Setting up Game State with UI

# Overview

In this walkthrough we will explore the basics of Game State and use UI to help manage our scene navigation inside of Unity.

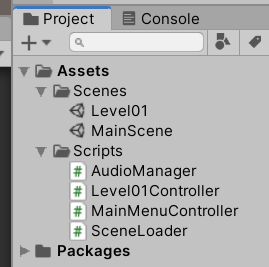
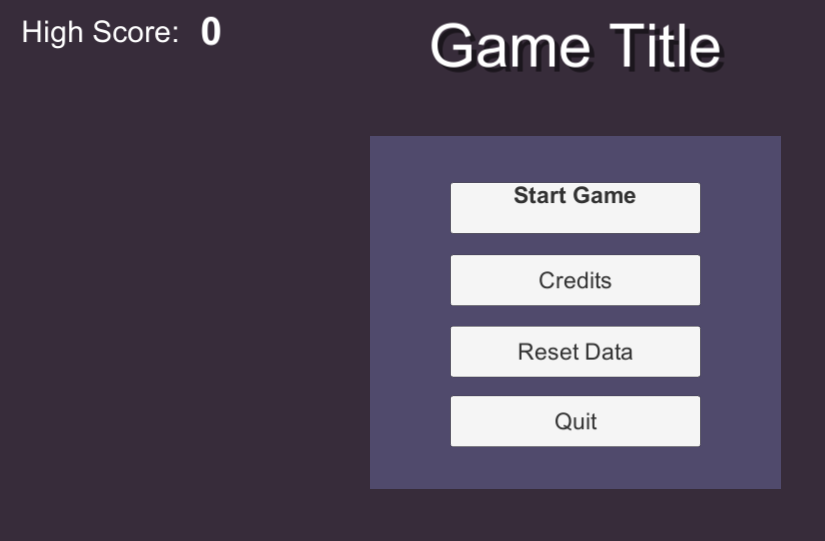
By the end you will have the following:

**AudioManager.cs** – Travels between scenes playing songs (and audio effects) when called

**MainMenuController.cs** – Controls menu and UI functionality in the MainMenu level

**Level01Controller.cs** – Controls UI and game actions inside of the Level01 level

**SceneLoader.cs** – Contains functions that can hook into buttons that allow level loading



We will also cover a variety of topics like GameState, UI, Button events and Saving data. First, let’s begin by setting up our scenes.

# Scene Setup

Once you start building a game that has more complexity than a simple character that navigates obstacles, you need to start thinking about how to segment your data into larger chunks. If you’re at all familiar with video games, you’d likely already recognize the terminology for ‘Levels’, and the idea of blocking out sections of your game into levels (or as Unity calls them, “Scenes”). We’ll start there.

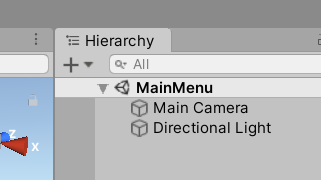
## Game States and Scenes

First let’s consider our game on a macro level. In this walkthrough I’m going to assume 2 major game-states: Menu and Game. You could definitely break Menu up into smaller states (and normally you would), but for simplicity we’re going to keep it higher level. We want our Menu scene to contain everything related to our title screen and game menu navigation before the game starts, and then our Level01 to contain everything related to our ‘in-game’ state.

If we wanted to break this up further, we could say that the “MainMenu” scene should contain several substates, like SettingsState, QuitState, CreditsState, etc. all contained within the same state. You could definitely do this if you want (look into Enums, or if you want further complexity, Finite State Machine pattern). For now, we’ll just simulate different panel states within the same scene by turning on/off different panels.

## Create your Scenes

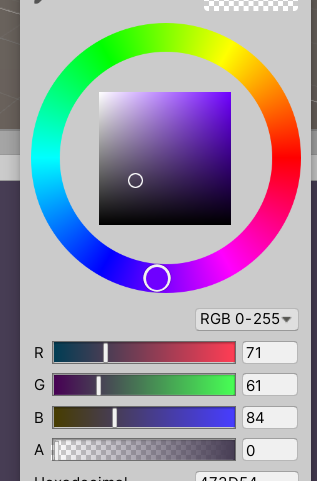
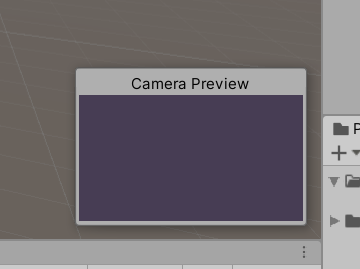
First, **begin from an empty scene (**File -> New Scene) and save your scene as **MainMenu**.



Select the Camera, and change the background color by doing the following:

* Camera ClearFlags – Solid Color
* Choose a color

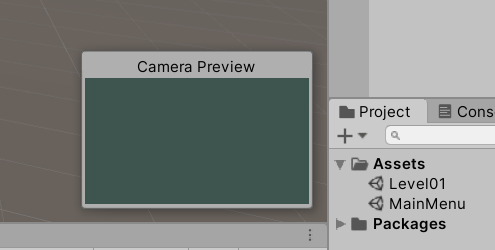
I recommend you don’t go full saturation, no matter what color you choose. So far, you should have an empty scene with a background color like so:

Save your changes. Next, let’s create our GameScene.

**Save your current scene as a new scene** by doing File -> Save As -> (name it “**Level01**”). This will duplicate everything in the current scene into a new scene named Level01 and open it.

Change the background color on this one to be something different and distinguishable. This is mine:

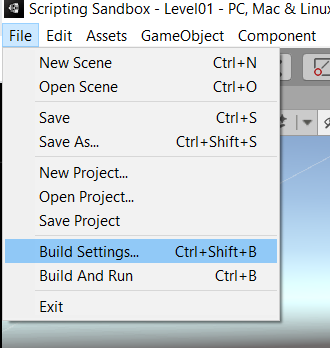


Save your changes. So far, you should be able to tell which scene you’re in by the background color you’ve assigned it. Feel free to double click each scene file to open them up and ensure they’re both different and correct.

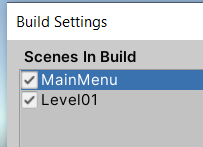
## Adding Scenes to the Build

Lastly, we need to add the scenes to our Build, to make sure that they get exported properly when we export out our executable.

**Click on File -> Build Settings** to bring up a window that contains all of our levels set to export.



If this is a new Project your window should be empty. Either way, make sure you **drag and drop your 2 scenes into the Build Window**. The scene listed first (index 0) is the scene that will open up when the executable loads. Your MainMenu should be the first thing that loads.



Now let’s start to create some UI Panels in our MainMenu scene.

# Creating the MainMenu UI

Open up your MainMenu level. First, let’s create the root menu that contains all of our primary navigation buttons. This will be the first thing the player sees when the load up the game.

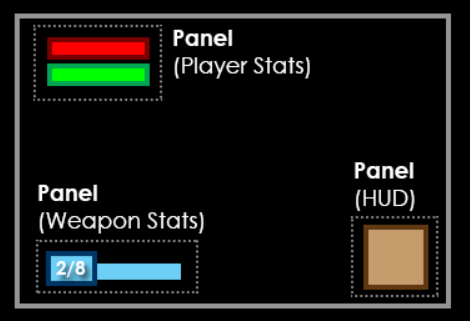
## Canvas, Panels, Images

First let’s define a few things:

Canvas – Holds all UI elements as child objects and contains behavior that sends the UI to the camera

Panel – An ‘island’ of UI elements that can be positioned on the screen as a grouping object

Image – Pixels/imagery that can be assigned in a x-y space.



The basic structure is:

Canvas

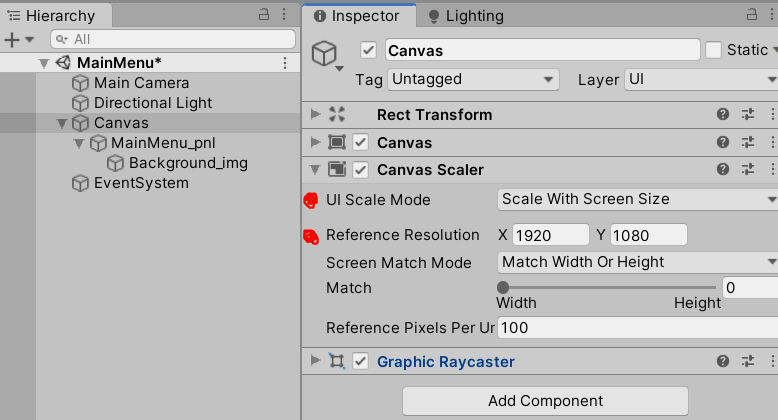
* Panel
  + Image, text, etc.
* Panel
  + Image, text, etc.

*Note: Technically, in Unity best practices is to treat a Canvas AS a panel, and resize/position them as needed using multiple as smaller ‘islands’ of information. This has to do with performance and rendering, but may be too advanced for now. Until you have more experience with UI, it’s fine to just use a single Canvas and define whatever panels you need on them. Did you know there’s not really a ‘class definition’ of a Panel in Unity? If you create one it’s just a preset Image with partial transparency. They just call it ‘Panel’ to label the weird preset it creates.’*

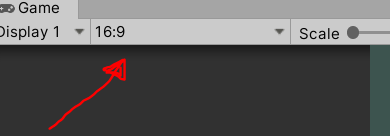
## Create your MainMenu Canvas

Make sure you’re inside of your MainMenu scene and create a new Canvas by doing the following:

* Right Click in scene -> UI -> Canvas (it’s possible this could change in future versions of Unity)
* Select the Canvas to get access to the Canvas Scaler component. On the Canvas Scaler do the following:
  + UI Scale Mode – Scale with Screen Size
  + Reference Resolution – 1920 x 1080



Next, we need to make sure our canvas scales properly in our Game Window view. On your Game Window, make sure you change the Aspect Ratio to 16:9 (or optionally, 1920 x 1080).



This will letterbox your game view so that it always shows you accurately a view of 16:9 proportions. We will use this to accurately view our UI as we make it.

If you’ve ever struggled with weird scaling issues with Unity’s UI system, the steps in this section will likely solve 90% of those problems (the other 10% is anchoring).

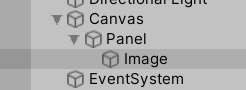
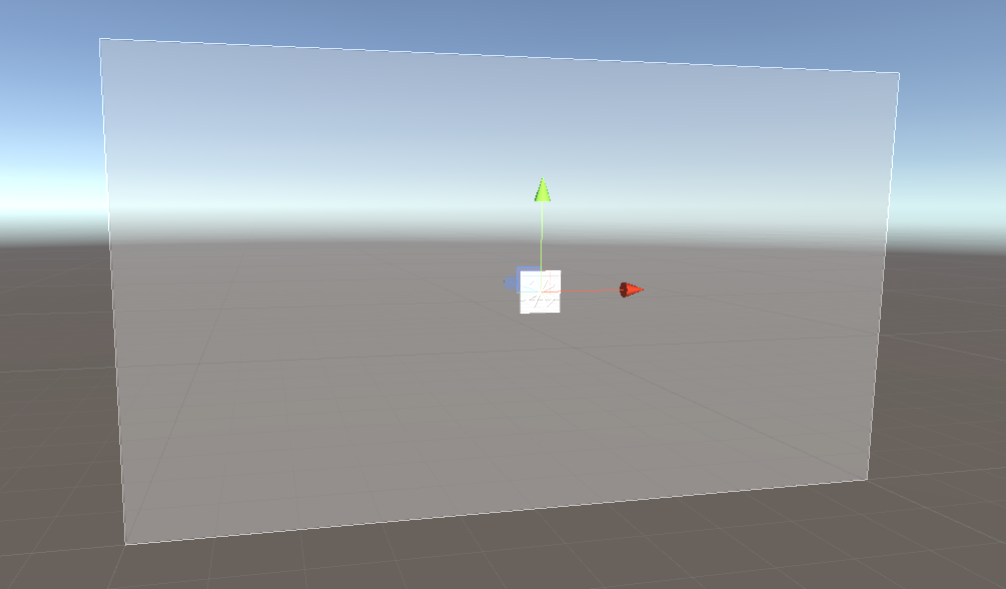
## Create your MainMenu Panel

Now let’s start putting things inside of our Canvas. **Right click the Canvas and add a panel** (UI -> Panel). This will create a new GameObject named ‘Panel’ that contains an Image component with partial transparency.

Note: In this example we’re going to use the ‘1 Canvas per scene’ method, which is arguably not the most optimized way to do it. Technically, each Canvas should be an ‘island’, similar to a panel, with UI elements within. We’re using the 1 Canvas per scene method for now to minimize scaling issues and get you comfortable with how UI works before you add complexity. I felt it was worth mentioning.

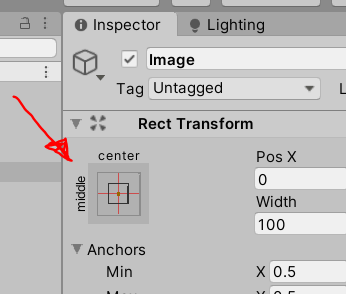
On your new Panel, **create a new Image UI GameObject as a child object** (right click the panel -> UI -> Image).

Right now you should have this:

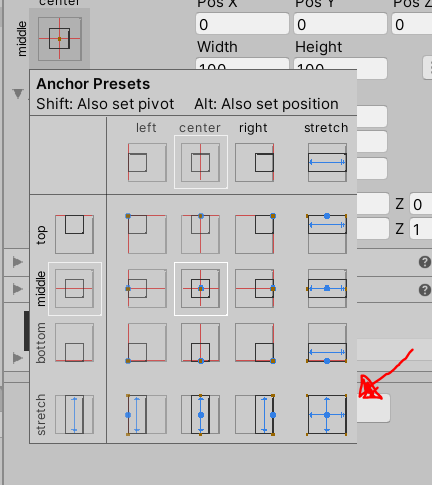
 

It looks extremely weird, but I want to explain what’s going on. We have a Panel that’s the size of our entire Screen (aka Camera) that has an image component attached, with a partial transparency. This is that light gray background color you see. Then, we created an Image as a separate child object inside of our panel as a 100 x 100 (default) image object. What we want is the Image to be the background of our panel, and for the panel to NOT have any Image information.

First, make the image the same size as the panel by using the anchor. Select the Image gameObject, and look at the Rect Transform. **Click the anchor thumbnail** (looks like a crosshair in the upper left) to open your anchor options.

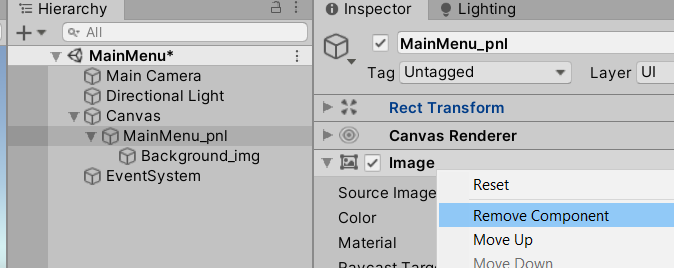


Your anchor options determine how a UI element scales according to its parent. In this case we want the background image to always scale perfectly with the size of our panel. To do this (with the anchor options open) **hold Shift + Alt and click the bottom right anchor option**.

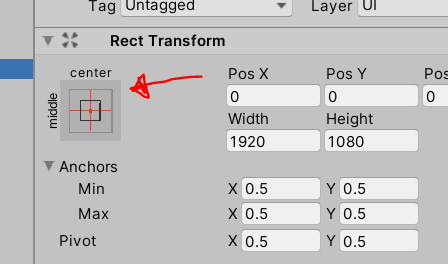


This will tell this image component to always match the scale of its parent.

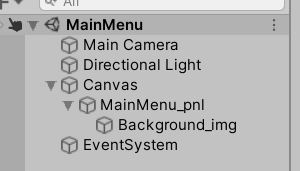
Next, we need to remove the Image component from our Panel (it’s unnecessary at this point). Since we’re defining a panel as a grouping element, we really don’t need it to have an associated visual, and our background image is already serving this purpose. **Select the Panel and remove its Image component** (right click component -> Remove)



Also, make sure that our Panel anchor is ‘centered’ so that we can position it in space. **Click the Anchor on the Panel, and click the Middle/Center option** (looks like a crosshair):

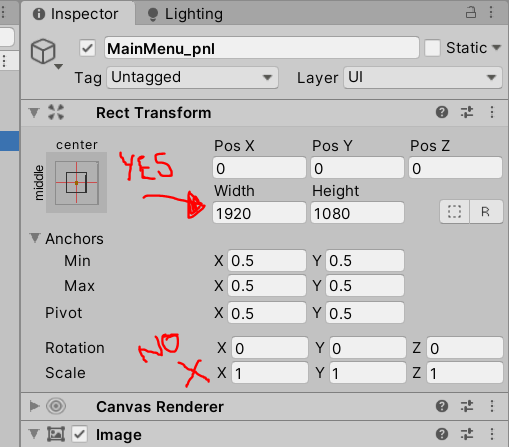


Lastly, let’s make sure our UI objects are named properly. Rename your Image “Background\_img” and your Panel “MainMenu\_pnl”. I prefer using a suffix convention to quickly communicate the ‘type’ of UI object, and the beginning portion to describe the name of the object. Use whatever convention you wish, as working with UI is complicated.

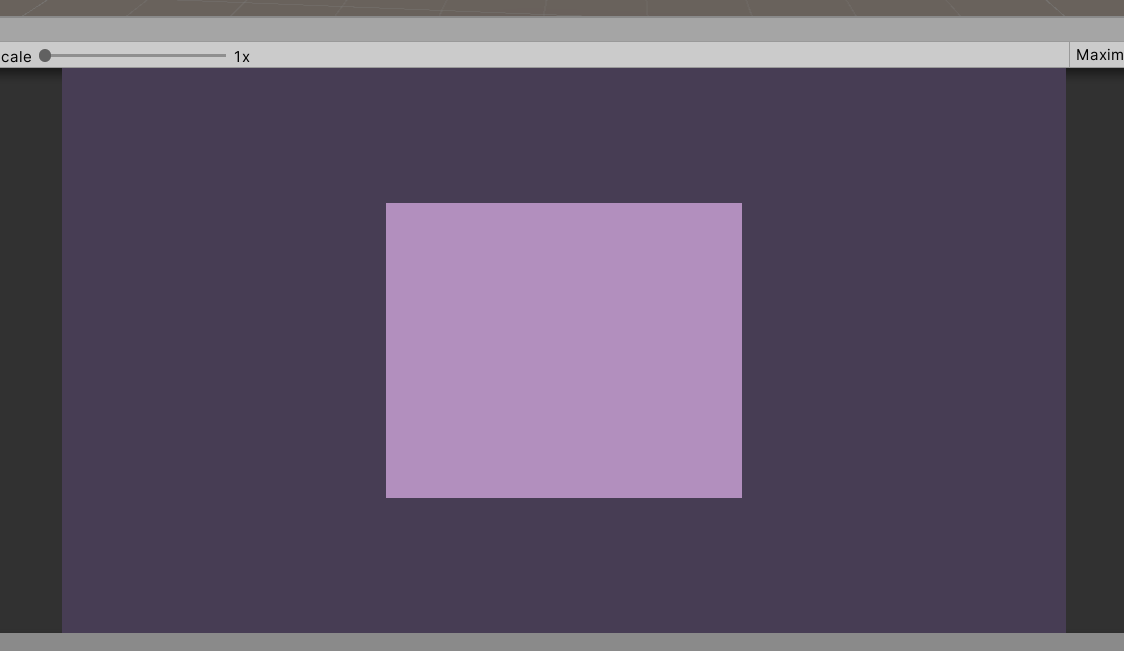
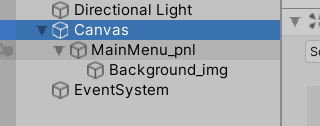


Lastly, let’s resize our panel to something a bit closer to what we want.

*It’s EXTREMELY important that when you’re working with UI you keep track of if you’re ‘scaling’ a UI element versus ‘resizing’ it. In almost all cases, we want to change the ‘width and height’ of an object, NOT scale. By retaining a scale of 1,1,1 and having our resizing done with Width and Height, we give ourselves the option to do procedural UI animation later on by animating the scale. Imagine that a panel could ‘pop in’ to existence by animating scale from 0 – 1 over time. We want to retain that ability.*



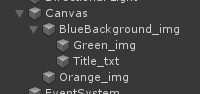
Select your MainMenu\_pnl, look in Rect Transform, and **resize the Width and Height** to be something a bit closer to the screenshot below. You can do this by either dragging the Width/Height parameter in the Inspector, or by using the ‘UI Scale Tool’ (hotkey T). Also, **change the Background\_img – Image Color** to be something a bit more interesting, whatever works with your background.

Whew. That’s a lot of setup. But, if we did everything right, the rest is easy. All we need to do is add UI elements into our Panel (or other parts of our Canvas).

## Caution with Rendering Order

When layering several UI elements together, you may want to choose what gets rendered on top, and what gets rendered on bottom. In Unity, this works on a bottom-up system, meaning that the child object gets rendered first, then the parent, etc. Additionally, the last child in a hierarchy gets rendered first, and then the sibling children get rendered behind.

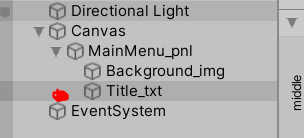
 

You can see the order of things rendered here. Our background is in the back, because it is the first child object. Then all the Background child objects are rendered on top of it. The text is rendered in front of the Green image, because it is below it in the hierarchy. Lastly, the Orange block is rendered on top of everything else because it is the bottom-most child object.

Another way to think about it is “Reverse Photoshop Layer Order”. This is one of the few times in Unity where your position in the hierarchy does actually matter. Be intentional with how you organize your UI panels, especially if you’re doing any sort of spawning UI things in the scene.

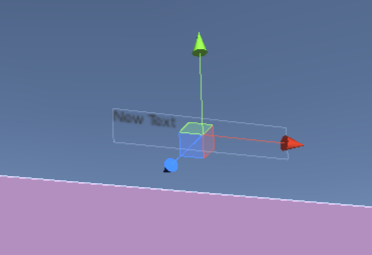
## Creating the Title Text

Let’s create some text to give our Game a Title. Select the Panel and create a Text GameObject as a child object of panel (Right-Click panel, UI, Text). Rename this GameObject “Title\_txt”.



*NOTE: You may see a UI object named TextMeshPro. Don’t use this for now! It’s actually much better than the standard UI Text, but at the cost of complexity for this example. In a real world scenario you would almost always use a TextMeshPro asset instead of a Text asset, but it will change the code in our examples and cause additional package importing which we’re just not ready for yet. Add it on your ToDo list to learn TextMeshPro after you have become familiar with the basics after this walkthrough.*

First, use the translate tool (Hotkey W) to **reposition the Title Text above our background**. It’s extremely small and you can barely see it! In order to see our letters, we need to talk about how sizing works.



## Styling the Title Text

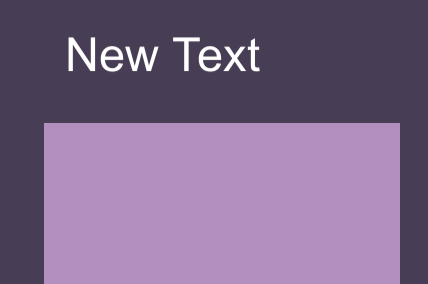
Similar to Width/Height vs Scale, we have an additional option for resizing text: Font Size. Increasing the Font size will give us larger text, without scaling the blurry pixels.

**Change the FontSize on your Text component** (on your Title\_txt object) to 90.

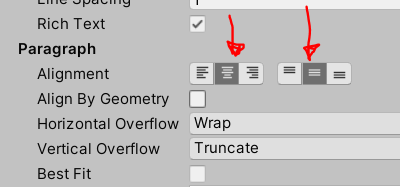
You’ll notice it disappeared! What’s happening??

The font size has increased our text to be beyond the constraints of the ‘size’ of our Text GameObject, so it is no longer being rendered. **Increase your Width x Height** on your Title\_txt gameObject Rect Transform to 600 x 150.

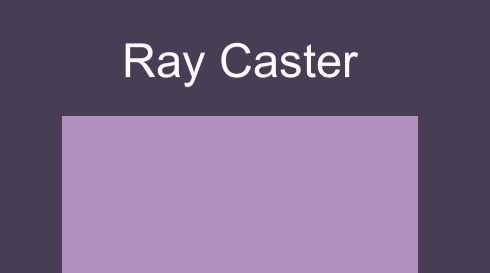
You should now see your text again, but only barely. **Change the Text Color** to something that will contrast with the background (in my case, White).



We’re getting closed, but now are text is off-center. Let’s look under the Alignment tools in our Text component, and **center our text** both Vertically and Horizontally.



Now let’s **give our text new characters**, to represent the actual title of our game. Look for the Text field on our Text Component, and give it new characters to represent the title of your Prototype.

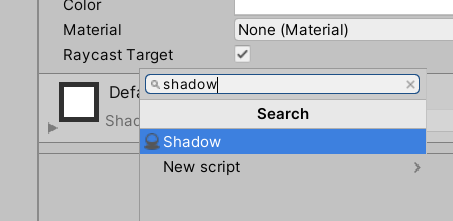


*The great thing about the Text field is that we can change the characters at runtime, through code. This is our primary of sending Data into the display -> We just replace a TextComponent’s .text field with our data. We’ll do this a bit later in this tutorial.*

Now let’s give our text some weight. **Make your Text Bold** by using the ‘Font Style’ dropdown.



Finally, let’s make it spicy with a drop Shadow. Add a new component (with the Add Component button at the bottom of the Inspector) and search for ‘Shadow’, and **add the Shadow component** to your Text object.



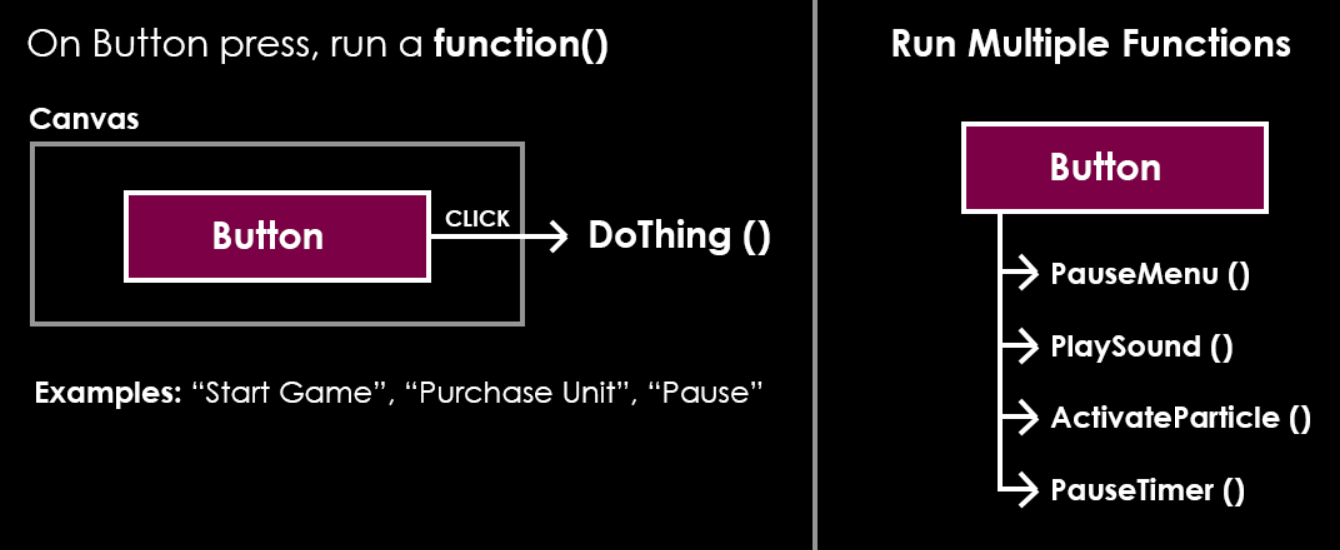
On the Shadow Component change the size of the Drop Shadow to 8, -8. These numbers are related to the pixel dimension of your scene, so in our case we needed to bump them up a bit.



We went through a few options on how to style text. From this point onwards, I’m going to assume you can use these options to style the text however you wish, and I won’t review it. Keep in mind… readable and ‘clean’ text is almost always better than fancy and convoluted. When it doubt, keep it simple (ESPECIALLY for Body text. Title Text can generally be fancier).

# UI Buttons

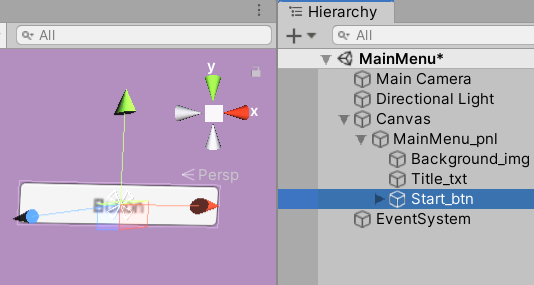
In UI, a “Button” is primarily just an Image that can do something when ‘clicked’. We want to be able to define an ‘OnClick’ event, then hook in for something to happen when the button is clicked. In most cases Text is also associated with a button.



Let’s make a few buttons for our MainMenu.

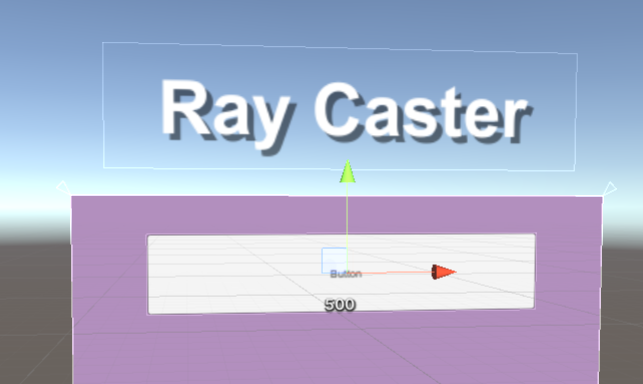
## Creating the Start Button

Now let’s make our Menu do something. **Add a Button to the MainMenu\_pnl** (right click panel -> UI -> Button). **Rename it “Start\_btn”**.

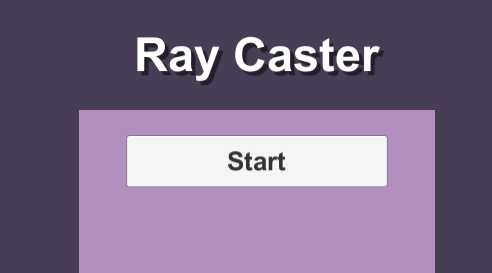


You’ll notice that the button is extremely small. Just like our text, this is because we put our reference size as 1920 x 1080. If you had gone smaller (1280 x 720) things might appear bigger. It’s all relative, and the goal is to get it as close to possible as the final device size, while still allowing for some dynamic resizing.

Just like our other UI objects, we will use the Width and Height to resize, NOT the scale. **Change the Width x Height of the button** so that it appears similar to my screenshot below (my button was 500 x 100).

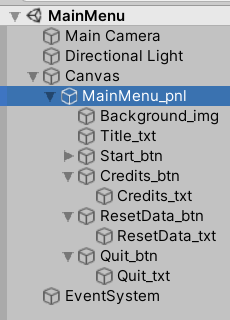


You’ll also notice that the Text on the button is extremely small. The text is actually a child object of the button, so we’ll need to expand the child object and resize our text, just like we did before. I’d also recommend **renaming the button text gameObject to “Start\_txt”.** Make sure to **increase the text Font Size**, as it’s already anchored to capable of stretching to the size of the button parent.



## Creating the other Menu Buttons

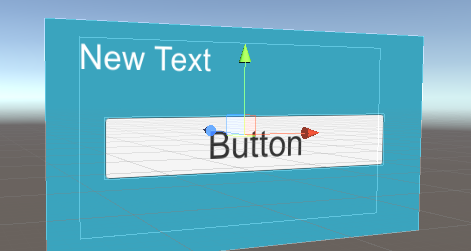
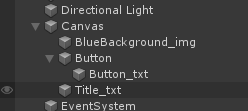
**Duplicate the Start Button a few times and position downwards**, so that you have 4 total buttons: Start, Credits, ResetData, and Quit. Resize/Style as necessary. Rename your Buttons appropriately.

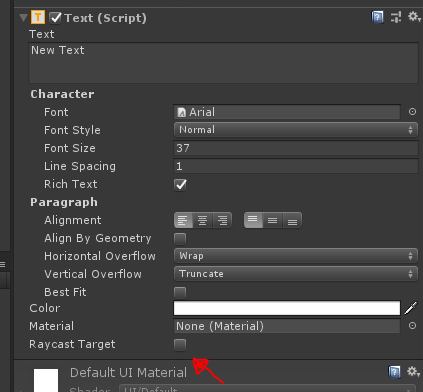
## Caution - Raycast Blocking

One thing you may notice is that you’ll occasionally get buttons that don’t ‘press’ for some reason. In order for a button to activate, it needs a Raycast event, which basically simulates a ‘press’ if our mouse is on top of a thing and clicks. In Unity, many of the UI objects can ‘block’ rays unless you tell them not to. For example, a default Text or Image object will both block rays by default.

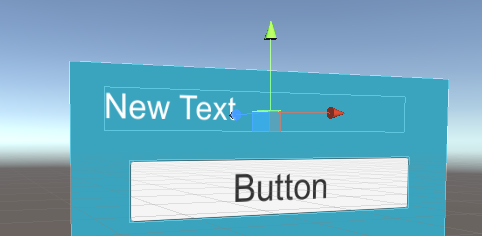
Consider this example:

In this example, we would NOT get a button press when we try. The problem is the Title\_txt is way too big, and the ‘size’ of the object is covering the button, even though we made the text fairly small. The UI Object is shielding anything underneath it from Raycasts. So how do we allow raycast to pass through in this situation?



Look for the setting called “Raycast Target” on Text and Images and disable the setting if you don’t need it. It’s actually more optimized this way too! Make sure you leave this option ON on buttons, but there’s plenty of times you may want 1 panel to ‘block’ another. It’s also best practce to keep the size of your UI image objects proper and not overlap into other UI objects. Overlapping UI elements is the fastest way to create weird Raycast Blocking.



Resizing our text object avoids the overlap and is more clean. Ideally you want to disable Raycasts on things that don’t need it AND avoid overlapping objects (if relevant).

Now that we have our buttons set up, let’s do something!

# Creating the Credits Menu

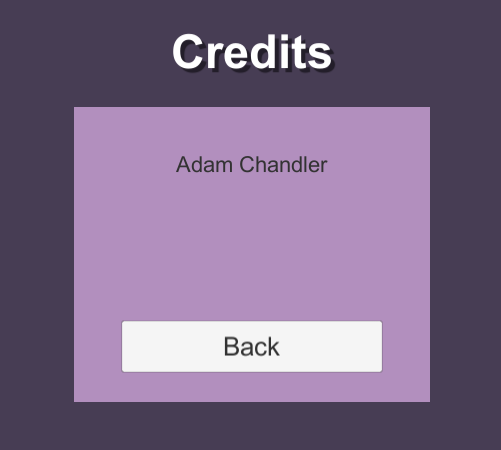
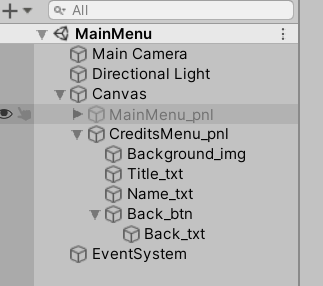
Let’s create a second menu, the Credits menu, and hook it into navigation using our buttons to navigate. Before we add the behavior, we need to actually create the menu.

## Building the Credits Menu

Start the infrastructure for your Credits Menu by doing the following steps (careful not to skip any!):

* Duplicate the MainMenu\_pnl (Ctrl + D)
* Disable the original MainMenu\_pnl, so that we’re only looking at the new one
* Rename the new Panel gameObject “CreditsMenu\_pnl”
* Delete the top 3 buttons (Start, Credits, Reset Data)
* Add a Text GameObject where the previous 3 top buttons were and add your name in the text. Resize/style as necessary
* Rename the Button (and Button Display Text) at the bottom “Back”.
* Change the Title\_txt to display “Credits”.

Your UI should look similar to this:

Our goal here is to be able to have the following functionality on our buttons:

MainMenu – Credits (button) -> Disable MainMenu, Enable CreditsMenu

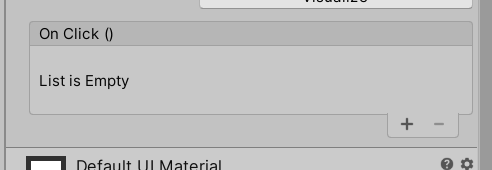
Credits – Back (button) -> Disable CreditsMenu, Enable MainMenu

If we can control the panels like this, then we should be able to navigate through several panels to progress our game.

*Note: Any of you experienced programmers may recognize that doing this puts the ‘state of the game’ in the active panels, without tracking it anywhere else. With this setup we have no way of knowing where we are in our Panel Layout! Longterm we want to move into FiniteStateMachines to track the substates (MainMenu, Credits, etc.). For an easy in-between, you can consider defining each substate as an Enum, and then call something like MenuState.Main, or MenuState.Credits, etc. whenever you switch. That way if we ever wanted to know what state we’re currently in, we can look back and get the CurrentMenuState. This is too complex for the demo, but I did want to mention it.*

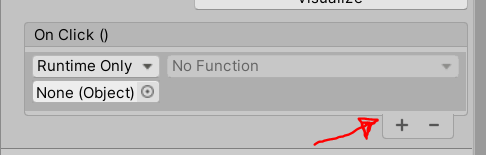
# Adding Navigation Behavior to the Buttons

Let’s use our buttons to navigate by enabling and disabling the panel. We’re going to do this through a prebuilt Button event named OnClick. Technically this is a ‘UnityEvent’ (a type of event defined by Unity that gives us flexible Inspector options), but the concept of an event is always the same -> If we call the ‘event’ (OnClick) properly, we can assign functions to run in response, when the event is triggered. In other words: When OnClick happens, do these things.



**Disable the Credits Menu, and Enable the MainMenu** (if you have not already done so).

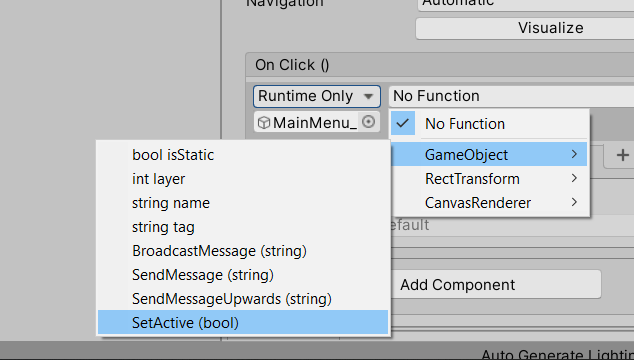
Select the Credits\_btn object (now you know why we were naming these things), and **Click the + button on the OnClick()** portion of the Inspector. This will add a new ‘listener’ to our event, which we can then assign a function to. This function will get called when we click the button.



*Right now, it’s empty. We need to give it a GameObject, and once we do we can access any of the components on the GameObject. From there we can access any of the public functions inside of that component and run it! Let’s start with something easy first.*

**Drag the ‘MainMenu\_pnl’ GameObject into the new empty slot** we have created in OnClick(). From here you’ll see you can click the Function dropdown in the upper left of the box, and select a component. We should see GameObject, RectTransform and CanvasRenderer. Technically, if you had any functions you wanted to run in any of those components, you could assign it and it would happen when you click the button!

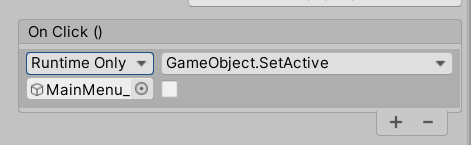
Select the GameObject component, and select SetActive.



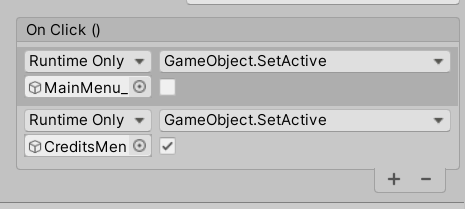
This is just like calling gameObject.SetActive(true/false) inside of our scripts, but now we’re calling it through visual editors and assigning it to a button.

Once you have SetActive assigned, you’ll notice in the OnClick() box you get a checkbox. Depending on the function, you may get an input field for string characters (string input), numbers (int/float input), or checkbox (boolean input).

Since SetActive takes a true or false, we can use the Checkbox to determine if we want to call SetActive(true) or SetActive(false). In this case, we want to turn OFF the MainMenu\_pnl**, so make sure it is not checked**, to tell it SetActive(false) on the MainMenu\_pnl.



Additionally, on this button press, we want to turn the CreditsPanel\_pnl on. Do the same thing on the Credits\_btn; **+ to add a new listener, drag in the CreditsMenu\_pnl, gameObject -> SetActive (true)**.



## Events Listeners run in sequence

**CAUTION**: *Technically this will work because of the way SetActive calls (it finishes everything else on the object before disabling), however this is extremely bad practice. Look at what we’re doing here:*

*FIRST, we’re disabling the current MainMenu (and all child objects underneath).*

*THEN, we’re enabling a different menu.*

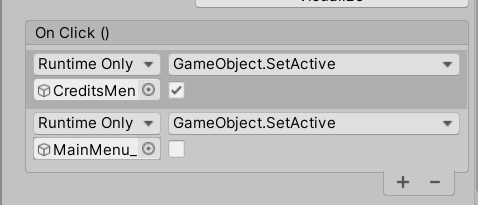
*The problem is that event calls like this do not happen asynchronously. This means, it will do each event, one by one, starting from the top and going downwards. In our setup, we’re actually disabling the button when the MainMenu\_pnl disables (which this OnClick event is attached to), and then trying to do other events. It’s not wise to attempt to call events on a Disabled gameObject. The better way to do this would be:*

*FIRST, enable the Credits Menu*

*THEN, disable this Main Menu*

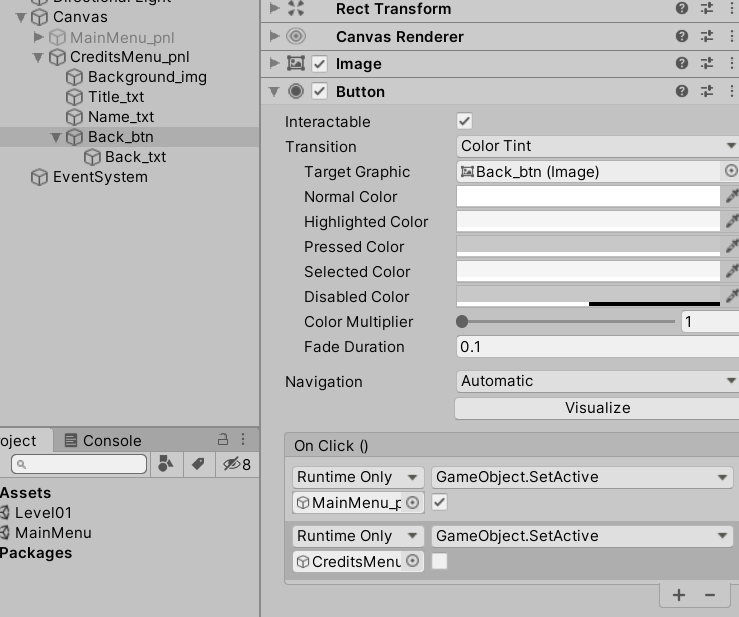
Again… technically it will work. But you need to be very aware when you are disabling/enabling things in sequence. Disabling the current object this script is running on should always be the last thing you do, if you’re going to do it.

I’m going to recommend you rehook up these 2 events (hey it’s good practice!) to the proper order: Credits enable first, mainMenu disable second. Remove the 2 events (with the minus button) and do so.



**Hit the Play button to test**. When you click the Credits button, it should enable the Credits panel and Disable the MainMenu.

But now we’re stuck! Let’s **add functionality to the Back\_btn to navigate back to the MainMenu** and complete the loop.



**Test it one more time** and make sure you complete the loop!

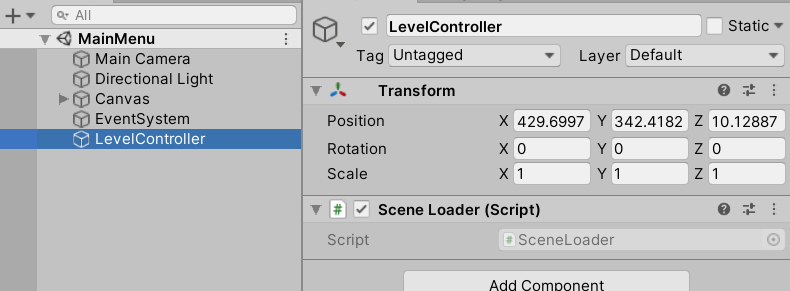
If you’ve ever thought UI wasn’t too bad… you’re half right. It takes an extremely long time, and we only barely touched the surface of what you can do. Now let’s talk about how to load our other scene through our UI buttons… it’s time to Start the game!

# Loading Level 01

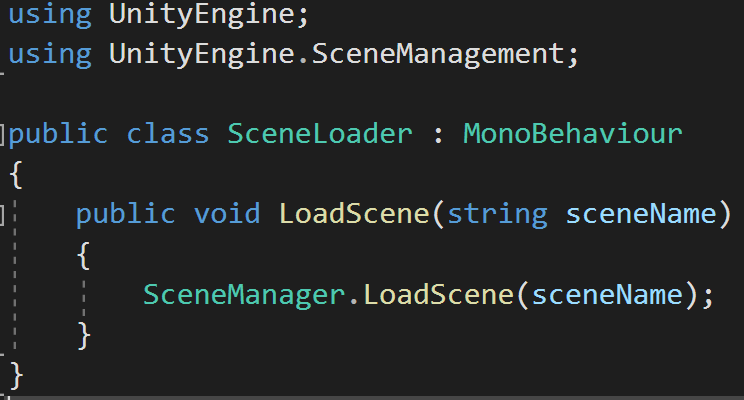
In order to Load a level we need to call a LoadScene() function and give it either a string reference (the name of the Level) or an index (which number in the Build order in Build Settings) for the level we want to load.

If you look at Unity’s API, you’ll notice that they already have a prebuilt static function that will Load the Level of the name you give it. The problem is… in order to call it on a button press, it needs to exist on a script attached to something in the level (an instance). Let’s create a custom method that calls this, so that we can attach LoadScene() behavior to a button.

**Create a new script named SceneLoader**. **Create a new GameObject called LevelController**, and **attach our SceneLoader.cs script** to it.



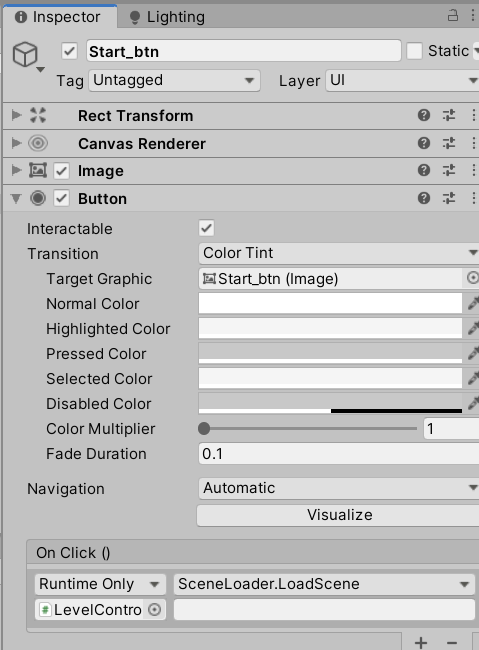
Add the following code to the SceneLoader script:



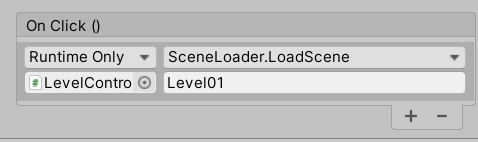
*Don’t forget to add the ‘using UnityEngine.SceneManagement’ at the top so that you can access the SceneManager functions.*

We want to call this public function from a button, and pass in the name of the Scene we want to load, and then the SceneManager will load it (provided that our spelling is correct and the scene exists in the Build Settings).

**Select your Start\_btn, and add an event OnClick**, just like we did before. Drag in your LevelController, access the SceneLoader component, and run the LoadScene() function.



Make sure to pass in the parameter, which should be the name of your level. Triple check your spelling! Make sure the level you’re calling is inside of the Build Settings!



We can use this method for making any public function we want and calling it from a Button press! Just keep in mind it won’t let you pass in any parameters that are more complicated than a simple int, float or string.

## Caution - Persistent References

Later in this assignment, we’re going to introduce the concept of Level Objects that persist through multiple scenes. Avoid hooking transient level objects into UI OnClick events (unless you’re doing this automatically through code). If you have it hooked into one scene, then it leaves to travel to another, then returns to the original scene… it’s lost its hooks.

This may sound odd right now, but just remember… if it’s hooked into a button it’s best practice to keep it in the scene. In other words, if you’re hooked into a method in a script on an object (like our SceneLoader), just make sure that that GameObject is always in the world and doesn’t really move around.

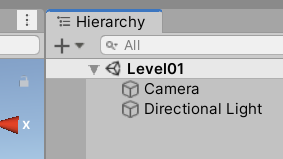
Test your Level and make sure your Start button loads your Level01 scene.

# Building Level01

We’ve made it into Level01 with our navigation, but we don’t have a way to return back to the MainMenu. Let’s add that functionality with a simple press of the Escape key.

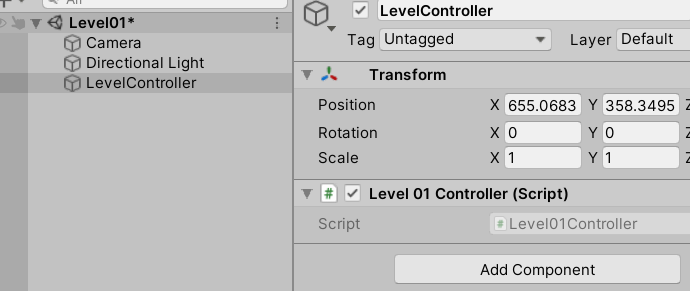
## Load MainMenu Functionality

Currently, your Level01 should still be empty.

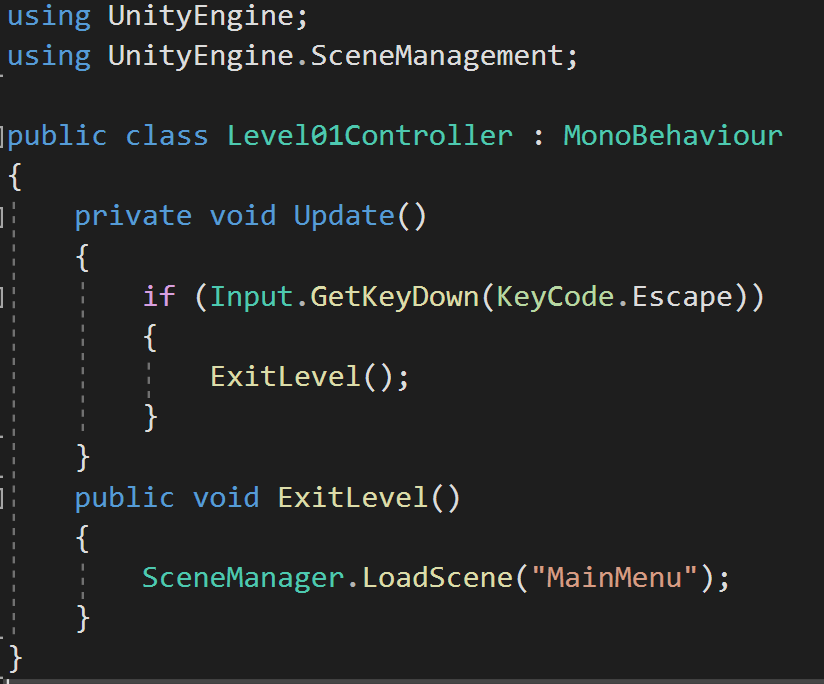


**Add a new GameObject named LevelController.**

**Create a new script named “Level01Controller” and attach it to the LevelController gameObject.**



Add the following code to the Level01Controller script.



This detects input, an if you press Escape it will load the “MainMenu” scene, assuming your scene has that same name.

*Note: Technically, you could reuse the previous SceneLoader script that we made before. If you wanted to do that, you could either make the function ‘static’ and call it directly, or you could attach the SceneLoader script onto this same GameObject and find it with a GetComponent search. It’s not really necessary since it’s only 1 line of code though, and we’re going to add more functionality to the ExitLevel() function later in this demo.*

If you Test your Level and hit the Play() button you’ll notice that we can navigate back. We’ve completed our full GameState loop! Now let’s move on to some extra functionality.

## Data Persistence

For our prototype, let’s define some behavior we may want. A few of the features we’re going to implement:

Scene Persistent Music Player

Tracking Current Score

Saving High Score

Loading High Score from last play-session

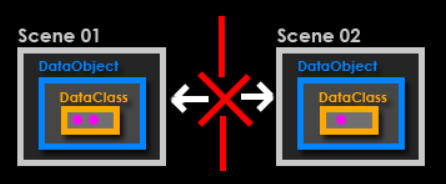
Display High Score on the Main Menu

In order to do some of these features, we need to consider how we are passing data around. Currently, we can load different scenes but in each of our scenes all of our gameObjects and scene Data is self-contained. Everything inside of a scene is Create()d on SceneLoad, and Destroy()d on SceneExit. We don’t really have an easy way, currently, of passing data between scenes.

First let’s look at different types of data persistence.

### Scene Persistence

All of our data exists only inside of our scene. This is ideal for most of our data, because there’s no real reason that our Level01 objects should know about our MainMenu objects, or vice versa.

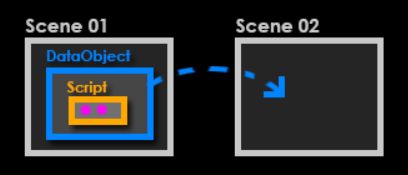


It’s possible a few objects (like maybe a player) might have to pass information through, but in general scene persistence is a good thing because it encapsulates the data:

**Encapsulation** – Keeping data internal and hidden from things that don’t need it

### Game Persistence

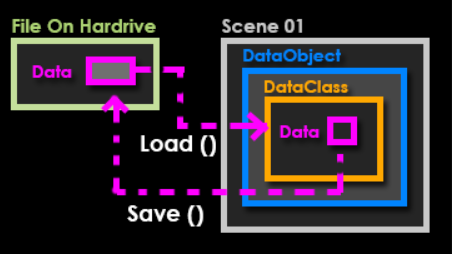
Sometimes, we may want objects that persist through multiple scenes. The most common example of this is a Music Manager. You may want the same song to continue through a Level Load and into the next level -> The music hiccup would be noticeable and jarring.



In cases like these, we can tell specific objects to ignore the ‘Destroy’ call from Unity when it is Destroying all the objects in the current scene, and persist inside of a Persistent level. Other objects can access objects in this ‘persistent’ level if needed, but it’s especially nice for objects that just need to continue to do things in the background (like a Music Player). If you’ve ever used the Singleton pattern, you need to be very aware of what’s happening with your persistent objects to help troubleshoot related bugs.

### File Persistence

Lastly, we may want objects that persist from previous game sessions, where we have exited the game and reopened. In this case, our data needs to exist on file (the computer’s hard drive) somewhere, so we can access it again later.



In a lot of scenarios, this is going to be the most ‘clean’ way to save data. There’s very few transient objects crossing paths, and it’s direct… Find the file, pull in the data. Save the data back onto file when you need to. The one thing to watch out for here is how you access this data, as multiple objects trying to load/save from a single file can get messy. If we want to Save a HighScore, we’ll need to save it somewhere onto the device we’re using.

With all of that in mind, let’s consider what we’ve done so far. By default, we have Scene Persistence (our level loading). We want to explore Game Persistence next, with an AudioManager that plays music. Lastly, we’ll explore File Persistence by creating Saving/Loading functionality with Unity’s PlayerPrefs system.

First, let’s create our AudioManager.

# AudioManager

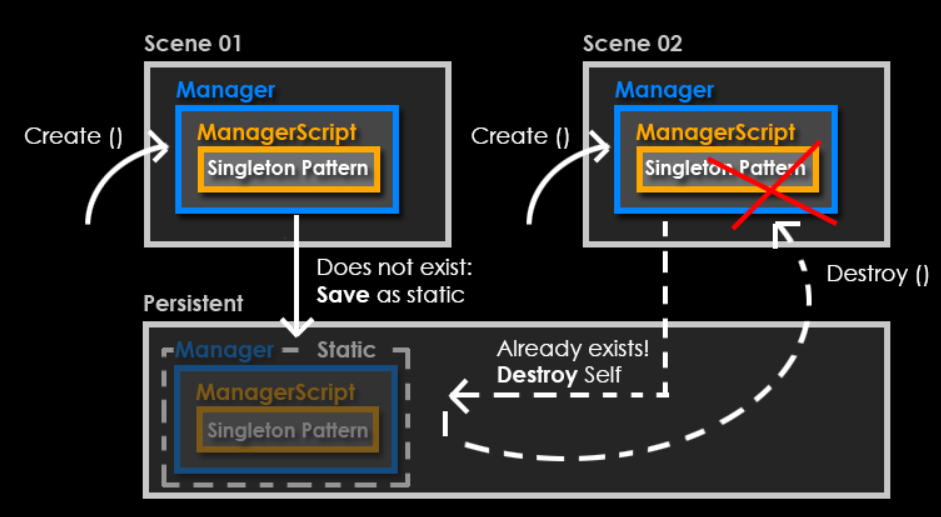
We want our AudioManager to be able to travel between scenes, so that we can Play() music without any interruptions. We also want to be able to easily access our AudioManager to send it new songs to Play(), no matter which scene we’re in. In order to meet both of these criteria, we should investigate a fairly controversial Design Pattern: The Singleton.

## Singleton Design Pattern

The Singleton Pattern enforces 2 things: Only 1 of this thing, and Easy Access. With a Singleton, we’re stating “There will only ever be 1 AudioManager, therefore if we’re looking for an AudioManager we should know exactly where it is”.

Let’s walk through how we can implement this pattern:

* Create a script with a static instance of itself (meaning there is no instance, there’s only 1)
* When creating new script of this type, check and see if it already exists (is the static instance null)
  + If it already exists, Destroy this one
  + If it’s null, we now have our static instance! Don’t create any more
* Access the static instance with ScriptName.Instance.(whateverYouWant)



Essentially, we can create an AudioManager, then check to see if we already have one, and if we do Destroy the excess ones. Then we can call our AudioManager no matter where we are.

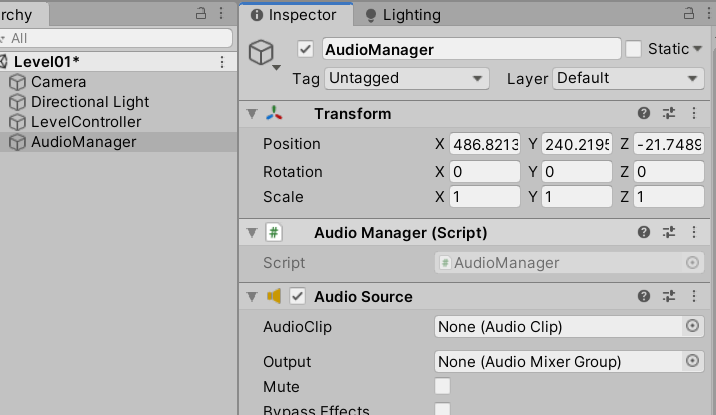
*Note on Singletons: The reason I say the Singleton Pattern is controversial is because it discourages good practices. It’s very quick and easy way to get global access to a class, which has its pros and cons. It also solves 2 problems, when likely you may only need it to solve 1: Convenience and enforcing there’s only 1 of a thing. It also means that if a Singleton can travel between scenes, it creates scenarios that are hard to produce because you can’t gaurantee everything that’s in and out of your scene at any one point in time. If everything in your scene is Create()’d and Destroy()’d, then you always know what’s there.*

*The truth on whether or not a Singleton is bad or good lies somewhere between… if your problem is you need access to a thing there will only ever be one of… it ‘can’ be a good, clean solution. If you use a Singleton just because it’s faster than bypassing normal defensive coding practices… it can easily become bad very quickly. An occasional Singleton can save a lot of time spent planning architecture (for better or for worse), but over-using them with sprinkle your codebase with dependencies and very quickly create a folder full of spaghetti.*

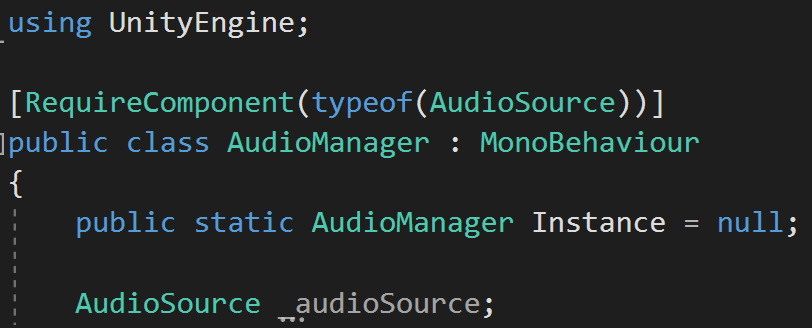
With this in mind, let’s try creating our AudioManager as a Singleton to see what kind of power (and restrictions) this Design Pattern affords us.

## Creating the AudioManager

**Create a new GameObject named AudioManager. Create a new script named AudioManager and attach it to our AudioManager gameObject**. **Attach an AudioSource** (you can leave AudioSource settings at default), since we’ll need an AudioSource to play our music.



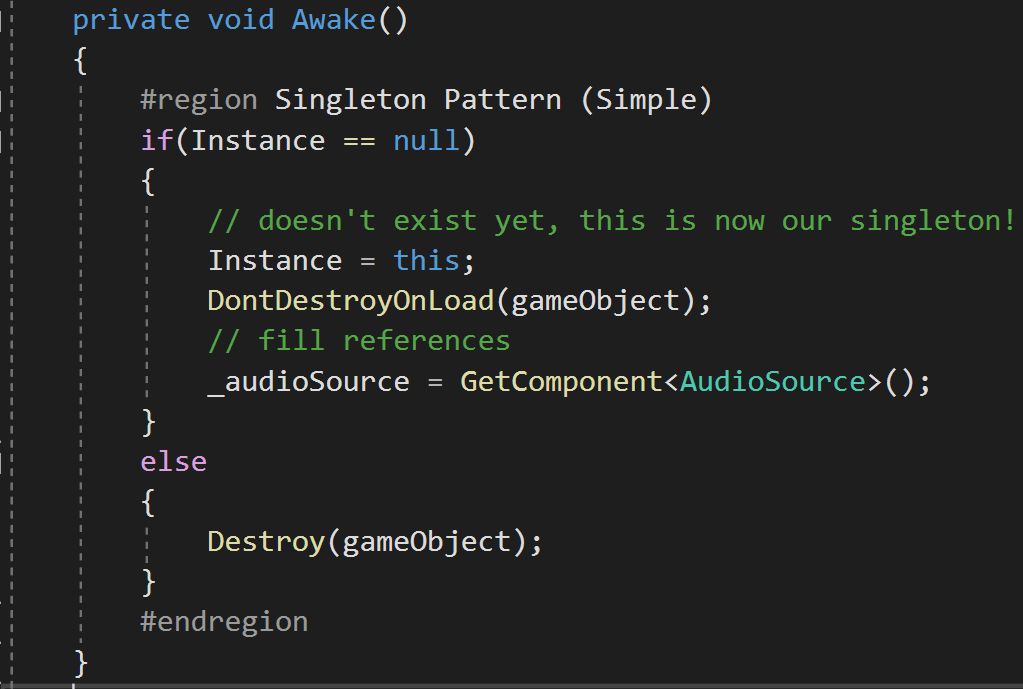
Inside of the AudioManager.cs, add the following code to create the Singleton Pattern



First we are requiring an AudioSource to be attached to this gameObject, which we have already attached.

Next, we’re creating a public static Instance of the same class as the class it is contained within. This is the crucial part of the pattern: the class keeps track of whether or not the ‘one’ instance exists. If this is confusing, well… that’s understandable. If we keep track of this one instance, and static data is accessible from anywhere, then this Instance will be our access point that lets us see things inside of our AudioManager class, no matter where we are.

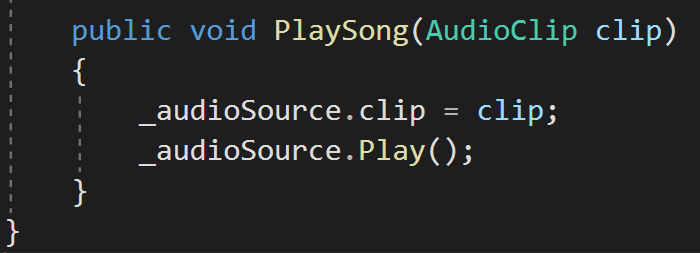
Continue to add more code:



I have a #region and #endregion to label our Singleton Pattern just to make it easier to collapse, if we want to (it’s optional really). In Awake, when a new AudioManager wakes up in the scene it does a check. Is there already an AudioManager? (aka is the Instance null or not). If there’s no Instance, this is now our one AudioManager so assign it and Don’t Destroy this object anymore. If there’s already one, just Destroy this one… we can’t have 2.

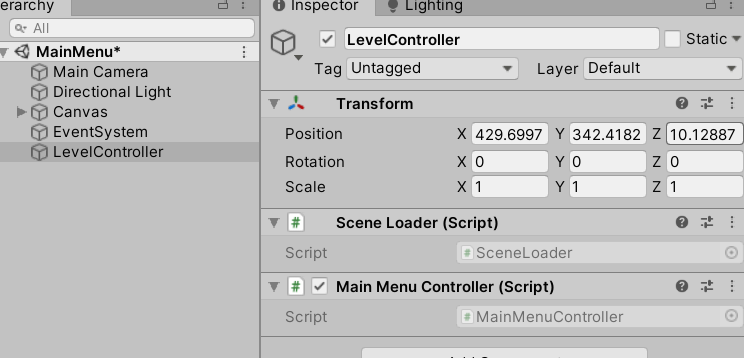
Additionally note that I’m only filling my references (the GetComponent search) at the one moment we decide this is our AudioManager. It saves us a few calls from excess AudioManagers spending resources searching, but then destroying themselves anyways.

Finally, let’s give it the functionality to play a new music clip:

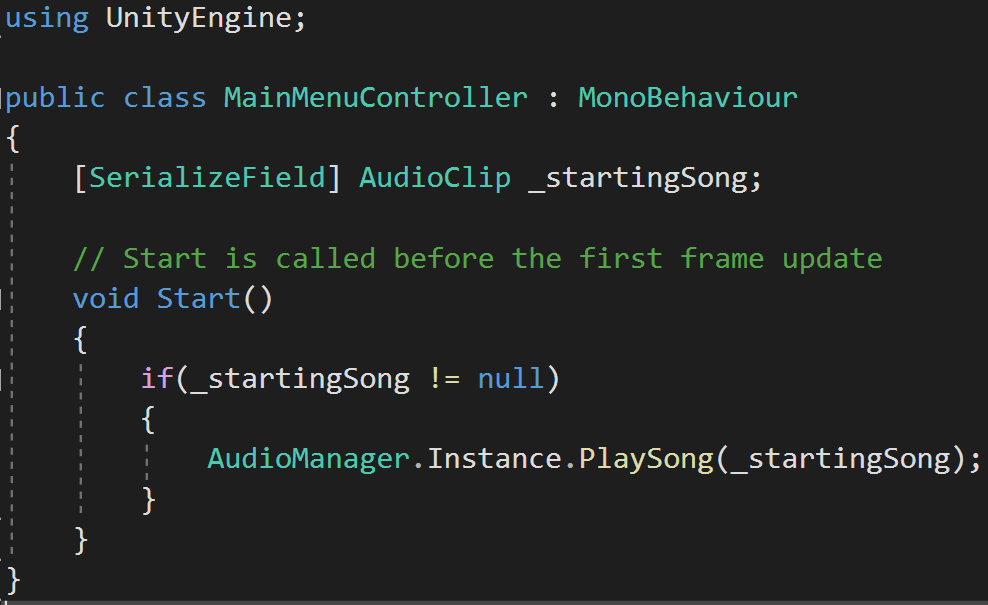


Let’s open up our MainMenu and call a PlaySong() on Start.

**Open the MainMenu scene, and create a new script named MainMenuController. Attach this script to the LevelController.**



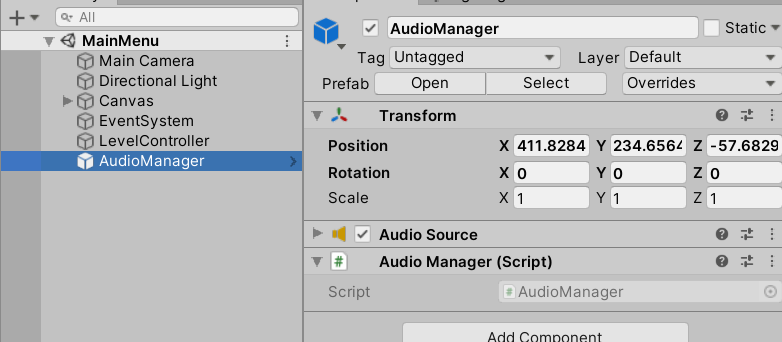
Add the following code to the **MainMenuController**.cs script:



Make sure to **import a music track**, and **fill it in the Inspector** on your MainMenuController.

The song’s not playing yet, but you see how we can easily get access to our PlaySong function? That’s the power of Singletons, but we still need to ensure there’s an AudioManager in the scene.

The last thing we need to do to ACTUALLY play our AudioManager music track is add an AudioManager to the scene. While in MainMenu scene, add a new GameObject named AudioManager, and attach the AudioManager.cs script (or, ideally, make the AudioManager a prefab and drag it into both scenes)



Keep in mind that if you leave “PlayOnAwake” checked, it will play when it starts. You may consider unchecking it and controlling it specifically.

Test your level and make sure it works! Don’t forget to set your Music tracks to loop.

*Notice that in this Setup, each time we return to the MainMenu it’s Restarting the song (since MainMenuController Start() always gets run each time we return to the scene). In a real game you’d have to specifically decide when you want each track to change. If you wanted this track to only play throughout, without any restarting, you’d probably want to move the code from the LevelController Start(song), into the AudioManager Start(). That way the song is only active during the duration of the manager.*

Note: Lazy Loading – Another thing you may be wondering… what happens if we don’t have an AudioManager in the scene, even though we’re expecting one? Currently… you’ll get an error. There’s a coding pattern called ‘Lazy Loading’, which means you only ever create the GameObject the first time it is accessed by something. Many Singleton patterns use this concept (it’s the safer way to build it), but I didn’t want to get into properties for now. If you wanted to implement it, look into Lazy Loading and it will gaurantee you ALWAYS have a Singleton when you need it, regardless of if you remembered to put one in the scene or not.

# Tracking Current Score

Now that we’ve gone over Game Persistence, let’s talk about File Persistence with our High Score system. As with all of our other features, let’s figure out how we want it to work before we start building it.

## High Score Concept

High score functionality summary:

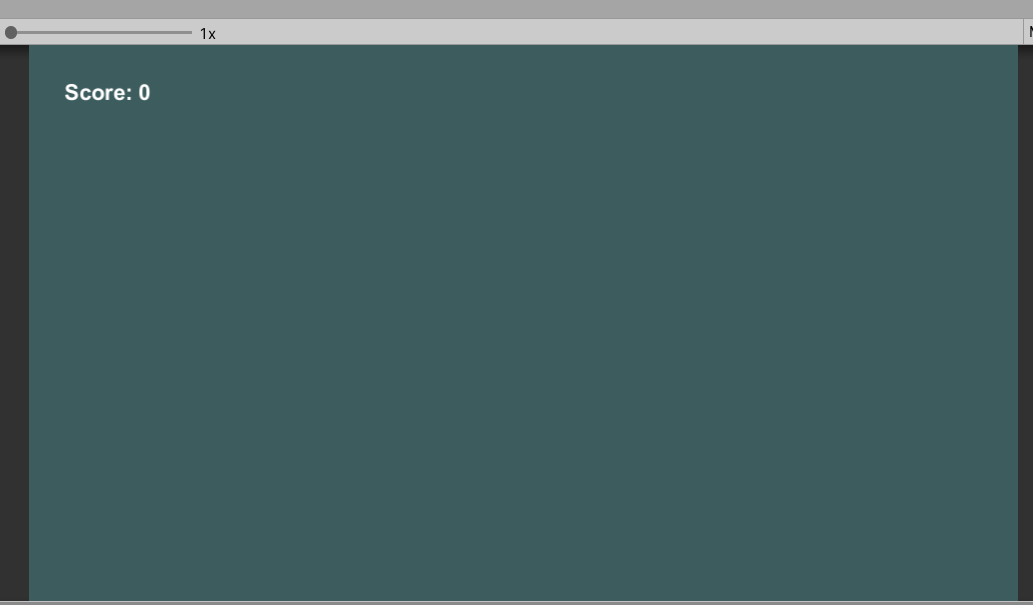
* While in the game level, keep track of the current score
* When trying to exit the level, check and see if the current score is higher than our HighScore saved on file
  + If it is – save our current score as the new high score! And return to the menu.
  + If it isn’t – throw out our current score, since it wasn’t high enough. And return to the menu
* While inside the menu, load our HighScore from file and display it in the UI

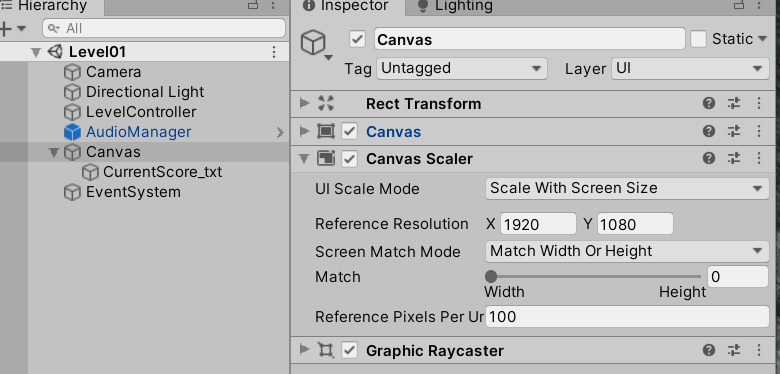
If we can do those things, we’ll have our HighScore system. Also, if we can implement these things, then we can additionally save whatever we want to file. This is a common technique in games (Save file, player levels, last position, last level played, etc.) and will be a crucial concept for building anything beyond a basic prototype.

## Implementing Current Score

First we need some kind of Display for our current score while we’re in the level. **Open up your Level01 scene** (or whatever you called it).

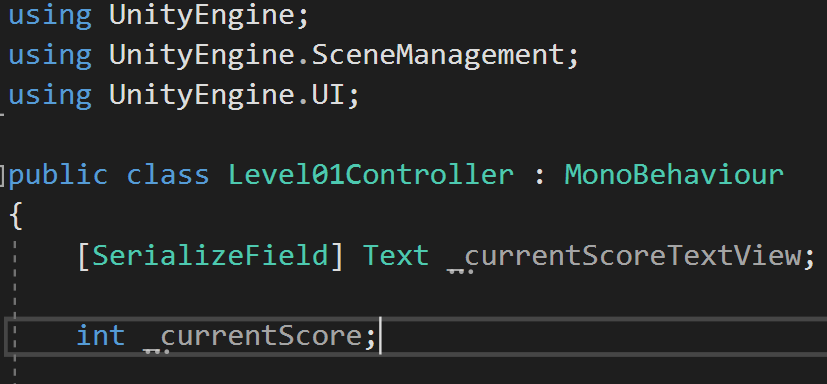
Build a Canvas with a Text UI object that displays “Current Score: “ like the screenshot below. I’m assuming at this point you can look at the previous example and build this yourself. Don’t forget to set your Canvas Scaler to “Scale with Screen Size” and 1920 x 1080!





Next, let’s keep track of the current score inside of our Game level, as this is the only place where we’ll be able to affect our score.

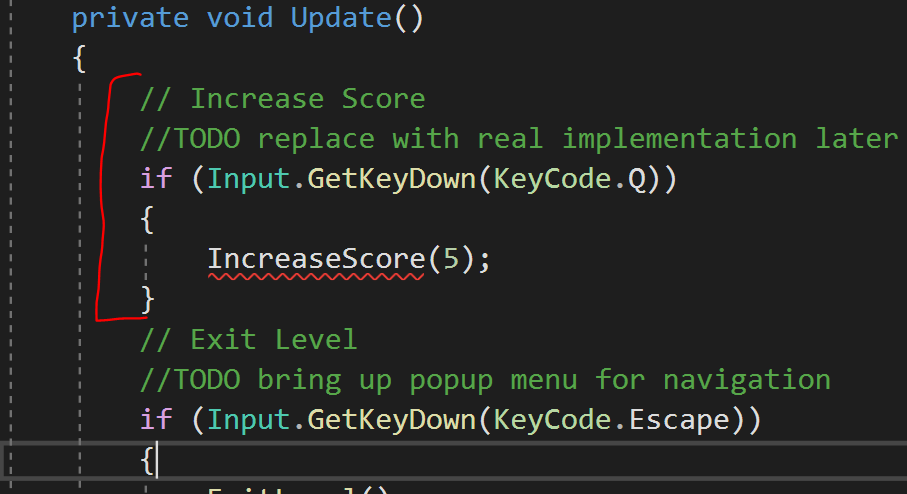
**Open the Level01Controller script**, and add the following code to what’s already there:



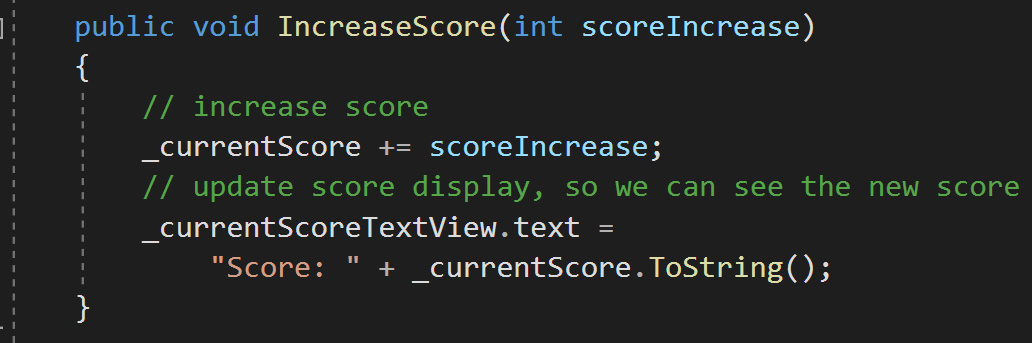
Don’t forget “using UnityEngine.UI” at the top, so we can access UI. I’m getting a reference to our Text UI object we just made. I’m adding ‘TextView’ at the end of the name just to be really clear about what this object is… it’s a UI object that’s just used for displaying (the word View comes form Model/View/Controller architecture. Feel free to look more into this if you’re interested in coding UI systems). Lastly, we have a currentScore that we’ll increase and decrease. Any time we increase/decrease the score we need to update the TextView with the currentScore.

*Don’t forget! Just because we increase the score doesn’t mean the view will update. In your head you should be thinking about your internal value (the current score) and your display (the text UI object). Anytime your score changes, you also need to update the display! There’s automated ways to do this with data binding or event systems, but for now we’re going to do it manually because it’s good practice to reinforce the concept.*

Continue with the code:



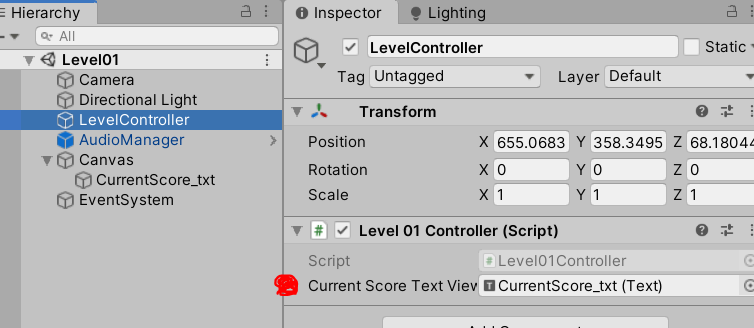
For testing, we’re putting an ‘Increase Score’ on a button press, Q, but eventually we want to move this to an in-game event: defeating an enemy, collecting a thing, performing a task, etc. For now we just want to prove the system.



In this function we’re just increasing the score count, and making sure the display knows to update the score count into the view. Note that we have to add “Score: “ + int.ToString(). This makes sure we’re just not replacing all of the text with a number (we still need the label) and .ToString() is making sure that we convert our number into a string.

*Don’t forget – the text field needs a string value, so if we put in anything other than a string into that field we need to convert it! .ToString() is a handy function attached to ints and floats that allows easy conversion.*

Don’t forget to fill in your reference to your currentScoreTextView!



Test it to make sure that it works (it should). If not, make sure you fix any bugs before continuing.

# Saving/Loading High Score

We’re increasing our current score, but we still need a way of saving and loading our HighScore to file. We’ll tackle this next.

## Saving/Loading Primer

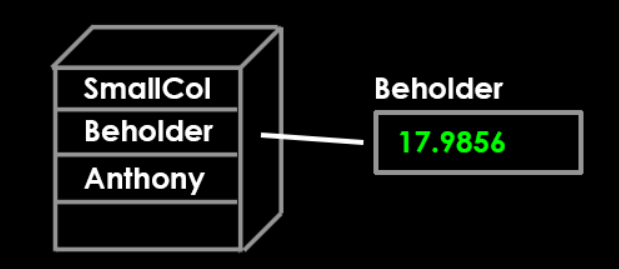
For this demo, we’re going to use a thing called PlayerPrefs. PlayerPrefs is a simple saving/loading system in Unity that allows you to easily store data inside the PlayerPrefs data file. We could store any simple data we want (int/float/string/etc.).

*One thing you should know about PlayerPrefs is that it’s not encrypted -> meaning, it’s easy to tamper with by players. Normally you wouldn’t want to put any gameplay data in here (player level, money, high score, etc.) because the player could easily open up the PlayerPrefs file and give themselves 99999999 gold. I’m sure you see the issue. The reason we’re using PlayerPrefs in this example is because it’s the easiest method for communicating the concept, and I think that’s more important for now. If you want to learn how to use a “proper” Save System for gameplay data, I recommend looking into JSON utility or the Binary Formatter. There’s many methods of saving, and File Management is a whole discipline of programming waiting to be explored.*

The way PlayerPrefs works is by defining the data type and a get or a set. Some examples:

PlayerPrefs.GetInt(“PlayerLevel”) – This would look in the PlayerPrefs database for a variable named “PlayerLevel” and return whatever value is stored there

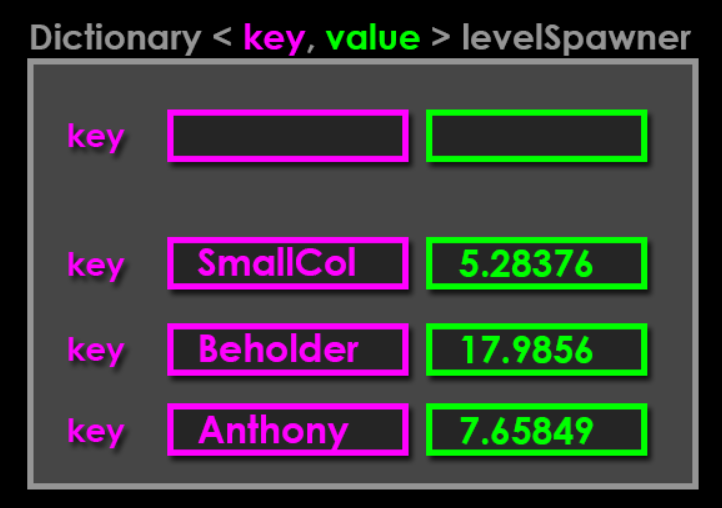
PlayerPrefs.SetFloat(“CompletionTime”, 16.8f) – This would look for a variable named “CompletionTime” in the data base, if it doesn’t exist it would create one, and store the value 16.8 inside. If we did a Get/Set for this variable name later, it would return the value we just stored, until we store a new value.



Note: PlayerPrefs is actually using a collection type called a Dictionary.

**Dictionary** – Collection type that pairs a key with a value

A Dictionary takes in a “key” and stores a value associated with that key. Just like a Dictionary! You can also think of it as a drawer with a label on it. If you look for the drawer named “blah” you open up that drawer and get whatever data is inside.



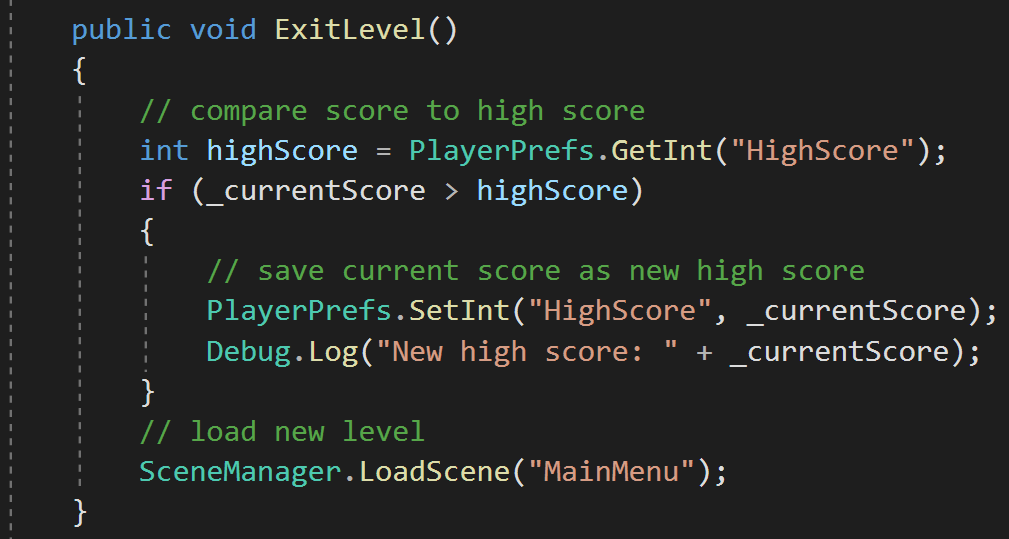
An even easier way to think about it is Get = Load, Set = Save. Get/Load/Read vs Set/Save/Write. They’re all terms for similar concepts.

Let’s use these concepts to implement our High Score.

## Implementing HighScore Saving with PlayerPrefs

When we exit the level, let’s check our HighScore to see if our CurrentScore is higher than what’s in Playerprefs already. If it’s not, we’ll save the current score.

Open the **Level01Controller.cs script** and add the following code:



Before exiting the level, we’re doing our check to see if our current score is higher than whatever we have stored under “HighScore” in our PlayerPrefs database. If it is, we store it! (and check it with Debug.Log() since we aren’t displaying it… yet).

*Note: You’ll see that we’re leaving ourselves open to spelling mistakes here by passing in a lot of string references. What if we mispelled HighScore?? Best practice is to store your string at the very top of the script and just reference the string stored in that variable. That way you’re only spelling it out once, and minimizing mistakes.*

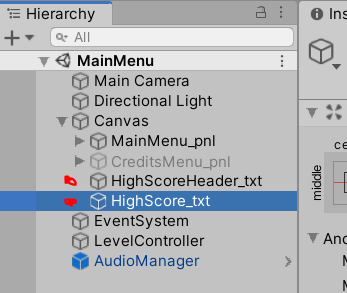
Test it by hitting play, pressing Q a few times, then Escape to exit, Start game button, pressing Q a few times, etc. See when it’s storing a new value and when it’s not. It’d be great if we could actually display the HighScore to make it more visible, so we’ll do that next.

## Displaying the HighScore

Finally, we want to Display the current HighScore on the Title screen by loading the HighScore from PlayerPrefs. Again, be very careful with spelling here.

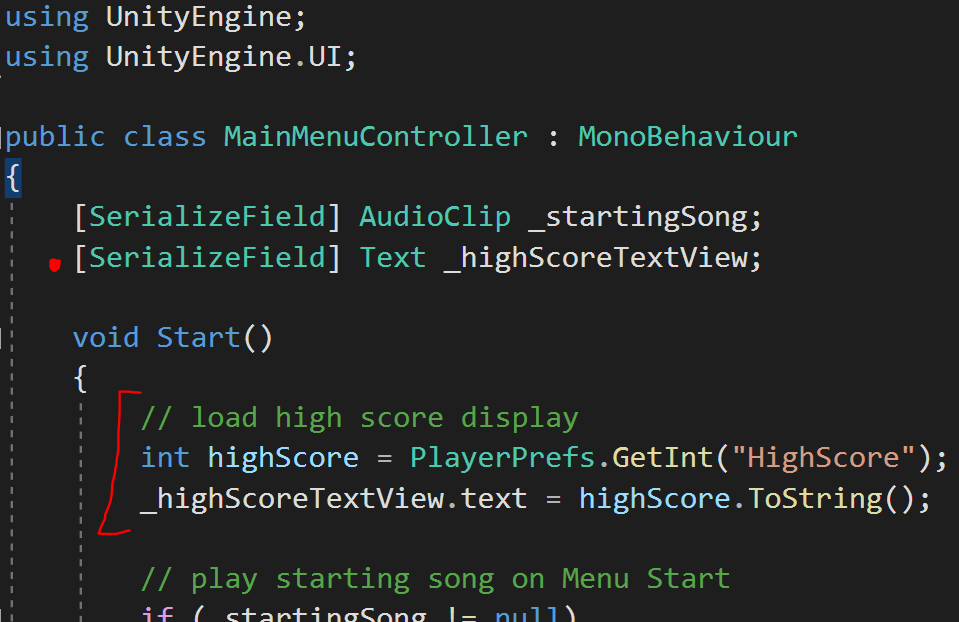
Open up the MainMenu level and build a Text display for your HighScore, as shown below. Note that this time I’m using 2 separate Text objects: one for the ‘Header’ and one for the ‘Score’. This way I’m just feeding the score into the ScoreText object, and the label stays consistent. It also allows me to add different styling on the number itself. Really, I just wanted to show you both methods. Go ahead and build this yourself.



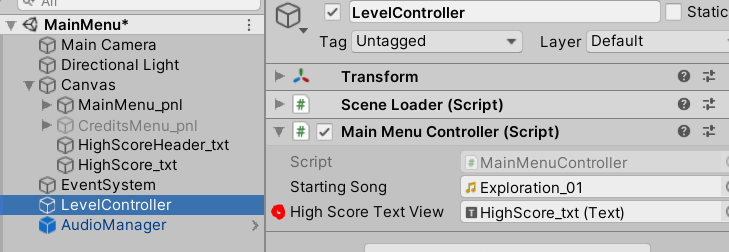


Once you have the display set up, it’s time to add Behavior.

**Open the MainMenuController.cs script** and add the following code to what’s already there:



Lastly, don’t forget to fill in your references in the Inspector.

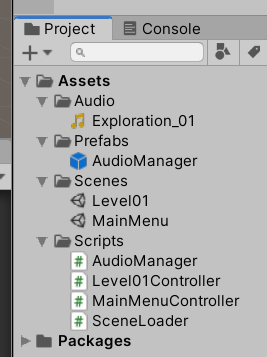


Test it out! We should see our High Score display the current HighScore (from file) each time we enter the MainMenu. Note that PlayerPrefs automatically handles error checking for us, so if it doesn’t exist yet it doesn’t return anything. If you wanted to write your own Save/Load system you’d definitely want to check whether or not the data exists before displaying it, but in this case it’s fine.

Don’t forget to test it in your Executable too, just in case. Especially with Save/Load systems you always want to check it on the final device.

# Cleanup

Don’t forget to organize your files! Here’s my final organization for my Project Assets:



# Summary

In this walkthrough, we built the skeletal structure of our game. Let’s review some of the terms we learned in this lesson:

**Game State** – a “section” of your game with specific behavior

**Scene** – the container that holds a portion of your game behavior, isolated from other sections of your game

**Canvas** – (Unity specific) contains objects and tells them how to render to the camera

**Panel** – Contains objects in a group, as an “island” in your visual display

**Button** – Image that DoesSomething() on click

**Event** – A container of ‘methods’ that sends a notification for when those things should happen (aka OnClick)

**Event Listener** – Method that “watches” an event notification, and reacts when it the event is invoked

**Singleton** – Enforces only one instance, global access

**Lazy Loading** – Creates the object the first time it is accessed by something in the scene

**Scene Persistence** – Data only exists within the scene. Scenes cannot pass information to each other.

**Game Persistence** – Data that travels between scenes, but does not need to exist longer than the game session (AudioManager)

**File Persistence** – Data that exists between play sessions, outside of the game and on file (HighScore)

**Encapsulation** – Data is hidden from things that don’t need to see it.

**PlayerPrefs** – (Unity specific) prebuilt system for quick saving/loading. Not encrypted!

**Get/Set** – Loading the Data, Storing the Data

We left ourselves open to expansion by defining the different portions of our game (MainMenu, Level01, etc.) and setup some scene navigation between the game states. We also talked about Singletons and Manager-type things, and how to easily access them. Finally, we discussed Saving/Loading and how you can easily implement that in your games using Unity’s PlayerPrefs.

I hope that with these concepts you feel more comfortable expanding your prototypes into something bigger and more robust.