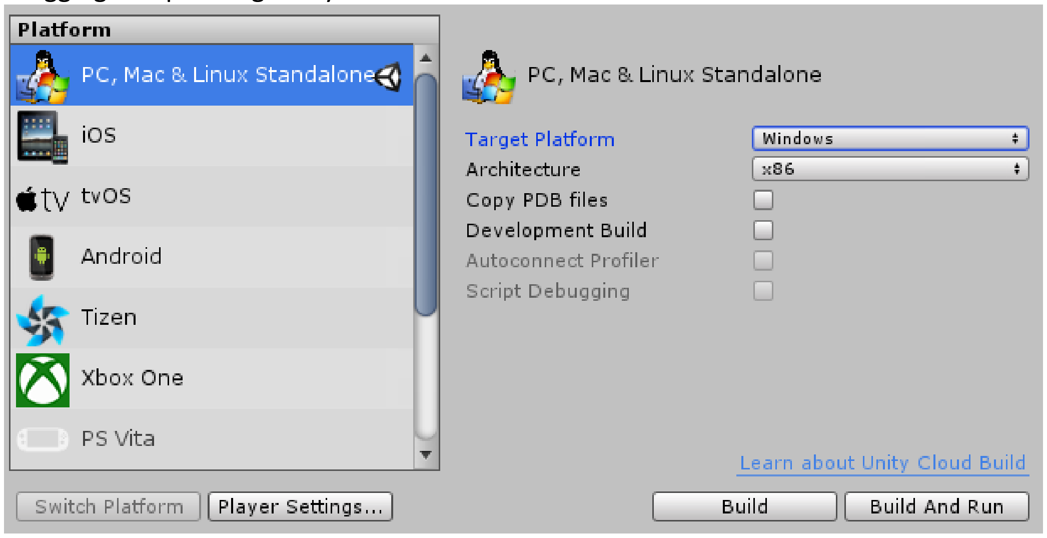
2. Click on the *Add Open Scenes* button, circled in the previous Figure. This should load your Scene into the Build window. With the box checked next to it, you should see a number appear on the far right (circled in the figure below). This is the Scene Index number. You can use that number to refer to the Scene. We did that in the last task for our `SceneManager.LoadScene(0)` method call.



☑ ICS 123 Lab 2 Scene 0

Figure 5: Build Settings Window with the Scene Loaded

3. In the bottom half of this Window, you have the *Target Platform Chooser*. For this example, the Target Platform will be for Windows. The Architecture drop-down allows you to choose 32-bit (x86) or 64-bit (x86_64). Leave it as x86. The other options are for debugging and profiling and you can leave those off.



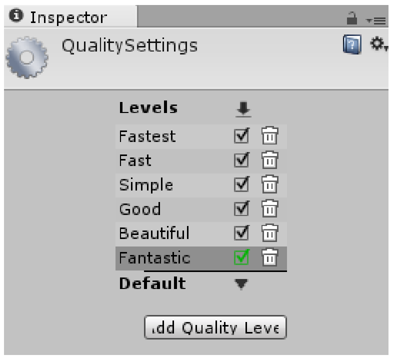
PC, Mac & Linux Standalone

Target Platform	Windows
Architecture	x86
Copy PDB files	☐
Development Build	☐
Autoconnect Profiler	☐
Script Debugging	☐

Figure 6: The Target Platform Chooser

4. The *Player Settings…* button will open a bunch of options in the Inspector Tab. For example, you can set the company name, the name of the game, and the game icon. There's a slew of other settings here too. You don't, however, need to change anything.

5. There's also the *Edit -> Project Settings… -> Quality settings*. At the top, you can pick one of the Quality Level Presets to improve the Graphics Quality of the Game at the cost of Performance (or vice versa). In the figure on the next page, the *Fantastic* quality preset is chosen to give maximum graphics quality to the game, as default.


Inspector

QualitySettings

Levels

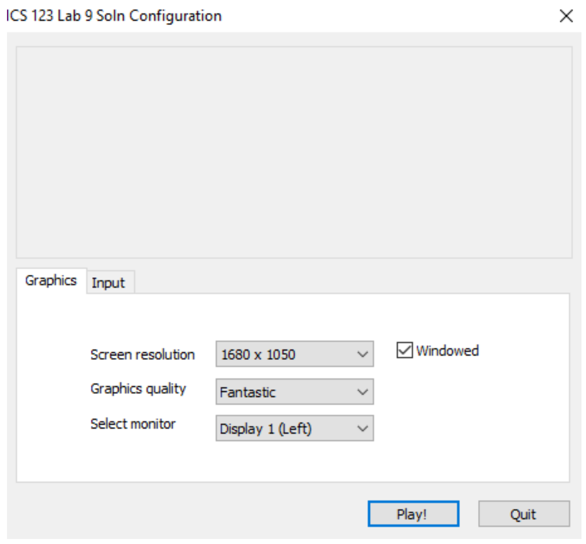
Fastest	✓
Fast	✓
Simple	✓
Good	✓
Beautiful	✓
Fantastic	✓

Default

Figure 7: The Fantastic Quality Level

6. Back in the Build Settings Window, you are ready to Build and Run your Game. Hit the *Build and Run* button. Choose where you want to save the stand-alone executable. Once you hit OK, Unity will build the executable and launch the game.

7. When the game is launched, Unity opens a Configuration Window. Here, you can change the resolution of the game, whether it should run in a window or full-screen and the Quality level. You can even change the Inputs! Once you have made your selections, hit Play!


ICS 123 Lab 9 Soln Configuration

Screen resolution: 1680 x 1050 [✓ Windowed]

Graphics quality: Fantastic

Select monitor: Display 1 (Left)

Figure 8: Game Config Window

8. The *Start Again* and *Exit Game* buttons should now work!

Submission:

Demo to the Lab Instructor:
THAT EVERYTHING WORKS!!! Especially that you can Start the Scene again and also Exit the Game

- The mark for this will be part of your project mark.