# Quest 7 - Steps

## 1) Import

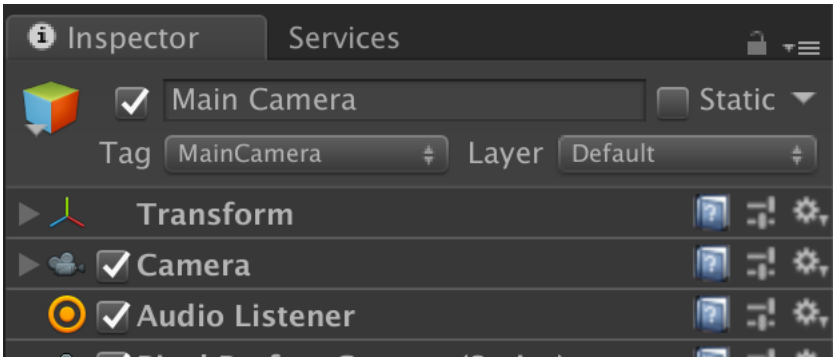
Download the Q7 files from Canvas and import the “shoot,” “miss,” and “impact” to your project’s “Assets/Resources/Sounds/“ folder.

## 2) Audio Listener

Audio has three core requirements in Unity: a) an Audio Clip object to play, b) an Audio Source component to play it from, and c) an Audio Listener component to hear it

The default Main Camera comes with an Audio Listener by default.

Select your Main Camera to confirm that it ALREADY has an Audio Listener. Do NOT add a second Audio Listener or you will get errors.



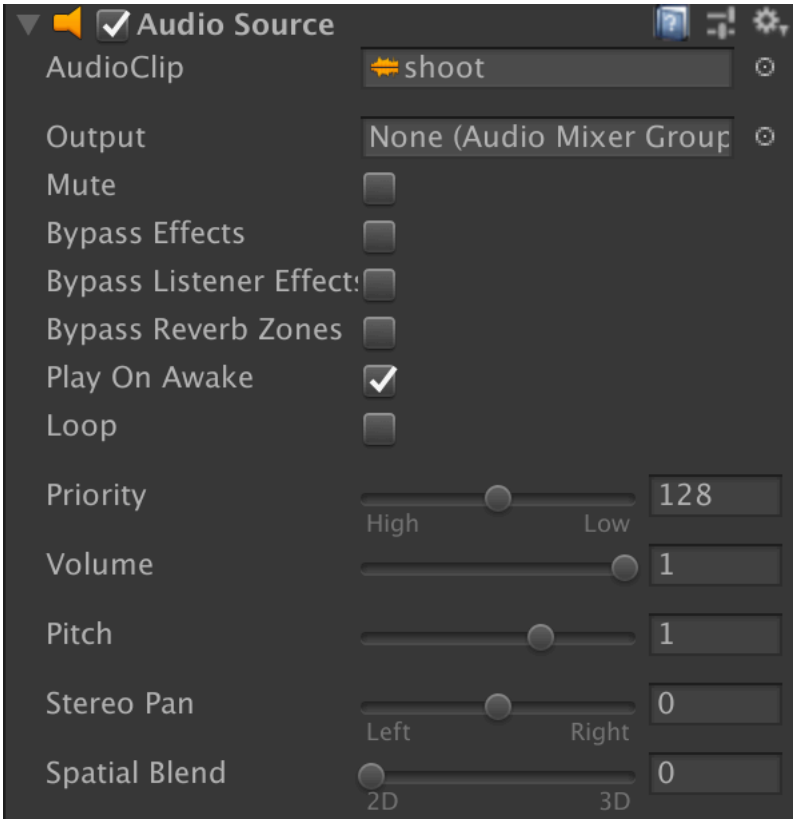

## 3) Audio Source

Your Audio Source is the gameobject that plays the sound. Playing sounds from different objects can give a sense of relative position between the Audio Source and Listener in both 2D and 3D games.

Select your Projectile Prefab and add an Audio Source component. Make sure “Play On Awake” is checked because we want the shoot sound to play right away and set your “Spatial Blend” to 2D to match our style of game.

Drag the shoot sound file into the AudioClip blank.

Now, whenever you fire a projectile, it automatically plays the shoot sound.




# 4) More Sounds

We also want Hit and Miss sounds when the projectiles hit the Targets and Ground, respectively. We can't have the projectiles or targets play these sounds because they (and their Audio Source components) are destroyed right after impact.

Instead, we will create a separate SoundManager object to play the different sounds.

Create a new Empty Game Object in the scene named SoundManager. Attach an Audio Source component to it along with a new Sound Manager script.

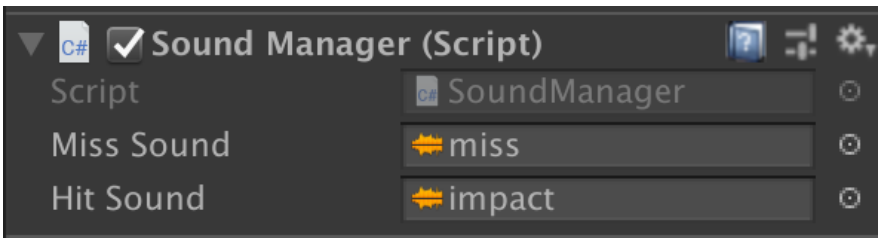
Open your SoundManager script. It will be accessed using a Static Instance Reference similar to the GameController in the other projects.

```
public class SoundManager : MonoBehaviour {

    public static SoundManager instance;

    void Awake() {
        instance = this;
    }
}
```

The actual sound files we want to play are not a part of our script or game object, so we have to create Public Outlets in our code to reference them. Similarly, we have to create an outlet to reference the Audio Source component;

```
lic class SoundManager : MonoBehaviour {

 public static SoundManager instance;

 // Outlets
 AudioSource audioSource;
 public AudioClip missSound;
 public AudioClip hitSound;

 void Awake() {
     instance = this;
 }

 void Start() {
     audioSource = GetComponent<AudioSource>();
 }
```

Fill in the blanks created by coding your public outlets.



Next, we'll create public functions that other objects can call to trigger sounds.

```
public void PlaySoundHit() {
    audioSource.PlayOneShot(hitSound);
}

public void PlaySoundMiss() {
    audioSource.PlayOneShot(missSound);
}
```

In our Projectile script, we will set up the conditions for our different sounds and call the SoundManager appropriately.

```
void OnCollisionEnter2D(Collision2D collision) {
    if(collision.gameObject.GetComponent<Target>()) {
        SoundManager.instance.PlaySoundHit();
    } else if(collision.gameObject.layer == LayerMask.NameToLayer("Ground")) {
        SoundManager.instance.PlaySoundMiss();
    }

    Destroy(gameObject);
}
```

# 5) Introducing Save Data

Unity can save Persistent Data by storing either a complex file format of your own design in the device's documents or simple datatypes in the "PlayerPrefs." This tutorial covers the latter which is the most common.

These simple datatypes include string, int, and float.

# 6) Score Counter

We are going to update the PlayerController to save an ongoing score counter, so the player can quit the game without losing their score.

Use a public static outlet to make the PlayerController easy to reference.

```
public class PlayerController : MonoBehaviour

    public static PlayerController instance;
```

```
// Methods
void Awake() {
    instance = this;
}
```

PlayerController needs a property for storing the score as well as an outlet for the score UI. Don't forget to add using UnityEngine.UI; to your namespaces or the Text type will throw an error.

```
public Text scoreUI;

// State Tracking
public int jumpsLeft;
public int score;
```

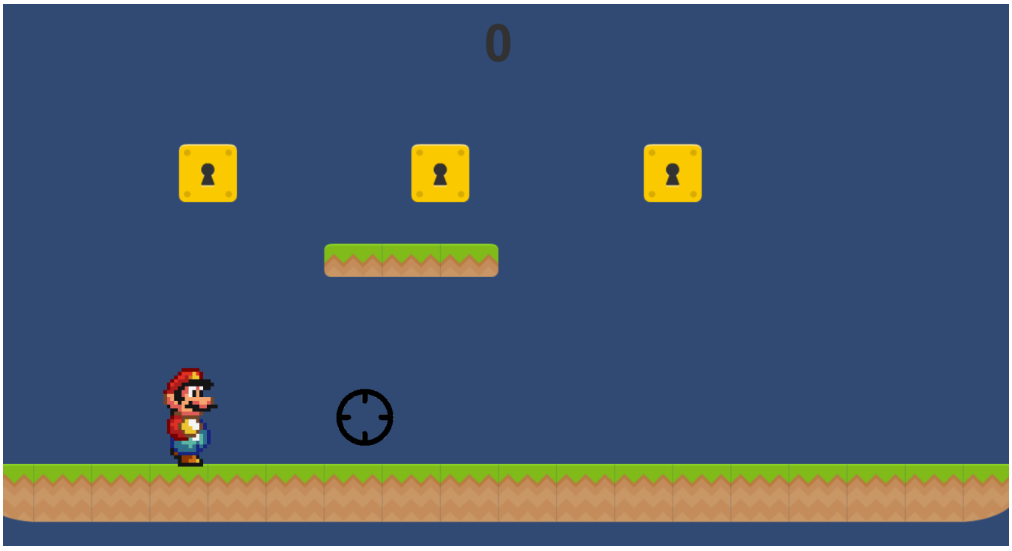
In the Update of our PlayerController we will update the UI with our current score.

```
void Update() {
    // Update UI
    scoreUI.text = score.ToString();
```

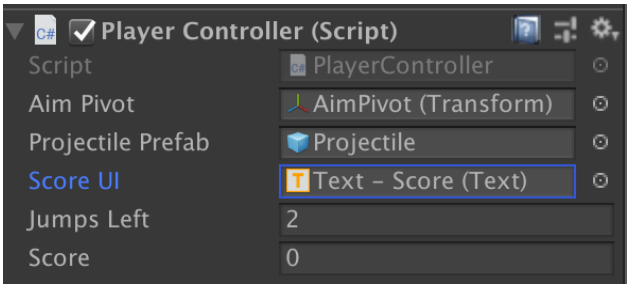
In the Target script, update the collision event so it also adds to the player's score.

```
lic class Target : MonoBehaviour {

void OnCollisionEnter2D(Collision2D collision) {
    if(collision.gameObject.GetComponent<Projectile>()) {
        PlayerController.instance.score++;
        Destroy(gameObject);
    }
}
```

Create a Text UI object and adjust its formatting so it shows in the top-center of the screen. Remember to create UI that scales and is always readable across devices, just like previous exercises.



Don’t forget to assign the public outlet in the PlayerController.




# 7) Making Score Persistent

If you play the game in its current state, the score system will still reset when you restart the game.

To persistently save our score, we also need to store and update an entry in PlayerPrefs whenever score changes during our Target script.

```
lic class Target : MonoBehaviour {

 void OnCollisionEnter2D(Collision2D collision) {
     if(collision.gameObject.GetComponent<Projectile>()) {
         PlayerController.instance.score++;
         PlayerPrefs.SetInt("Score", PlayerController.instance.score);
         Destroy(gameObject);
     }
 }
```

PlayerController must also reload during its Start event.

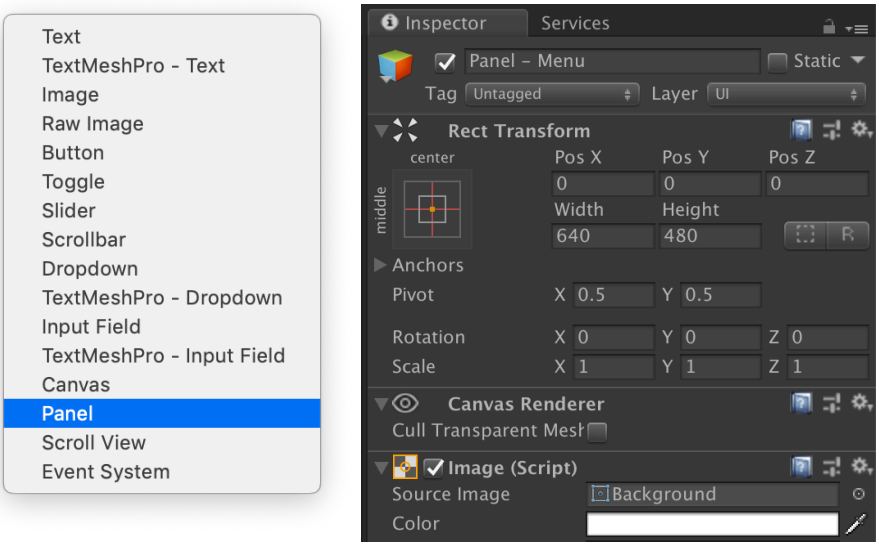
```
void Start() {
    rigidbody = GetComponent<Rigidbody2D>();
    sprite = GetComponent<SpriteRenderer>();
    animator = GetComponent<Animator>();

    score = PlayerPrefs.GetInt("Score");
}
```

Now when restarting the game, score returns to where the player left off instead of 0.

# 8) Menus and Submenus

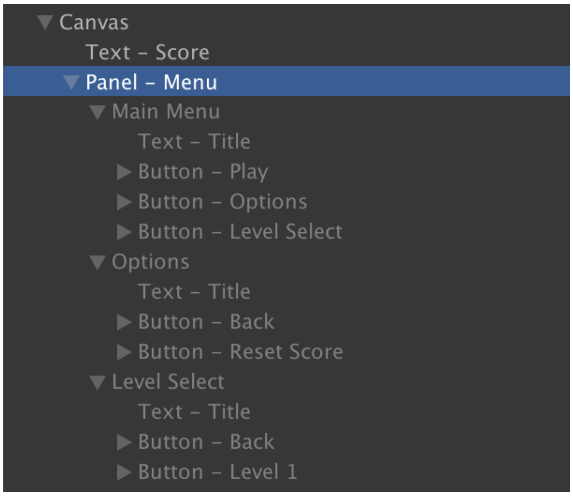
Submenus are usually just nested UI objects that are turned on and off as needed, rather than separate scenes.  We will set up a Pause Menu with submenus for Options and Level Select. All of these menus will be contained in a UI Panel which gives us a rectangle background.

Remember to create UI that scales and is always readable across devices, just like previous exercises.

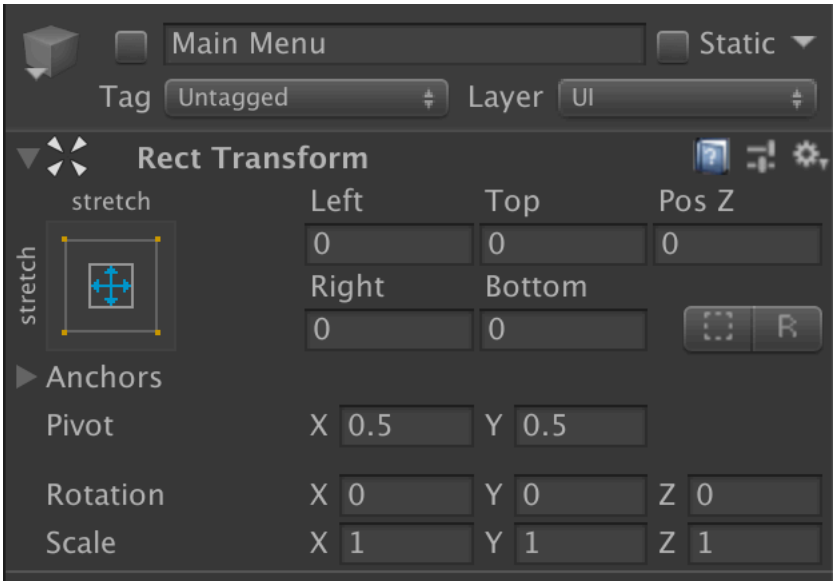




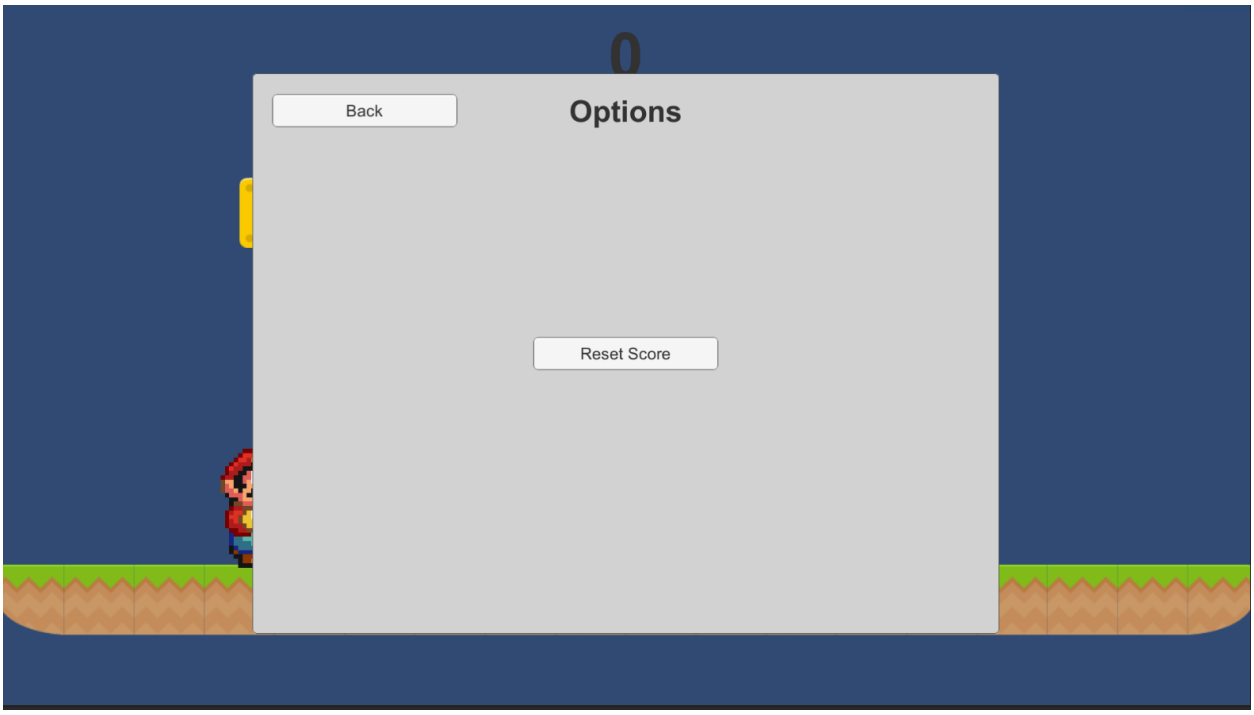
Refer to this Scene Hierarchy and these screenshots for setting up your Menus.



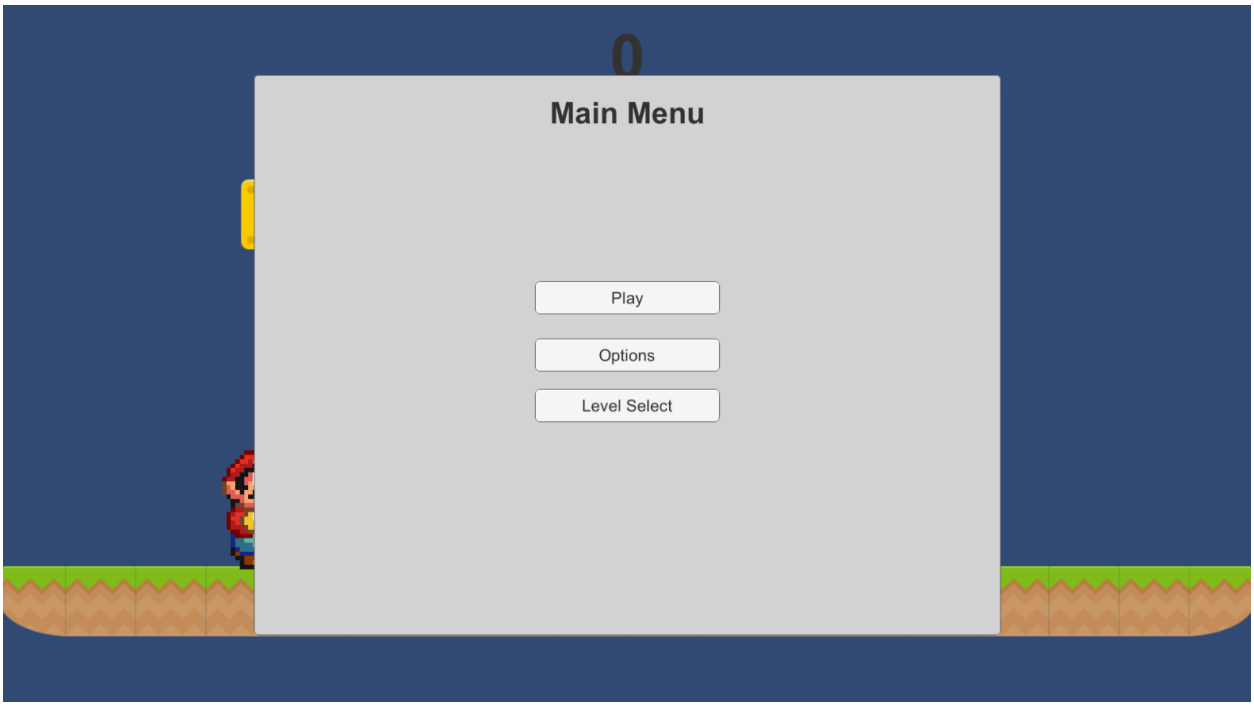

The “Main Menu,” “Options,” and “Level Select” game objects are just empty objects stretched to fill the parent:



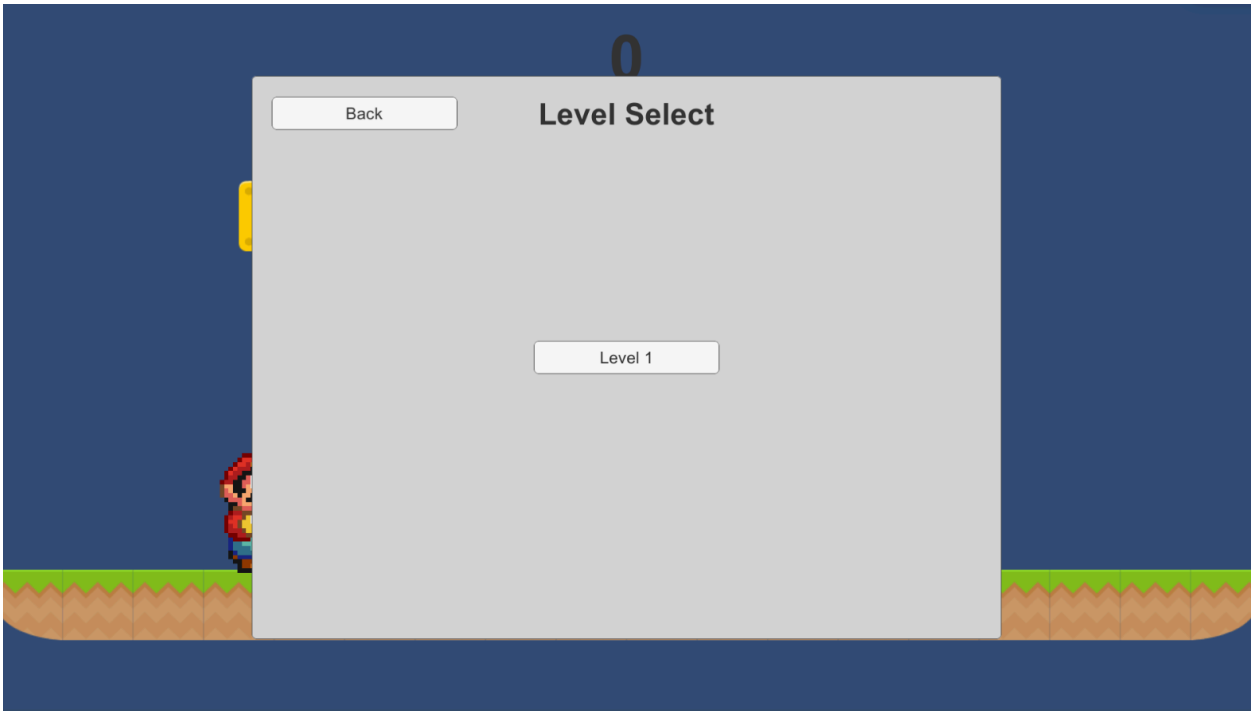

Main Menu content (when Options and Level Select are turned off):




Options menu content (when Main Menu and Level Select are turned off):




Level Select menu content (when Main Menu and Options are turned off):




# 9) Menu Coding

Create a MenuController script and attach it to your “Panel - Menu” object.

MenuController is in charge of all menu and submenu actions.

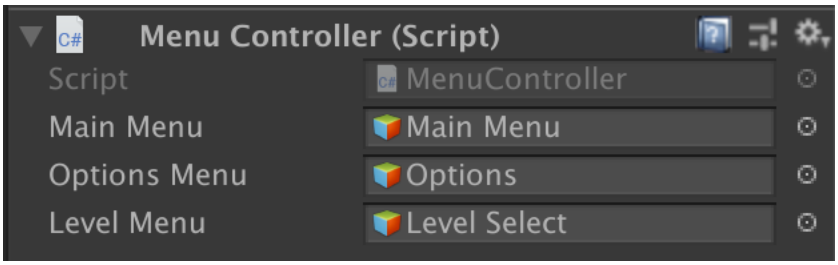
Use a static instance reference to make the menu easily accessible from other objects.

```
lic class MenuController : MonoBehaviour {

 public static MenuController instance;

 // Methods
 void Awake() {
     instance = this;
 }
```

In order for our MenuController to control our 3 (sub)menus it needs outlets to reference them.

```
// Outlets
public GameObject mainMenu;
public GameObject optionsMenu;
public GameObject levelMenu;
```

Don’t forget to fill in the blanks for the outlets.




We are going to setup some utility functions for managing our menus before we code the primary menu functionality.

This Switch Menu function manages the cleanup when switching between menus ensuring other menus are properly turned off when another is turned on.

```
void SwitchMenu(GameObject someMenu) {
    // Turn off all menus
    mainMenu.SetActive(false);
    optionsMenu.SetActive(false);
    levelMenu.SetActive(false);

    // Turn on requested menu
    someMenu.SetActive(true);
}
```

To further streamline setting up our Button component click events, we're going to create even simpler utility functions for switching to specific menus.

```
public void ShowMainMenu() {
    SwitchMenu(mainMenu);
}

public void ShowOptionsMenu() {
    SwitchMenu(optionsMenu);
}

public void ShowLevelMenu() {
    SwitchMenu(levelMenu);
}
```

Since we don't want our menu to appear right at the start of the game, the Hide function is called from the Awake menu.

It is very important to leave "Panel - Menu" on at the start of the game and let it hide itself, otherwise the static instance will not be setup and calls to the Menu will produce red errors/ crashes.

```
void Awake() {
    instance = this;
    Hide();
}

public void Show() {
    ShowMainMenu();
    gameObject.SetActive(true);
}

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

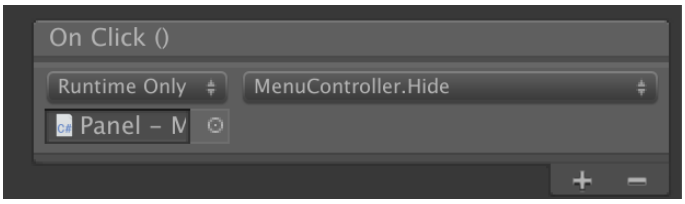
In PlayerController's Update function, the Escape key will trigger showing the menu.

```
if(Input.GetKey(KeyCode.Escape)) {
    MenuController.instance.Show();
}
```

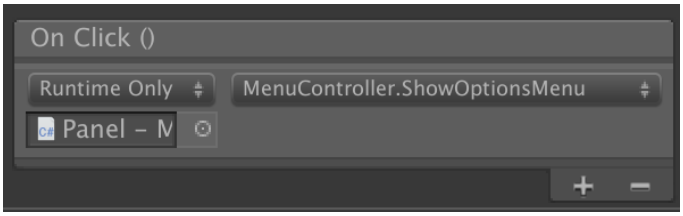
If you playtest the game now, you should be able to press Escape to show the Menu Panel, and it should show the Main Menu by default. None of the other two menus should be visible.

# 10) Main Menu Coding

The Play button hides the menu and returns to the game.



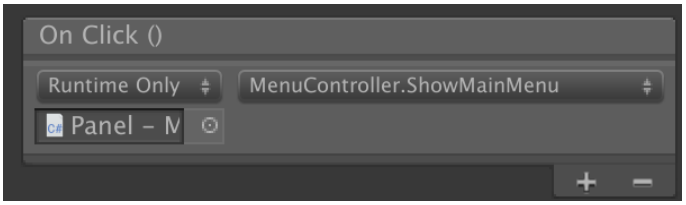

The Options button switches to the Options menu.




The Levels button switches to the Levels menu.


# 11) Options Menu Coding

The Back button switches to the Main Menu.



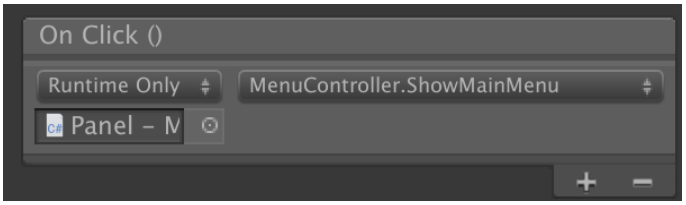

The Reset Score button requires a new function.

```
public void ResetScore() {
    PlayerPrefs.DeleteKey("Score");
    PlayerController.instance.score = 0;
}
```

Hook that function up to the Reset Score button.


# 12) Level Menu Coding

The Back button switches to the Main Menu.

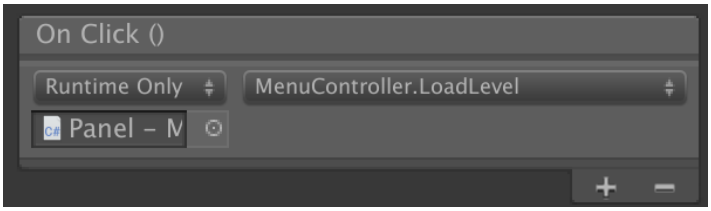



The Level 1 button requires a new function.  The game only has one level, so we'll just switch to our current scene, but this feature could be expanded for as many levels as you need.

Make sure to use whatever the actual name is of your game Scene, add it to the Build Settings as appropriate, along with any other SceneManager steps detailed in prior quests.

```
public void LoadLevel() {
    SceneManager.LoadScene("SampleScene");
}
```

Hook that function up to the Level 1 button.




# 13) Gameplay Tweaks

You may have noticed gameplay still continues even when the menu is on screen. To fix this, we have to make two changes to the game: a) Prevent the Update event progressing, and b) Freeze the engine's time scale, so animations pause.

For the first one, we will use an isPaused boolean in the PlayerController script.

```
// State Tracking
public int jumpsLeft;
public int score;
public bool isPaused;
```

We stop the Update loop execution right away if the game is paused.

```
void Update() {
    if(isPaused) {
        return;
    }
```

Finally, we freeze the time scale and update the isPaused property from the MenuController script whenever the menu is shown or hidden.

```
public void Show() {
    ShowMainMenu();
    gameObject.SetActive(true);
    Time.timeScale = 0;
    PlayerController.instance.isPaused = true;
}

public void Hide() {
    gameObject.SetActive(false);
    Time.timeScale = 1;
    if(PlayerController.instance != null) {
        PlayerController.instance.isPaused = false;
    }
}
```

## 14) PLAYTEST

Make sure you have no red errors from any functionality in your game.