Build a Working Game

At this point we know the basics of build a scene, adding in elements such as physics and triggers and basic programming using C#. Now we will put it all together to build a working simple game that includes audio, levels and start and end screens. We will also learn how to export the game into a simple stand alone working application.

Below is working version of the game loaded on to a server:

https://davidhurwich.github.io/RollerBallGame/index.html

Setup

Begin by downloading lab-working-game from iLearn. Unzip the file. Use Unity Hub to add the project.

Open Level-1 and take a moment to look over the scene. We have a simple gameboard that has green diamond objects on it. The green diamonds each have triggers and game tags of "*diamond*" attached to them. Also in the game is a red sphere called player that a script called *PlayerMove.cs* attached to it.

Take a moment to open PlayerMove.cs into a code editor. You will see that there is a variable called diamond that is set to zero. There is code in the *Update()* that control the sphere's movement. There is also a *OnTriggerEnter()* event that checks to see if the sphere colliders with an object with a tag of "diamond" applied to it.

Put the game into game mode and you'll see that you can move around the game board using the up, down, left and right keys. If you collide with the diamonds they disappear or are removed from the game board.

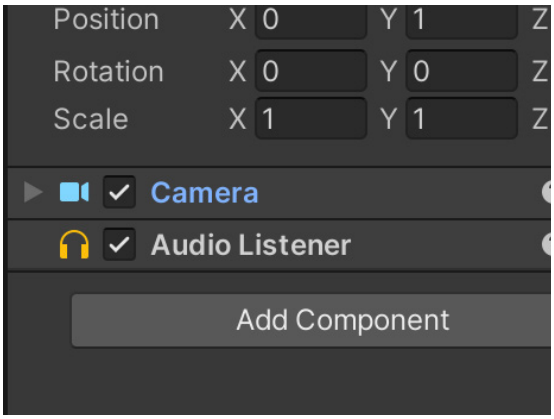
Add Audio to the Game

Audio in a game or virtual world can be very important. Background music can add atmosphere and life to your world. Sound effects are also important as they can give feedback to the player if what they did was good or bad.

Let's add audio to our existing game. To add audio to our game we must begin by having audio files added to our project. Unity can accept many standard audio formats including: mp3, ogg, and wav files.

Let's begin by testing our audio in our scene. Go to the Audio folder and select *RetroGameMusic*. In the Inspector on the lower right you should see a play button that let's you test the audio file. Make sure you can hear that sound file before continuing.

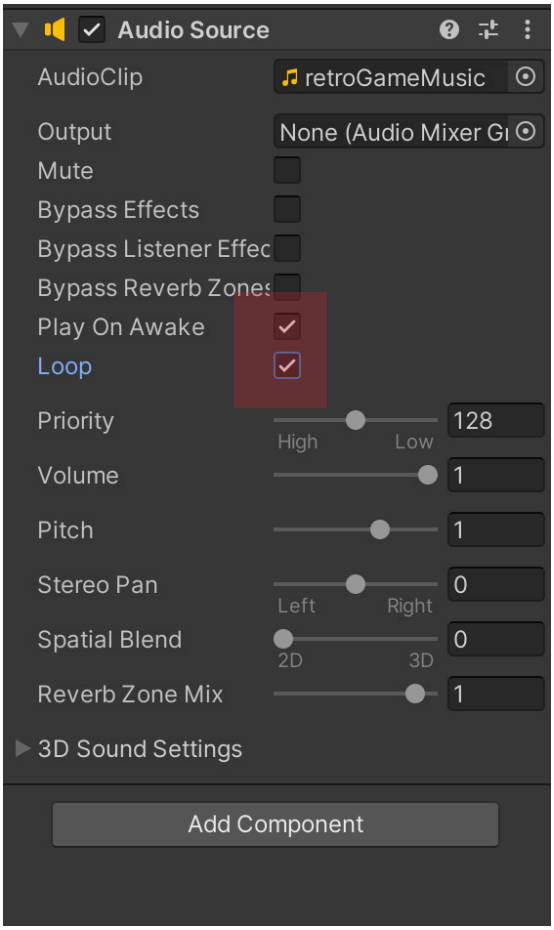
Audio Listener



The first thing we need to do before we add the music file to our game is check that our game has an Audio Listener component. The Audio Listener component is what let's the player hear sounds and music inside the game.

Typically this is attached to the camera object in the Hierarchy. The idea being since the camera is what the player sees from it helpful to be the audio source as well. Select the camera and check it has an Audio Listener attached. If it does not, check to see if it is applied elsewhere. Is not is present you can add one to the camera object by selecting the camera object and adding a new component. Choose *Audio > Audio Listener*.

Audio Source



Now that we have an audio listener, let's add an audio source. The audio source will hold our music file. The Audio Source can be applied to anything, but let's build an empty game object to hold it. The reasoning is that we'll game the game object, Audio Player. This will help us organize all of our sounds in our game

After creating the Audio Player object go to *Add Component* in the Inspector and choose Audio Source. You should see a panel that looks like the image on the right.

Drag the *RetroGameMusic* file into the slot that says *Audio Clip*. This will be the audio file to be used for this component.

You can see that in the panel there are options. Make sure the option that says Play on Awake is checked. This option means the sound file will play as soon as the game world is loaded. Since it is background music that is what we would want. We would also would want to make sure the loop checkbox is turned on as well. This would make sure the file loops over and over. Finally if needed we can use the volume option to raise or lower the volumen of the music.

Play the game and you should hear the music play upon startup.

Add Sound Effects

Now that we have music playing let's add some sound effects. We are going to use the ding sound effect when the player connects with the diamond.

Begin by adding an Audio Source to the diamond in the scene. Place *ding.mp3* into the Audio Clip slot. Unlike the background music we don't want the sound effect to *loop* or *Play on Awake*. If it played on awake it would play, even if the collision has not occured yet. So how can we cause the sound effect to play during a collision. The answer is by using C#.

Audio Script

Open *PlayerMove.cs* into a code editor. In the past we've made variables that were string, bool, int and floats. There is another variable type we can create called a *AudioSource*, it creates an audio source component.

Let's write some basic code that creates a *public AudioSource* called *AudioPlayer*. It would look like this:

```
public AudioSource AudioPlayer;
```

This will build an object variable called AudioPlayer.

Next we need to load the sound effect file into the AudioPlayer object. To do this we'll use:

```
AudioPlayer = GetComponent<AudioSource>();
```

This is a function that will go into the game object the code is applied to and retrieve the AudioSource component information. This is similar to in P5.js when we would create a variable to hold an image and then later use a function to load the image.

Finally, we need to decide when the audio file will play. Since it is a sound effect that when it collides with the player, let's use the OnTriggerCollide function. Add the following to your code:

```
void OnTriggerEnter(Collider other){

}
```

This will setup code to run upon collision with another element. The AudioPlayer variable we created is an object variable. As part of the object there are built-in functions. One of them being the *Play()* function. The *Play()* function will play the audio clip.

```
public int diamonds = 0;
public AudioSource AudioPlayer;

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

Let's add this inside the *OnTriggerEnter()*

```
void OnTriggerEnter(Collider other){
    AudioPlayer.Play();
    ...
}
```

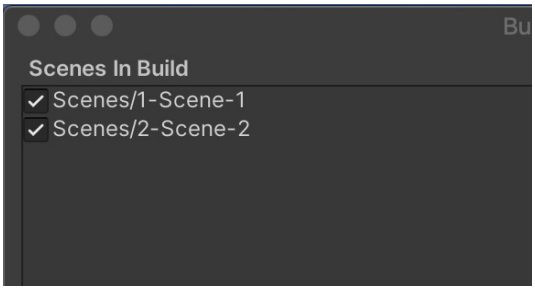
```
if (other.gameObject.CompareTag("diamond"))
{
    diamonds = diamonds + 1;
    AudioPlayer.Play();
    Destroy(other.gameObject);

}
```

Save your file and return to Unity. Play the game and collide with the diamond. You should now hear the sound effect occur upon the collision. Awesome!

Duplicate the diamond two more times using CTRL / Command + D. This will copy the copy attached to the diamond as well. Move the diamonds around the game board.

Create Levels

If you have ever played a video game before you are probably aware that there are levels. Typically the player starts on simple levels and as the game progresses the player plays harder more complex levels. We can do the same thing with Unity.



In the Unity Project open scene Level-1, then Level 2, and then Level 3. You'll notice that each scene has it's own setup. There is a player, a camera viewing the action and diamonds to collect. Let's set it up that if the player collects 3 diamonds the game will go to the next level.

Scene Manager

To have multiple scenes in our game we must set them up in the Build Settings. Go to *File > Build Settings*. You'll see a dialogue box that looks like the one on the right.

This is where you decide which scenes you want to include in your final game. To add a scene simply drag and drop them into the *Scene In Build* section.

The order of the scene go from top to bottom. The 'slot' is the index number. In programming the first index is 0. So in the image on the right Level-1 would be index 0, Level-2 would be index 1, etc.

Scene Management Code

Now that we have our levels setup in the scene manager we can use C# to go between levels. Open *PlayerMove.cs* and add the following at the top of the file.

```
using UnityEngine.SceneManagement;
```

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.SceneManagement;
5
```

This piece of code will import scene management functions for us to work with.

Next we need a variable to figure out how many diamonds we have. Create a public int variable called diamonds and set it to 0.

Next we want to set a *OnTriggerEvent* that if the player collides with a diamond it will add to the diamond variable. Add the following inside

```
diamonds = diamonds + 1;
```

Finally, the level is complete when we collect 3 diamonds. Let's write an if statement that if diamonds equals 3 then perform a block of code.

```
if (diamond >= 3){

}
```

Now let's add the *SceneManager.LoadScene('sceneName')*. This function allows you to load another scene in the project. Let's add the second level which is called Level-2:

```
if (diamond >= 3){
   SceneManager.LoadScene("Level-2");
}
```

In addition to putting the scene name you can reference the level index number.

Open scene ***Level 1***. Play the game and collect 3 diamonds. You should take you to the next level. Awesome! One problem is that to go from Level-2 to Level-3 we have to write a new script, otherwise Level-2 will link to Level-2 again. One way to fix this is instead of using a specific level name or slot would be to use code that tells Unity to go to the next level, regardless of what level or scene you are currently on.

To do this, change the code to this:

```
if (diamond >= 3){
   SceneManager.LoadScene(SceneManager.GetActiveScene().
buildIndex + 1);
}
```

This code means that if the player collects three diamonds, go to the next level. Since there this script repeats on every level, it should work and simply take you to the next level.

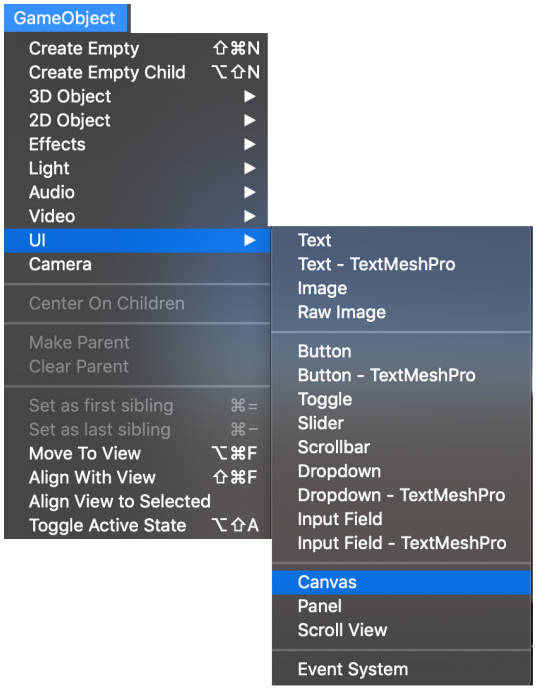
Save your code and play the game to make sure it works.

User Interface

One issue with our game is it hard to know how many diamonds we have collected. It would helpful if the game provided some feedback. This feedback is the user interface.

User interface in a game are usually 2D elements that available to the player that provide information about what is going on. The UI information could be a player's score, health bar and other helpful items. In this game we will add the logo of the game to the user interface as well as the number of diamonds that have been collected by the player.

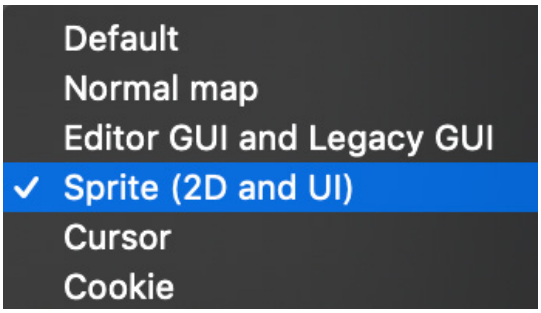
To begin building user interface go to *GameObjects > UI > Canvas*. The canvas is where we we hold our UI elements. if you zoom out of your scene you'll see white outline in the scene, that is the canvas object. To help with adjustment we will put the scene into 2D mode. This will allow us to the see the scene as a 2D space instead of a 3D space. To go into 2D mode, click the 2D button above the Scene.

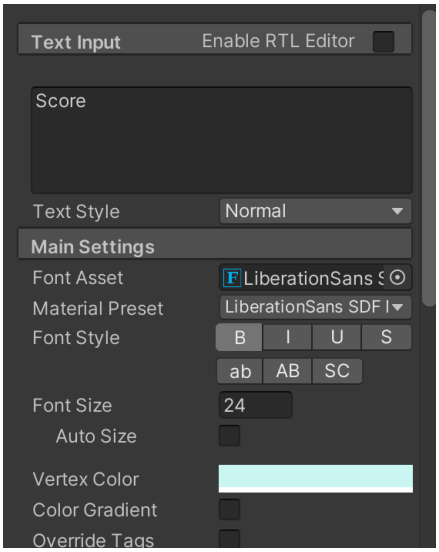


Add the Game Logo

It would nice if the logo for the game was always available in upper left hand corner. To do this we will add it to the Canvas. By going to *GameObjects > UI > Image*. This will add an image object to your canvas, you'll see a white rectangle with a small white box in your 3D space. Rename the Image in the Hierarchy to *Game Logo*. To help us arrange items we can use the rect tool. The rect tool allows us to edit, modify 2D elements.

Let's add the logo image to our UI Image. To do this we would use the source slot. Go to the *Images* folder and you'll see a file marked *Logo.png*, this will be the image to use. You'll notice that if we try to drap the image we will not be able to do it. The reason why is that the image has not been set to work with Unity's UI system. To change this select Logo.png and under Inspector and change Texture Type from *Default* to *Sprite (2D and UI)*. This will make *Logo.png* a UI sprite. Now that the image is UI Sprite we can drag it into the source slot. Use the Rect tool to position the logo at the upper left hand corner. You can use Game tab to view the actual placement of items.





Add a UI Score System

Now that we have the game logo in the upper left hand corner let's add in the score system. This will be a number that updates everytime the player collects a diamond.

To begin go to *GameObjects>UI>Text-TextMeshPro*. This will add a text string to our UI canvas. Relabel it ScoreUI in the Hierarhy While the text is in 2D model move the text to the upper left hand corner.

Use the Text Input to set the dummy text for the element. Put in the word "*Score*". We can also set the the size, font and color of the text by using the Font Asset, Font Size and Vertex Color options in the Inspector.

Add ScoreUI C#

To update the code open *PlayerMove*.cs into an editor. Begin by using the TextMeshPro library into our code by placing the following at the top:

```
using TMPro;
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using TMPro;

```

Next we need to make a text item that holds our score variable. Create a new public variable with a variable type of TextMeshProUGUI, in my example I called it *scoreText.*

```
public int diamonds = 0;
public AudioSource AudioPlayer;
public TextMeshProUGUI ScoreText;
```

Now that we have this text item we want to set the text for it. In the *Start()* add the following

```
Start(){
   ScoreText.text = "Score: " + diamonds;
}
```

This takes the text element variable we create and uses a built-in object function called *.text()* and sets to "*score*" and the number of diamonds the player has, based on the diamonds variable.

```
if (other.gameObject.CompareTag("diamond"))
{
    diamonds = diamonds + 1;
    AudioPlayer.Play();
    Destroy(other.gameObject);
    ScoreText.text = "Score: " + diamonds;
```

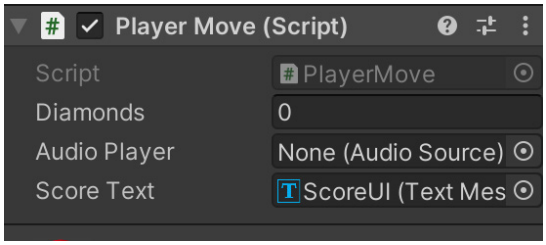
Putting this variable in the *Start()* sets up our variable at the beginning of the game, but we also need it when the player collects a new diamond.

Inside *OnTriggerEnter()* add the same code line. This will display the new updated code when the player collides with the diamond.

```
if (other.gameObject.CompareTag("diamond")) {

        Destroy(other.gameObject);
        diamonds = diamonds + 1;
        ScoreText.text = "Score: " + diamonds;

    }
```

Set the Text Variable to the UI Element

Now that we ScoreText variable creating text we need to link it to UI element we created earlier.



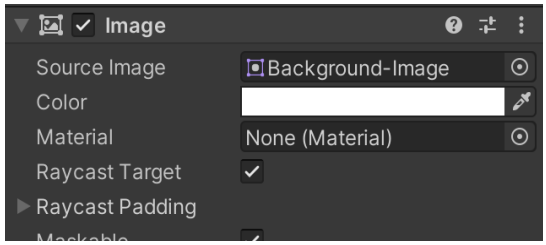
Select the player in Hierarchy and open up the *PlayerMove* script in the Inspector. You should see a slot marked ScoreText in the variables. Drag in the ScoreUI element from the Hierachy and viola! You have now set the ScoreUI element to the ScoreText variable.

Test the game and you should see the number of diamonds increase in the user interface.

Create a Start Screen

Most games when loaded do not start directly in the game, instead they have a start screen that contains options. These options include starting the game or modifying game options.

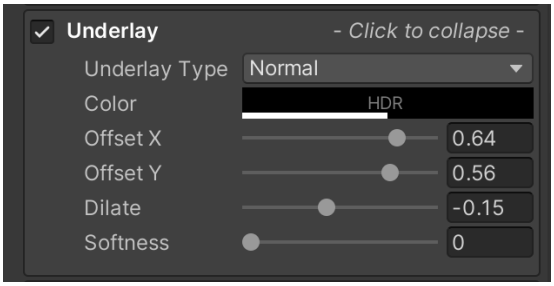
When a player wins a game many times a win screen will appear to let them know they have won and to allow them to restart the game.



Let's build a simple Start screen that will appear when the game starts.

Begin by creating a new scene. Save the scene as *StartScreen*. This menu will 2D so put your scene into 2D mode.

Begin by creating a new UI Canvas. Next we will add a user interface panel. You can do by going to *GameObject > UI > Panel*. The panel will hold the background of the menu.



Now let's add in the background image. To do this let's go to into the Images folder and select *Background-Image.jpg*. Like with the Logo we need to change the image type from Default to Sprite (2D and UI) and put into the Source Image slot.

When you preview you'll see that the background image is semi transparent. To fix this click on the Color property and change the alpha to full 255.

Add Game Name & Start Button

Let add the name of the game, RollerBall, to the start menu. Go to *GameObject>UI>Text Mesh Pro* to add text to the menu.

Go to *Text Input* and change the text to *RollerBall*. Let's also

change the font to Roboto, set the type to 80px, and use the bold option to make the text a bit thicker. To add a drop shadow to the text go to *Underlay* and click on the option "*Click to Expand*". We can then click the checkbox to turn on the shadow option. Use the *OffsetX* and *OffsetY* to move the shadow. Your game screen should look roughly like the image on the right. Feel free to make additional changes the text formatting if you like.

Next let's add a start button. The start button is what the user will press to go start the game and go to the first level. Highlight the Canvas in the Hierarchy and go to *GameObjects>UI>Button - TextMeshPro*. This will add a button to your menu. Relabel the button, Start Button in your Hierarchy.

Use the move and scale tool to place the text below the game title name. Highlight the text part and change the Text Input to Start Game. Set the font to Roboto, 50pt bold. Set the color of the text to a color of your choice.

You'll notice that the button has a rounded rectangle background. Let's turn that off. We can do that by unchecking the image checkbox. That will turn the green background off.

Awesome! The final step is opening the Build Settings again and adding the StartScreen scene as the first scene in our build.

Program the UI Button

Now let's add some code that will connect the button in our menu and have it connect to the Level-1 scene.

Let's begin by opening the *UIScript.cs* file that is in your Scripts folder into a code editor.This button will changes scenes in our game so you'll notice the using *UnityEngine.SceneManagement* at the top of our code.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine.SceneManagement;
using UnityEngine;

public class UIScripts : MonoBehaviour
{

    void Start()
    {

    }

    void Update()
    {

    }

    public void PlayGame(){
       SceneManager.LoadScene("Level-1");
    }


}
```

Let's begin by creating our own function. You may remember the process of creating our own function back in P5.js.

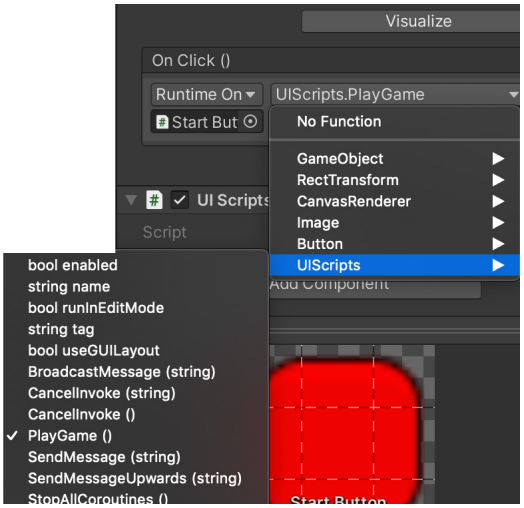
Add the following code below:

```
public void StartGame(){

}
```

The code we just wrote creates a new public function called *StartGame()*. Now that we created the function let's say when the function is called that we tell the game to load Level-1. The code would look like this:

```
public void StartGame(){
   SceneManager.LoadScene("Level-1");
}
```

Save and return to Unity.

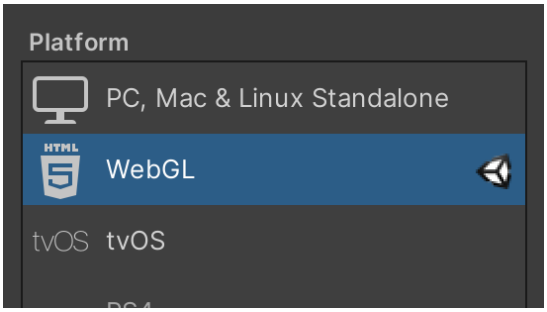


There is one last thing we need to do to make our game work, we need to add our *StartGame()* to our button. To do this begin by adding UIScripts to the Start Button in the Hierarchy. In the button go to the bottom where it says *OnClick()*. *OnClick()* allows you to attach a function to an item. Hit the "+" sign and make sure the setting is on *Runtime Only*. Drag in the *StartButton* and set the function to *UIScripts > StartGame()*.

Save and launch the *StartScreen* scene. Click on the Start Game button and it should take you to the first level.

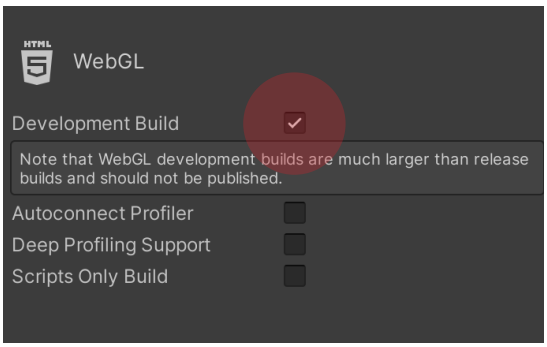
Export Your Game

Whew! That was a lot of work, but now we should have a game that begins with a start screen and when you press play will begin on Level 1. We have sounds, user interface and more. Now let's discuss exporting a working game from Unity. Unity's native file is the .unity file. Unfortantely, this file will only work if the person playing has Unity installed as well. Instead, we are going to take our project and export to a platform. A platform is method for the player to play the game, this may include playing the game on a Window / Mac computer, playing the game via an internet browser or exporting the game to an Android or iOS format.



Begin by going to *File > Build Settings*. Double check that your scenes are loaded into the game and are in the correct order.

Next we will export our game as a WebGL format. This format would allow us to export our game as a HTML and javascript and post the game to a server.



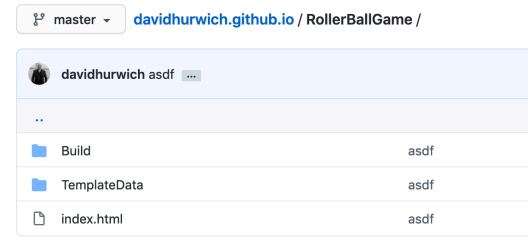
If you have not already you'll need to install the WebGL module. Download and install the file. After installing the WebGL module you will probably need to close and reopen Unity.

Go back to Build Setting and click on WebGL and click the *Choose Platform* option. You will see the Unity Logo to the right of the option.

In the options on the right make sure Development Build has been checked. Choose Build and Unity will begin to export the game, this may take a few minutes.

In the end you will have a folder that contains an HTML file called

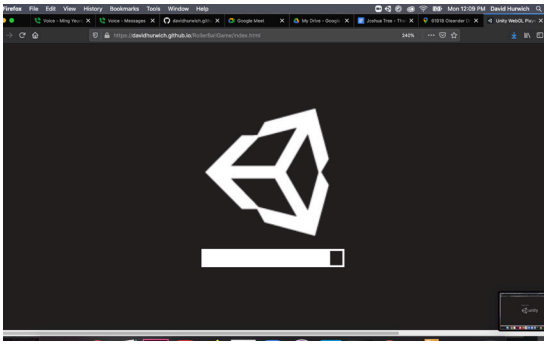
index.html and several javascript files. Do not change the order or placement of the files as they work with one another.



Finally, in order to play our game the folder containing the files must be uploaded to a server. For this lab let's upload the folder to our Github servers we worked with from the beginning of the semester. Upload the files and then take the URL to your index.html page. This will load the game into your browser.

Awesome! You know have a basic game you can play and share with other people!

Submission



Students will export the lab containing the start page, level-1,level-2 and level-3 as a WebGL game.

Students will upload the game to their Github Servers and then post a url to the game to iLearn.