Rune Raid Report Part 2

Contents

Making the Main Menu 3
- Imported Assets 3
- Backgrounds 5
- Buttons and Text 7
  - Button Code 11
  - Extra Button Design 18

Login an Register 22
- Register Code 22
  - Username 23
  - Password 24
  - Testing 26
- Login Code 29
  - Testing 31
- Meeting Success Criteria 34

Making Level 1 35
- Chest 35
- Quick Player Change 39
- Initial Level Design 40
- Changing the Level Design 47
- Making the new level design 49
- Puzzle 54
- Collecting Runes 59
- Meeting Success Criteria 64

Making Level 2 65
- Level Design 65
- Pressure Plate Puzzle Code 68
  - Testing 75
- Pressure Plate Player Code 75
  - Testing 76
- Checking Lists Are Equal 76
  - Testing 77

The Three Torches 78
Testing 81
Meeting Success Criteria 83

Development Part 2

Making the Main Menu

Before making the main menu I imported some assets from the Unity Engine Asset Store. The assets that I imported are all visual assets that will save me time as I will not have to design them myself, also they look more professional and of a higher quality than I could make.

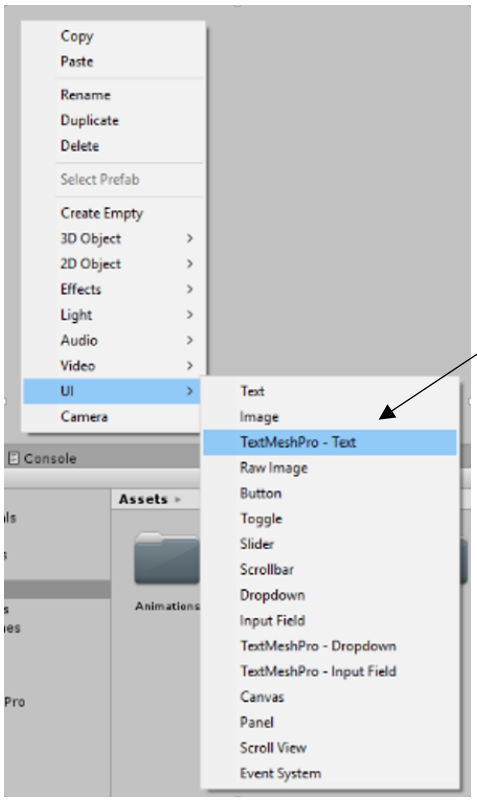
IMPORTED ASSETS

This is "Game Icons" which is contains multiple icon designs which peaked my interest. This is because they had an old/cartoon feel to them which I liked. I will not necessarily use these in Rune Raid but they might come to use depending on what I need in my levels.

This is "Platform Game Assets" which took my interest because there was some designs which I may use in Rune Raid such as the chest and gold coins. They also matched my cartoon style I want for Rune Raid.

This Asset pack "TextMeshPro" was highly recommended by many YouTubers I've watched In their Unity tutorials. Therefore, I decided to get the pack myself. It specialised in different text designs which I will mainly use for my Main Menu.

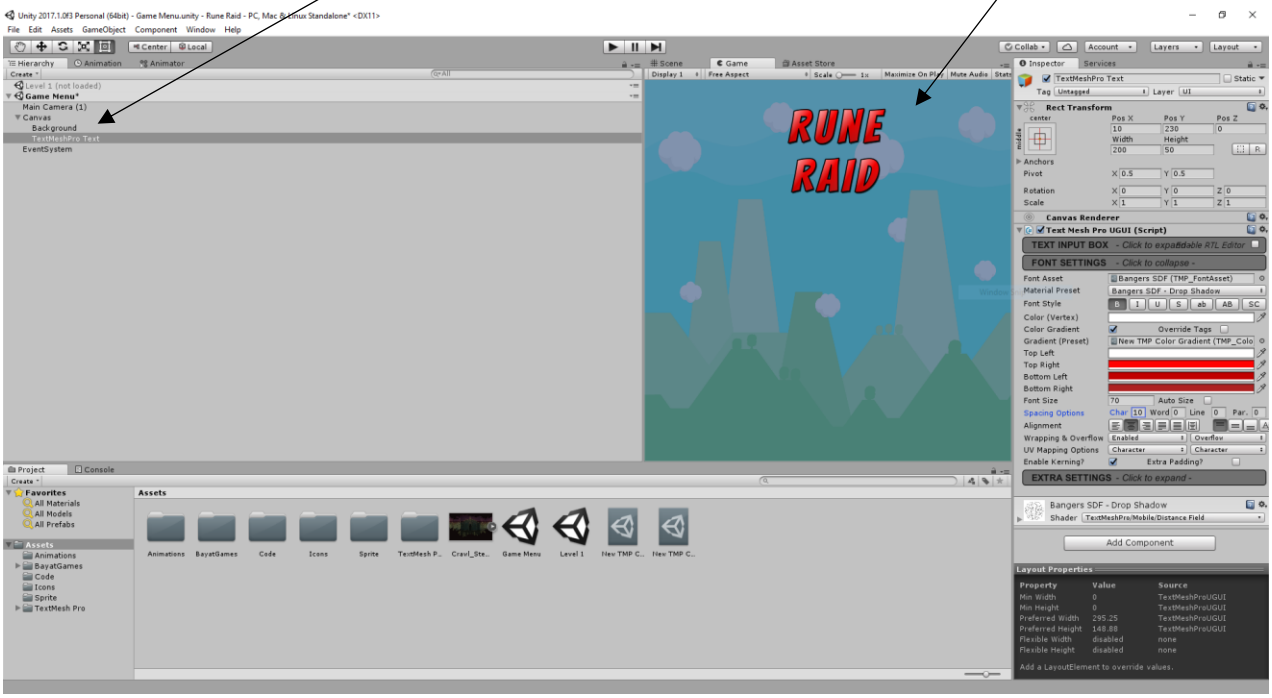
Once I imported all the assets I wanted, making the main menu mainly consists of utilising Unity's built in UIs and configuring them in the given windows. It looks like this:



There are many different UIs that Unity has to offer but for my main menu I will be using TextMeshPro – Text, Button and TextMeshPro – Input Field.

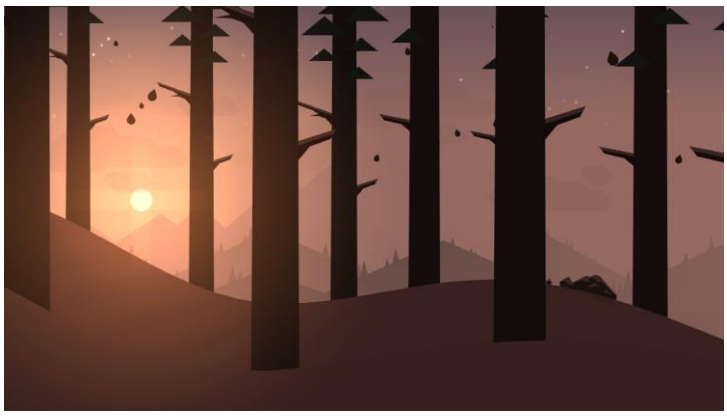
These can be easily implemented as shown on the left screenshot. Once added it creates layer in the scene open which then can be configured via the panels on the right.

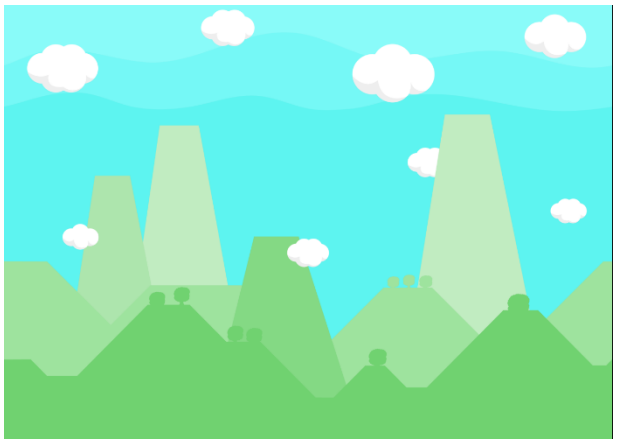
There is also a scene view which is the representation of what the scene looks and acts like. This is very useful as I can configure the layers whilst looking at the scene to see if my configurations are correct.

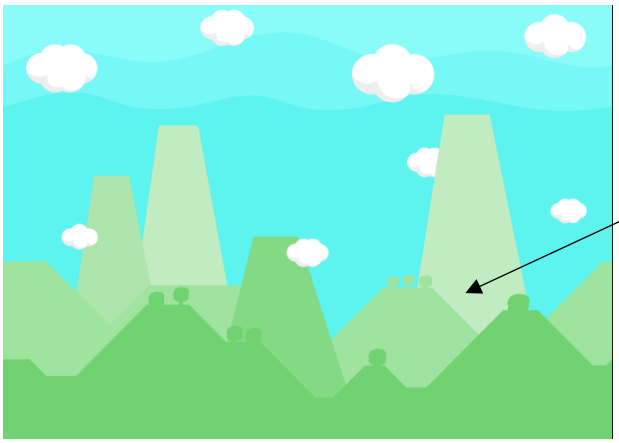


To start off I wanted to make the start screen which will contain the "Rune Raid" title in bold with 3 other buttons to interact with; Login, Leaderboard and Quit. I will also want a background image suitable for Rune Raid's genre, this will be an imported image.

BACKGROUNDS



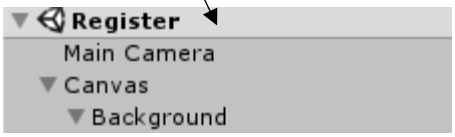


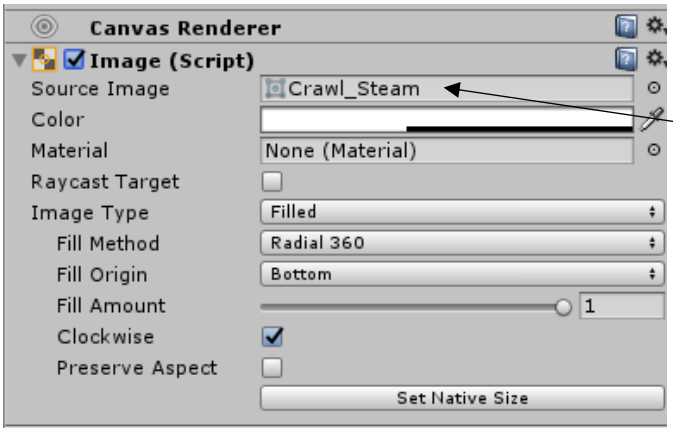


At first I decided to go with this background as it had a cartoon aspect which I liked.

However, I ended up going for this dark background because I feel it matches Rune Raid more as it is a cartoon/pixel design with dark aspects which is similar to that of a ruin.

To add a background to my main menu first a canvas and a background layer is needed which will be the layer that holds the background.



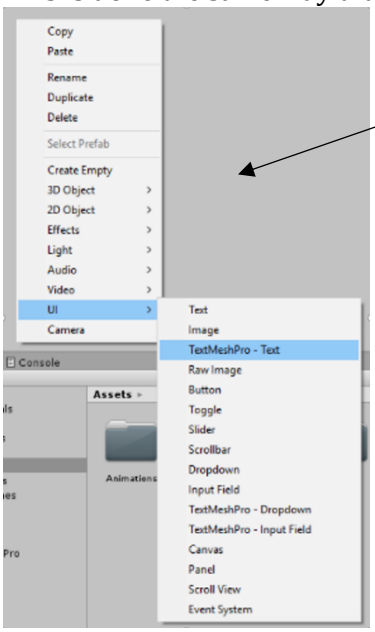


Once the canvas and background is created, in the components tab there is a source image option where the image wanted for the background is inserted, as shown in the screenshot on the left. The actual location of the background is in the "background" layer.

BUTTONS AND TEXT

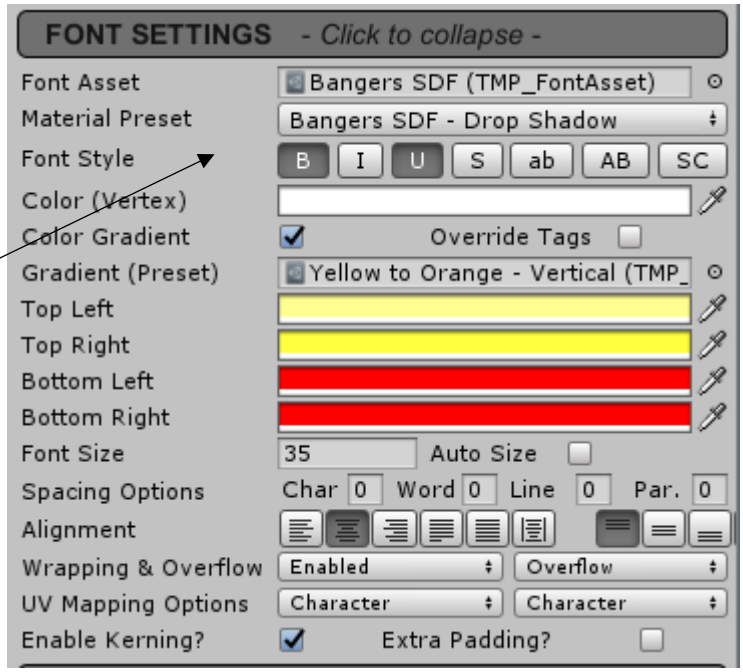
Once the background image was added the next step was to insert the game title “Rune Raid” and the 3 buttons; Login, Leaderboard and Quit.

This is done the same way that a canvas is implemented as a layer within the scene.



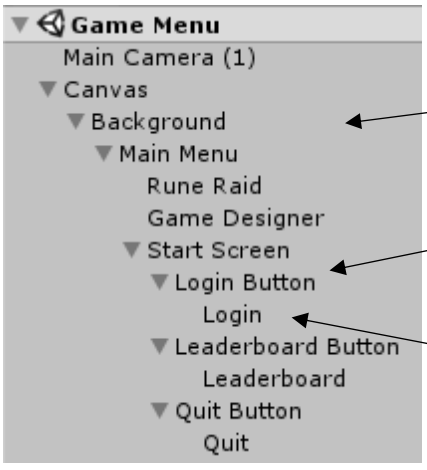
After some experimentation with all the UIs, I ended up making a start screen which I thought looked nice and fitted well with Rune Raid’s theme.

This configuration box for font settings allows any text to be customised with colour, size, font style, borders, etc. I also used the colour gradient aspect which is when the colour of the text is blended into a gradient of multiple colours.



I ended up with this as my start screen after some configurations.

I also added some text indicating the designer of Rune Raid, me Kieran Dhir.

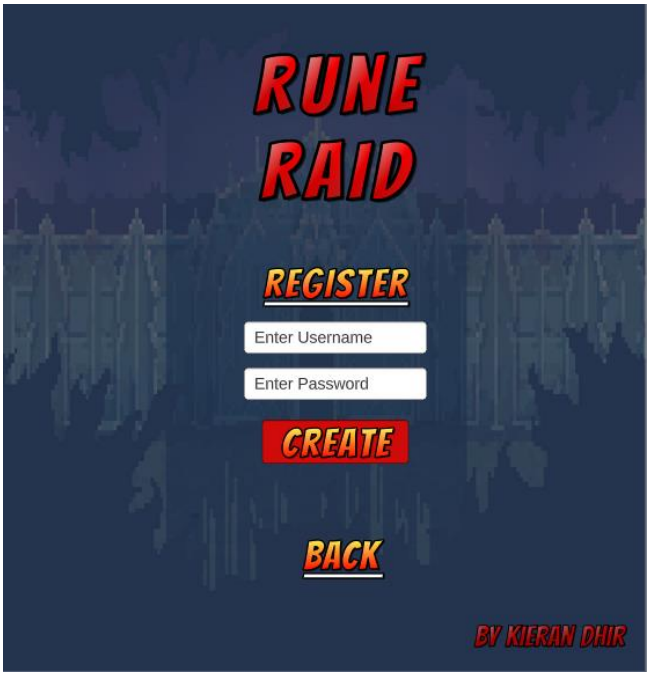


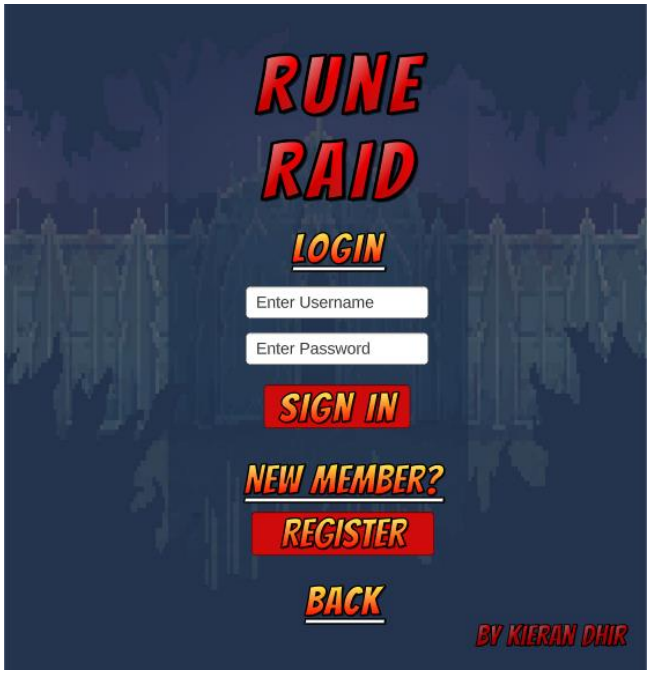
This is the layer format which makes the start screen shown above.

These are button layers

These are text

Next, I needed to create two more menu screens, which are the register screen and the login screen. It was fairly simple as the design is very similar to the start screen.

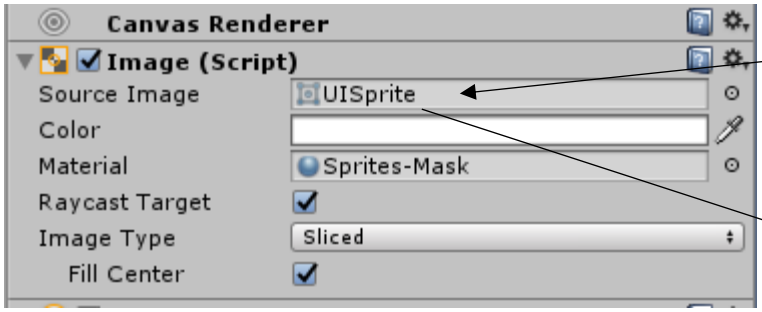




Register screen- this is where the user can create a new account for Rune Raid. It will be linked assessed via the Login screen with

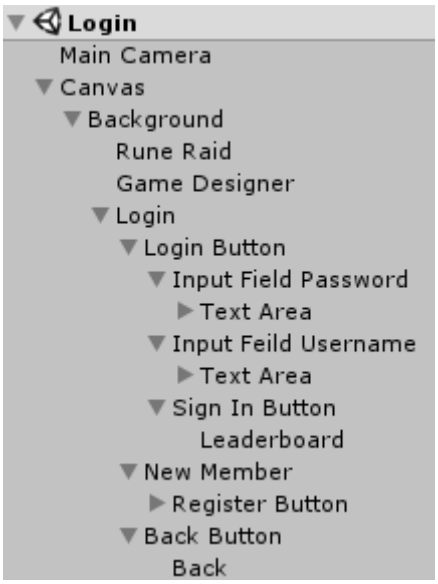
Login screen- this is where the user can sign into their account with a username and password or if they don't own an account, transition to the Register screen

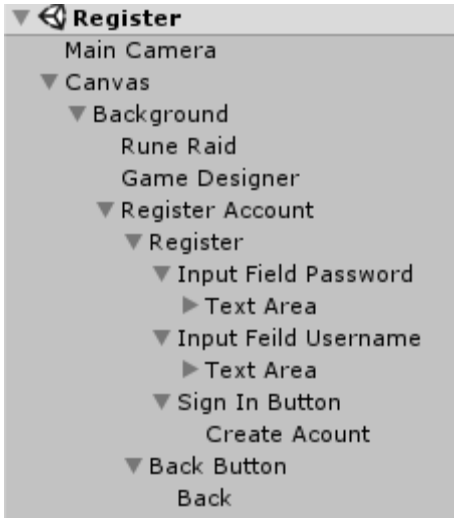
The text for "Back" and other button text are actually two layers, one for text and then once for the button. However, for some buttons I didn't want a background boarder that comes with the button. Therefore, I made the buttons invisible.

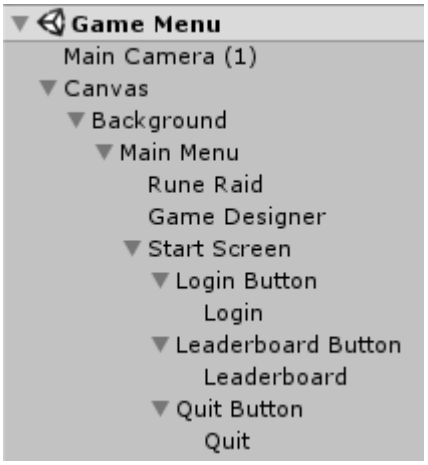


Setting the source image of the button to "UISprite" will make the button invisible.

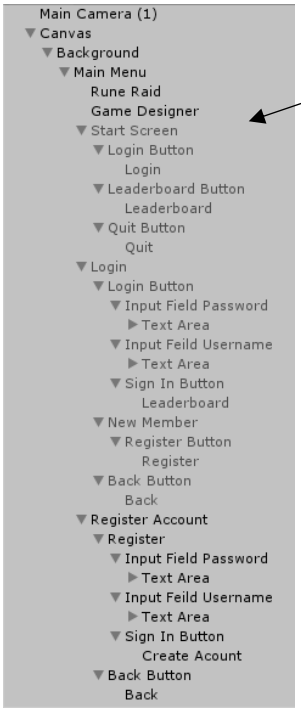
All the different screen as each there own scene within the Unity Engine for Rune Raid. This is what the layers of each scene consist of:





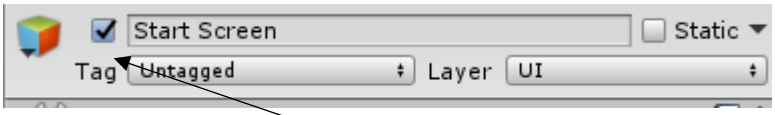


However, after some further research I found out that I didn't need for all the main menu screens to be their own scene. They could actually all be combined within one scene to make the scenes more efficient. This is done by putting them all in one scene but disabling two screen, whilst leaving the desired screen active.



This is the combined screens layer structure.

As shown, two of the main menu screens are disabled, Login and Start Screen, whilst one is active, which is the Register screen. It is shown by the faded out layers (disabled layers) and the full black layers (active layers).

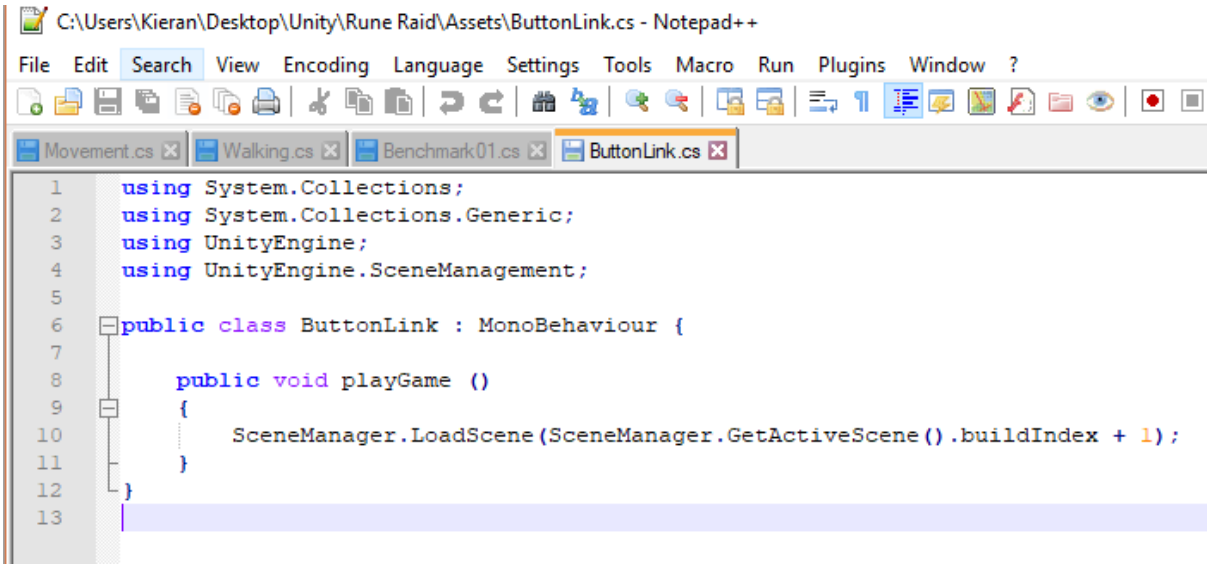


Disabling and activating the screens/layers is done by ticking or unticking this check box in the Inspector window.

Button Code

Once all the aesthetics were completed, the next stage was to implement all the links and code needed to make all the buttons intractable to the user's cursor click and to link them to the desired screen.

First, I needed to create a class where once a button is pressed, it will transition to another scene entirely. These will be the buttons "Sign In" and "Create", as if the user has entered their correct username and password or has just created an account it will load the first scene of the game.



Firstly the void start and start update needs to be removed as they are not needed for this code. as shown above the class does not contain it.

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

This line of code is implementing/ telling the class to access the Unity Engine's Scene Management which will be used within the code to change the scenes within the game. Without this, the code used to change the scenes would not work because it wouldn't have access to the Unity Scene Management which is what controls the scenes within the engine.

```
6   public class ButtonLink : MonoBehaviour {
7
```

This line is setting the class to the name of "ButtonLink", the "MonoBehaviour" is needed with every C# to set the class to be derived from the Unity Engine.

```
    public void playGame ()
```

This line of code is within the "public void" called "playGame ()". This is simply setting the code within the function to a given name, in this case "playGame"

```
    public void playGame ()
    {
        SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
    }
}
```

This line of code, once executed, will tell the Unity Engine's Screen Manager to get the scene's index that is currently active and change it by increasing the active scene index by +1. This will disable the current scene and transition to the next scene (the index that is (current + 1)).

As well as creating a function which changes the scene, I also needed a function that once pressed will close down Rune Raid, back to the user's desktop.

```
5
6   public class ButtonLink : MonoBehaviour {
7
8       public void playGame ()
9       {
10          SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
11      }
12      public void quitGame ()
13      {
14          Application.Quit();
15      }
16
17  }
```

The function is called "quitGame" and the code line 14 just closes the application (Rune Raid) when executed.

However, in the Unity Engine it doesn't show whether the application (game) closes as Rune Raid is not in its application form and therefore, will not show any signs of closing. To show that the function does work I have added and extra line of code that when the quitGame function is executed it will output "Quit" in the console log.

```
public void quitGame ()
{
    Debug.Log ("Quit");
    Application.Quit();
}
```

Now that all the classes and functions needed are created, I need to implement them into Rune Raid's buttons on the main menu and test if the code works.

To do this all that needs to be done is to add the C# script to the desired button layers, This I done in the Inspector window.

This is the "On Click" configuration box where any scripts to do with the button layers are added and configured.

This is the script used, dragging and dropping the C# script into this window is all that is needed to set the button to this script.

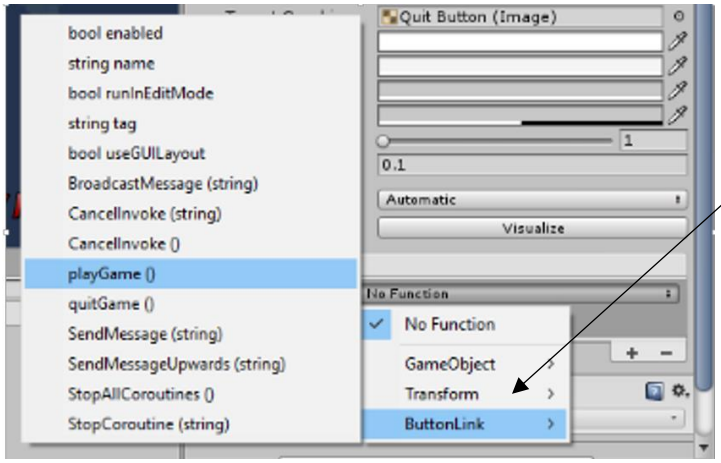
This drop down box is to set the specific function wanted. In this case it is the "ButtonLink" class with the

Now that the button Quit has had the C# script added and set to the function quitGame all that's left to do is test it in the Game window.

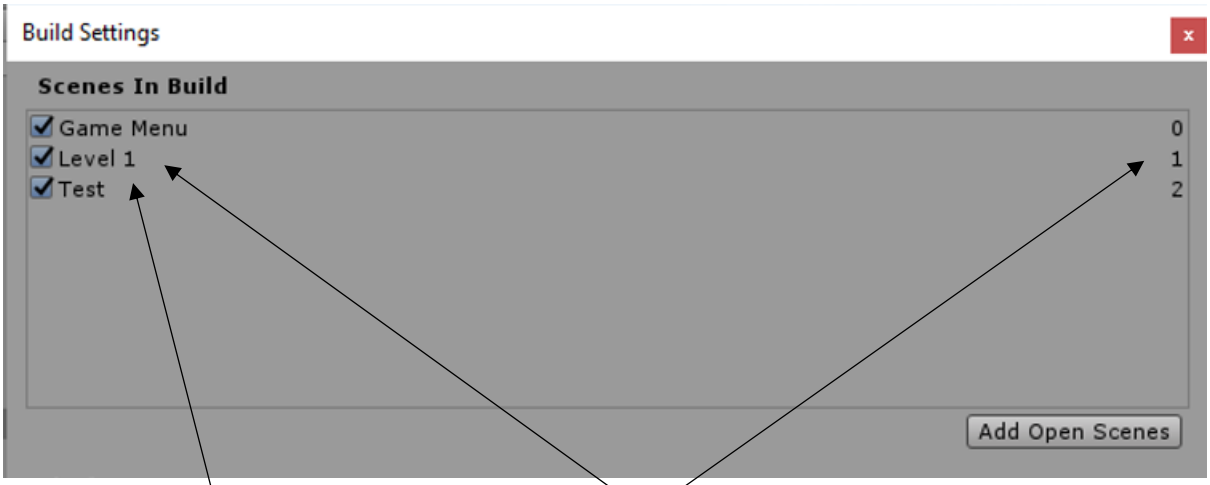
When Rune Rad is played, If the "Quit" button is pressed then it quits the application, as I said before it won't show me this but the console log does output "Quit" proving the script works and that the Quit button is linked up properly

Close up of the console log.

Next, is to do the same but with the "playGame" function instead, for the "Sign In" and "Create" buttons.



The final thing to do before testing the buttons is to set all the scenes to have a unique index. This is done in the Unity Engine's Build Settings.

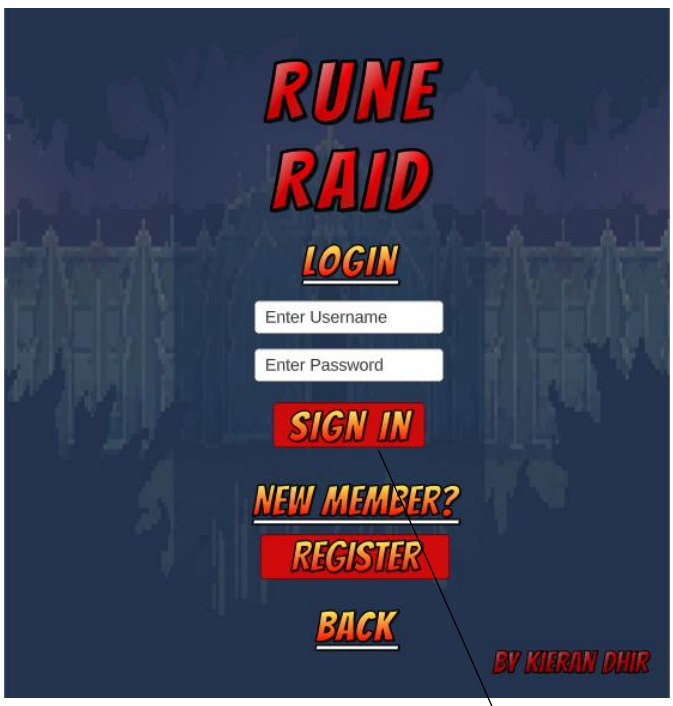


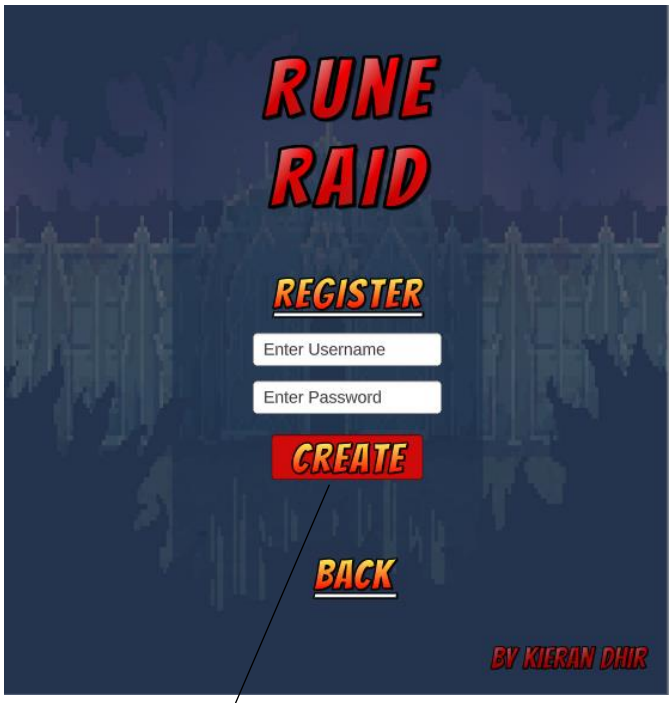
To add a scene either drag and drop the scene in the window or click the "add open scenes" and select the desired scene. Once added they are given a unique index as shown above.

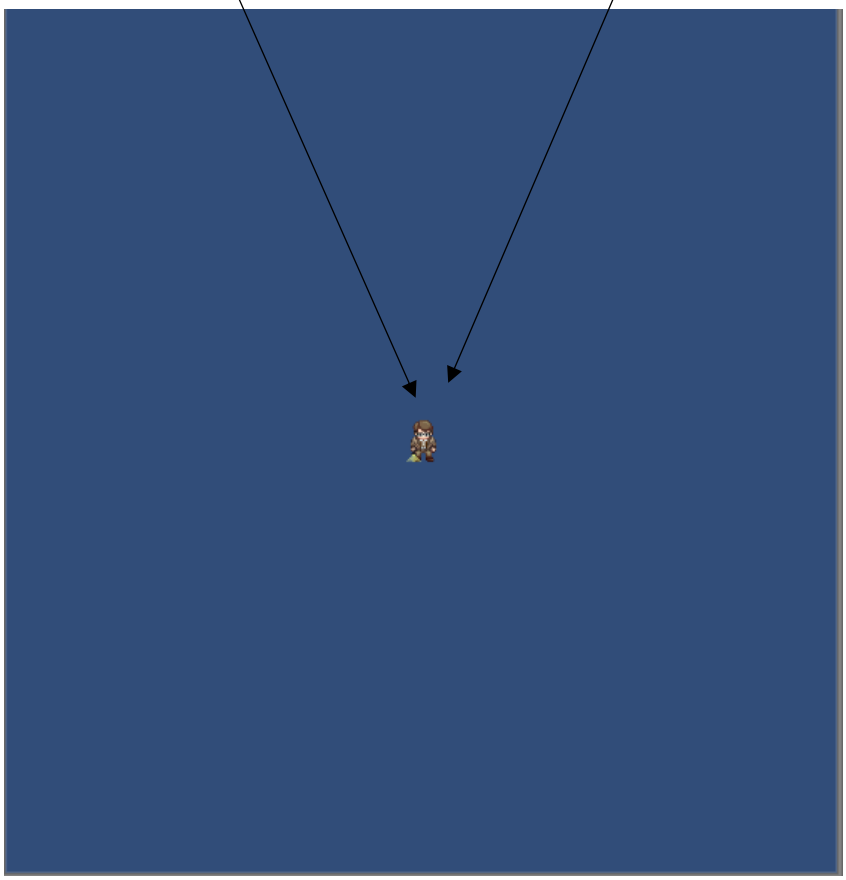
As the function "playGame" within the "ButtonLink" class C# script only changes the scene to the (current index +1) the main menu script be set to have 1 lower than the Level 1 script in order to the correct scene.

If the Test scene had the index value of 1 instead of "Level 1" then when the "Sign In" or "Create" button is pressed it will load the "Test" scene and not "Level 1" scene. Therefore, the scenes need to have the correct index values.

It is hard to show that the button does work with just screenshots but I have tested them and the “Sign In” button and the “Create” do change the scene to the Game scene (this is the scene index after Main menu) once clicked with the user cursor.



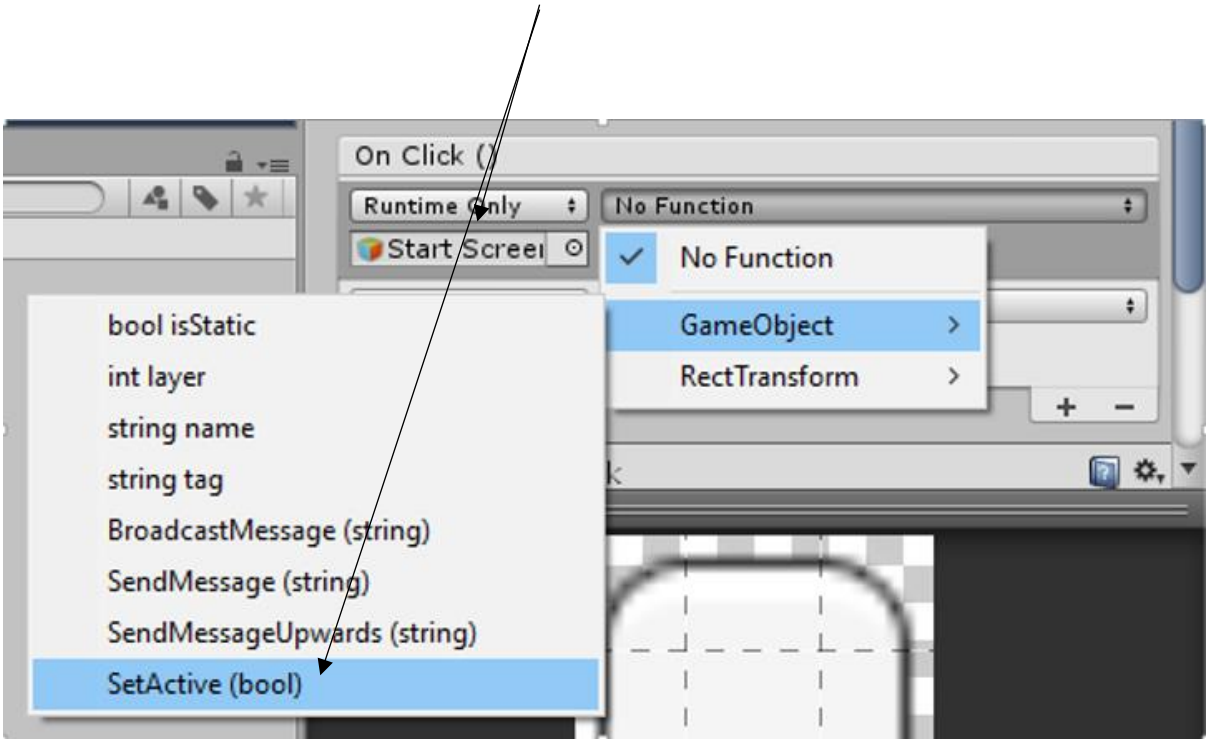




The buttons left are “Leaderboard”, “Register”, “Login” and “Back”. I will not be setting up the “Leaderboard” button just yet as I have not created the Leaderboard itself yet, the other buttons I will be setting up though.

This is done via the Inspector window within the Unity Engine and requires not extra code. The “On Click” button can also be set up in a way where it can enable and disable layers.

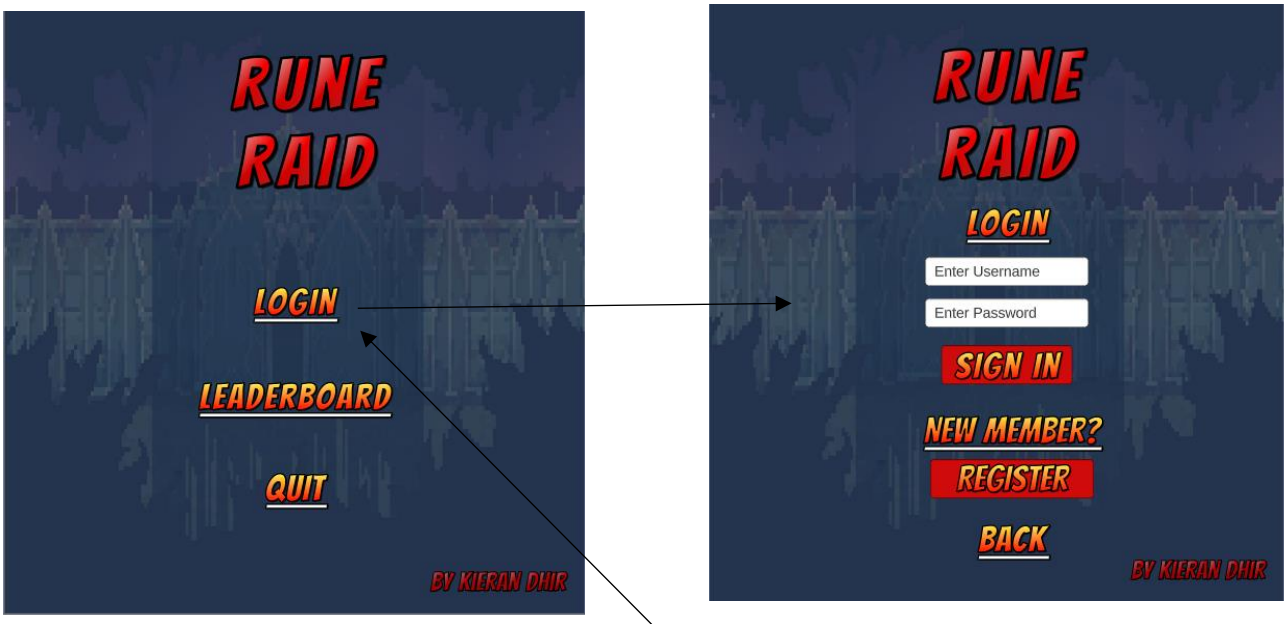
This is done by dragging and dropping the layer desired into the given box and then setting the function in the “GameObject” to “SetActive (bool)” – Meaning, setting the layer to true or false.



All that is left to do is set the check box to either active or inactive. There needs to be two or more functions because it works the way it works is that when the specific button is clicked then the layer with the check box set to active will activate and the layer with the inactive check box will disable.

In this case, when the button is clicked it will set the layer “Login” to active and set the layer “Start Screen” to inactive. Meaning the “Start Screen” will transition to the “Login” screen when the “Login” button is clicked.

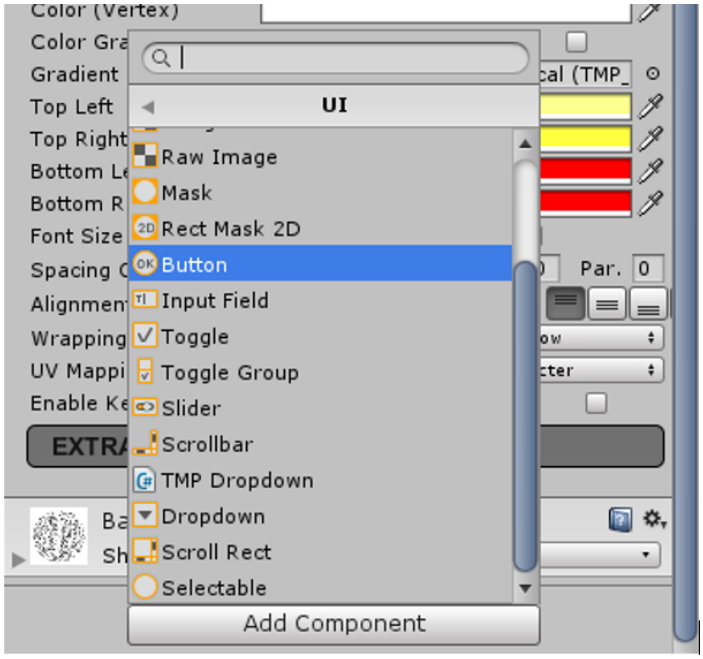
Yet again it's hard to show that the buttons do work with screenshots but I have tested the buttons and they do indeed work correctly.



Clicking the "Login" button does change the game from displaying the "Start Screen" layer to the "Login" layer.

Extra Button Design

After testing that all the buttons work (except for Leaderboard), I was doing some research to see if I could make a text layer also a button, which would simplify my layers and make them more efficient. I ended up finding out how to do this and it was quite simple.[1]

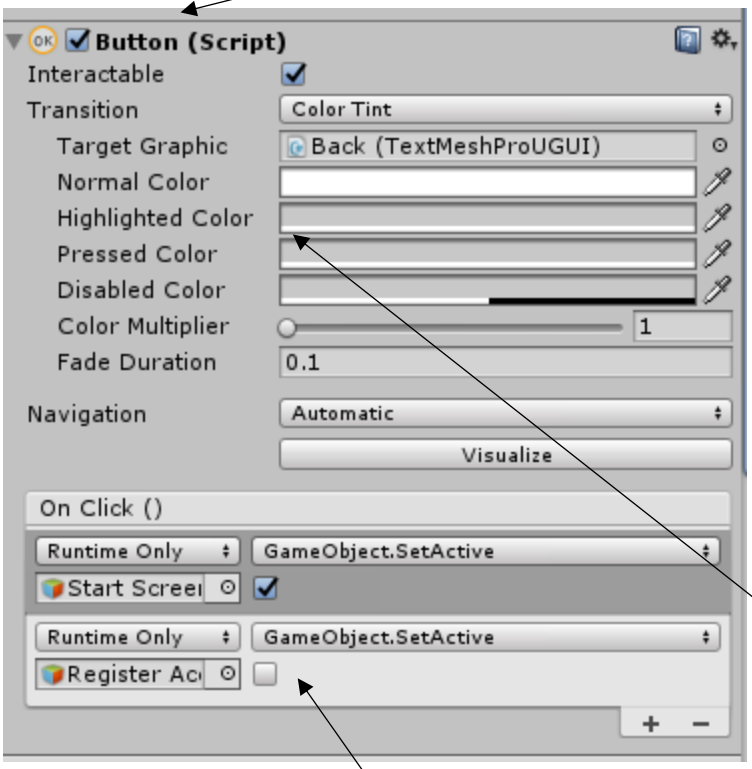


In the inspector window there is an option to "Add Component" which adds any other configuration components within the Inspector window.

I added the button component which will allow me to treat the text as a button as well.

[1] https://answers.unity.com/questions/940456/how-to-change-text-color-on-hover-in-new-gui.html - Showing text can also be a button – Date Accessed 01/03/2018

This is the box for button configuration



It also allows for customisation for what the button looks like when the user's cursor is hovered over the button and also when it is clicked.

Therefore, I decided to implement this for all my buttons as tells the user that they are indeed hovering over a button and confirms to them when they press the button because it changes colour.

This in turn will make the main menu interface more user friendly.

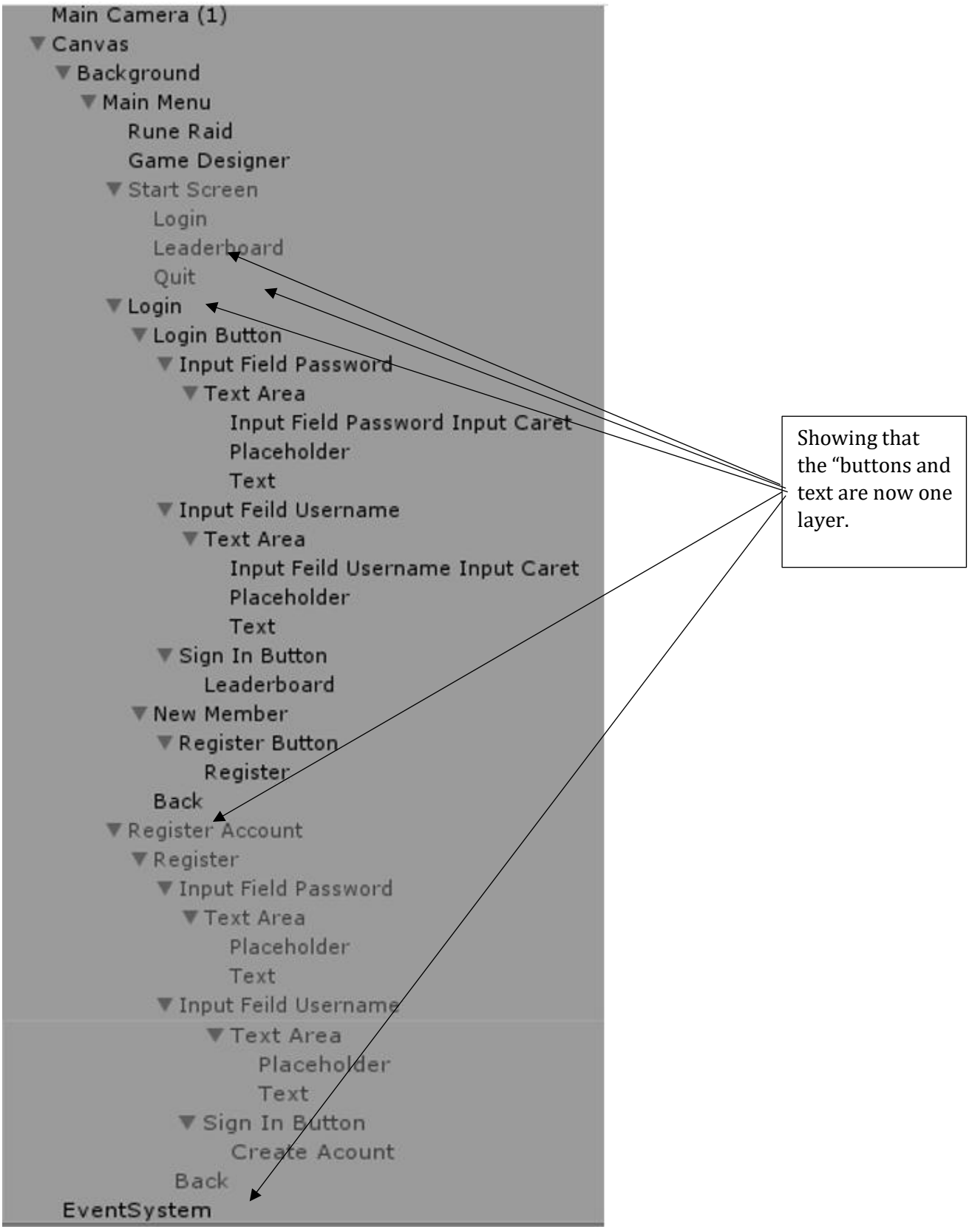
It is done by changing the "Highlighted Color" and the "Pressed Color" to a different colour than the original.

Again, I've set up the "On Click" for the text/button to the same functions as the original button had.

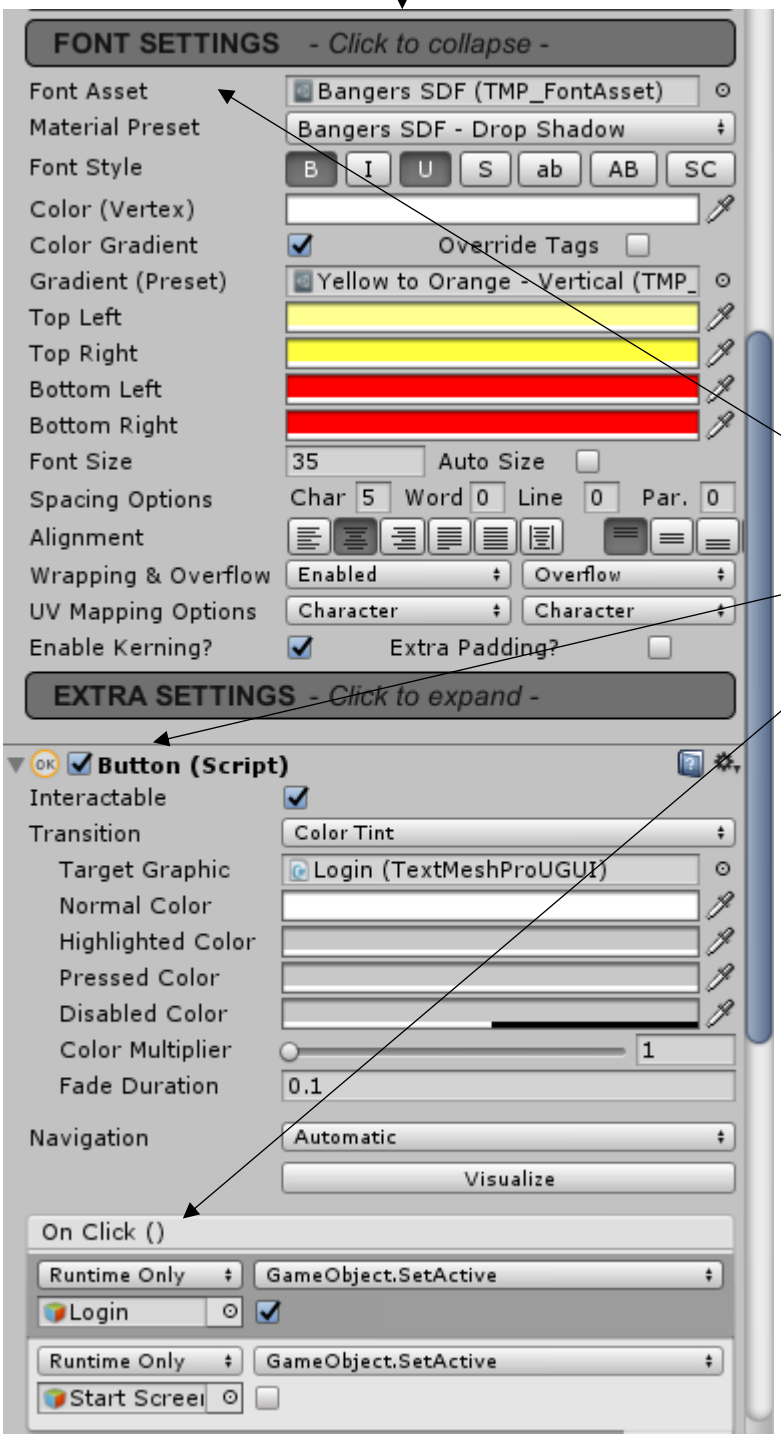
Shown below, the image on the left is the "Back" button when the user's cursor is not hovering over it or it hasn't been pressed. Shown by the colour being the original colour.

On the right, the image is the "Back" button when the user's cursor is hovering over it or it has been pressed. Show as its colour is slightly darker than the original colour.

Now that the button and the text layers have been combined into one layer the layer structure is now much more efficient and easier to understand. However some buttons and text layers have remained separate because I wanted to have a red button behind text which will change colour instead when interacted with. An example is the Register button.



Further evidence that some of the button and text layers have combined into one.



"Font Settings" component and "Button" component with "On Click" are all in the same Inspector window for a layer, which is one of the combined layers I talked

Login an Register

As I've already created the designs for both the Login and Register screen all I needed to do was to create some C# scripts to allow the user to create an account with a Username and Password, which will then be stored and used to Log back in later on.

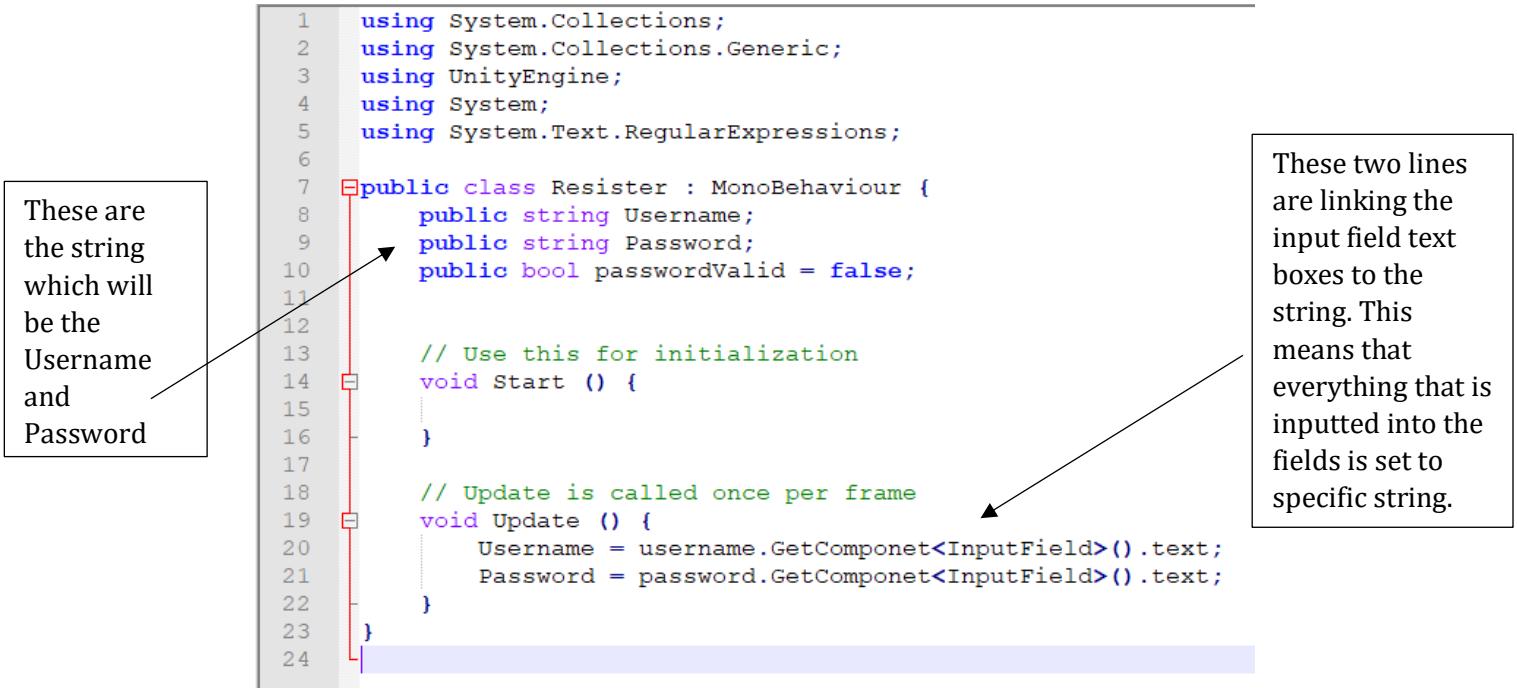
There will be a few restrictions that I will need to implement which will be:

- All Usernames must be unique.
- All passwords must be between 8 and 16 characters.
- All passwords must contain at least 1 number.

REGISTER CODE

First of I will need to find out how I will be storing the Usernames and Passwords and the way I found out was through a brilliant YouTube video that explained as way to easily store and call any data. [2]

After watching all three videos multiple times to fully understand what each line of code was doing, I started to code the system myself.




```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using System;
5  using System.Text.RegularExpressions;
6
7  public class Resister : MonoBehaviour {
8      public string Username;
9      public string Password;
10     public bool passwordValid = false;
11
12
13     // Use this for initialization
14     void Start () {
15
16     }
17
18     // Update is called once per frame
19     void Update () {
20         Username = username.GetComponet<InputField>().text;
21         Password = password.GetComponet<InputField>().text;
22     }
23 }
24
```

[2] https://www.youtube.com/watch?v=vFs0_skd0E4- Date accessed 03/04/18 – 3 part video on how to create a login and register system.

Once the two input text buttons were set to each counterpart string, I decided that because the majority of input fields allow for TAB transitions (moving across input fields with the TAB button). This is just one way to make the Login and Register more user friendly.

```
22      void Update () {
23          if (Input.GetKeyDown(KeyCode.Tab)) // if Tab key is pressed.
24          {
25              if (username.GetComponent<InputField>().isFocused) // if the username input field is selected.
26              {
27                  password.GetComponent<InputField>().Select(); // transitions to the password input field.
28              }
29          }
30          Username = username.GetComponent<InputField>().text; // assigning the input text to Username.
31          Password = password.GetComponent<InputField>().text; // assigning the input text to Password.
32      }
33  }
34
```

When the user pressed the TAB button it checks if the Username input field is also selected via line 25 and if it is then it will transition to the password input field by selecting it via line 27.

Username

```
22      void CreateButton(){
23          bool userN = false;
24          bool passW = false;
25
26          if (Username != "")
27          {
28              if (!System.IO.File.Exists(@"E:/Unity/UserData/"+Username+".txt")) // checks if username exists in the file.
29              {
30                  userN = true;
31              }
32              else
33              {
34                  Debug.LogWarning("Username is all ready taken.");
35              }
36          }
37          else
38          {
39              Debug.LogWarning("Username field empty.");
40          }
41      }
42
43
```

Creating two local bool variables to use later on.

The function createButton is a function that when executed will create a .txt file called the user's entered "Username". However, it is only created if the Username.txt will be unique.

If the Username entered is "null" then it just outputs "Username field empty." If it is now empty then it runs the next segment.

If there is a .txt file with a name identical to the user's entered Username, then it will prevent the user from creating an account with that username as all usernames must be unique. It will just output that the Username is already taken and stop until it is called again.

The way it checks if there is an existing Username.txt with the same username as the user has entered it by line 28.

"System.IO.File.Exists" checks if there is currently a .txt file located in the "UnityData" folder with the name of the entered "Username".txt. The "!" at the start just changes it so that is it runs line 30 if it doesn't exist. If it doesn't exist then the "userN" becomes true, if it does exist then it runs line 34.

```
89          if (Input.GetKeyDown(KeyCode.Return))
90          {
91              StartCoroutine("CreateButton");
92          }
```

I've coded so that when the "Enter" button is pressed it calls the "CreateButton" function.

Password

Next was to create the password code which will also be nested within the "createButton" function.

```
if (Password != "")
{
    if (Password.Length < 17 && Password.Length > 7) // checks if Password is between 8 and 16 characters.
    {
        if (Password.Any(char.IsDigit)) // checks there is an integer in the Password.
        {
            passW = true;
        }
        else
        {
            Debug.LogWarning("Password must contain at least one number.");
        }
    }
    else
    {
        Debug.LogWarning("Password must be between 8 and 16 characters.");
    }
}
else
{
    Debug.LogWarning("Password field is empty.");
}
```

I've just copied the same if statement to check first if the Password inputted into the password input field is not "null". If it is "null" then the else statement runs and outputs "Password field is empty".

If the Password is not "null" then it runs the second if statement nested within. It is checking whether the "password" is between 8 and 16 characters. "Password.Length" is an integer equal to the amount of characters of the Password string.

If the password is not between 8 and 16 characters then it outputs "Password must be between 8 and 16 characters."

If it between the range, it runs the next nested If statement, which checks if the password string contains any numbers. "password.Any(char.IsDigit)" checks if any character in the Password string is an integer and if so it sets the " passW" to equal true. If not then it outputs "Password must contain at least one number."

[3]

[3] https://stackoverflow.com/questions/18251875/in-c-how-to-check-whether-a-string-contains-an-integer - Date accessed 04/04/18 - how to check if a string contain at least one number.

```
    if (userN == true && passW == true)
    {
        if (Username == Password) // checks if the Username and Password are the same
        {
            Debug.LogWarning("Username and Password cannot be the same.");
            userN = false;
            passW = false;
        }
        else
        {
            form = (Username+"\r\n"+Password); // sets the form of how the strings are placed in the .txt file.
            System.IO.File.WriteAllText(@"E:/Unity/UserData/"+Username+".txt",form); // creating the .txt called the users "Username" file and adding in the strings.
            Debug.Log("Registration complete");
        }
    }
}
```

Once the two bool variables passW and userN are both equal to true, meaning they both pass the restrictions, then it runs this if statement.

First, it checks that the Username and Password are not the same as one last restrictions because that would make an extremely weak password. If they are equal then it sets both the bool variables back to false to allow the user to try again and enter different inputs for their username and password.

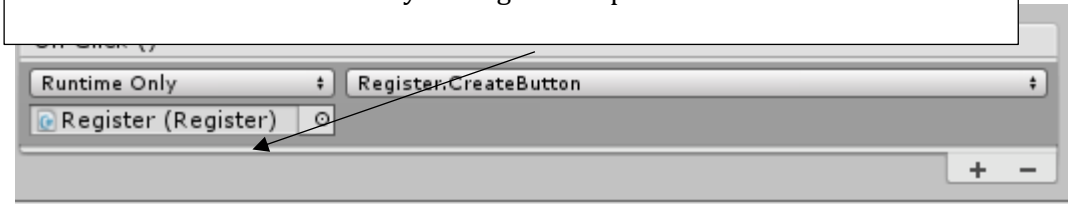
If they are not equal then it runs the else statement. The "form" string variable I've added is to create a string that will format how the Username and Password is entered into the Username.txtx file.

```
10  public class Register : MonoBehaviour {
11      public GameObject username;
12      public GameObject password;
13      private string Username;
14      private string Password;
15      private string form;
```

The "form" string formats the Username and Password so that the first line in the .txt is the Username and the second line is the Password.

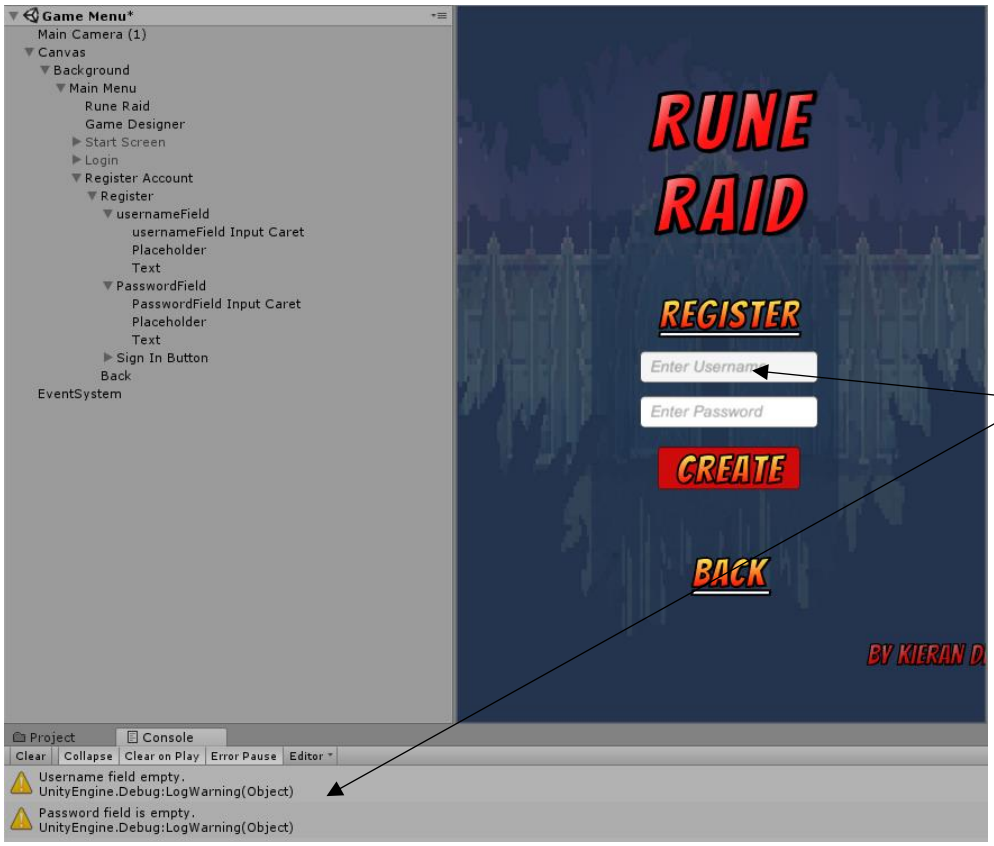
Then the Username.txt is created on the next line. "System.IO.File.WriteAllText" creates and writes all data given, in this case it's a Username.txt file created in the "UserData" file and appends the Username on the first line and the Password on the second by using the "form" variable.

The last thing to do was to make the Register button also call the "createButton" when clicked by linking the script to the button "OnClick".



Testing

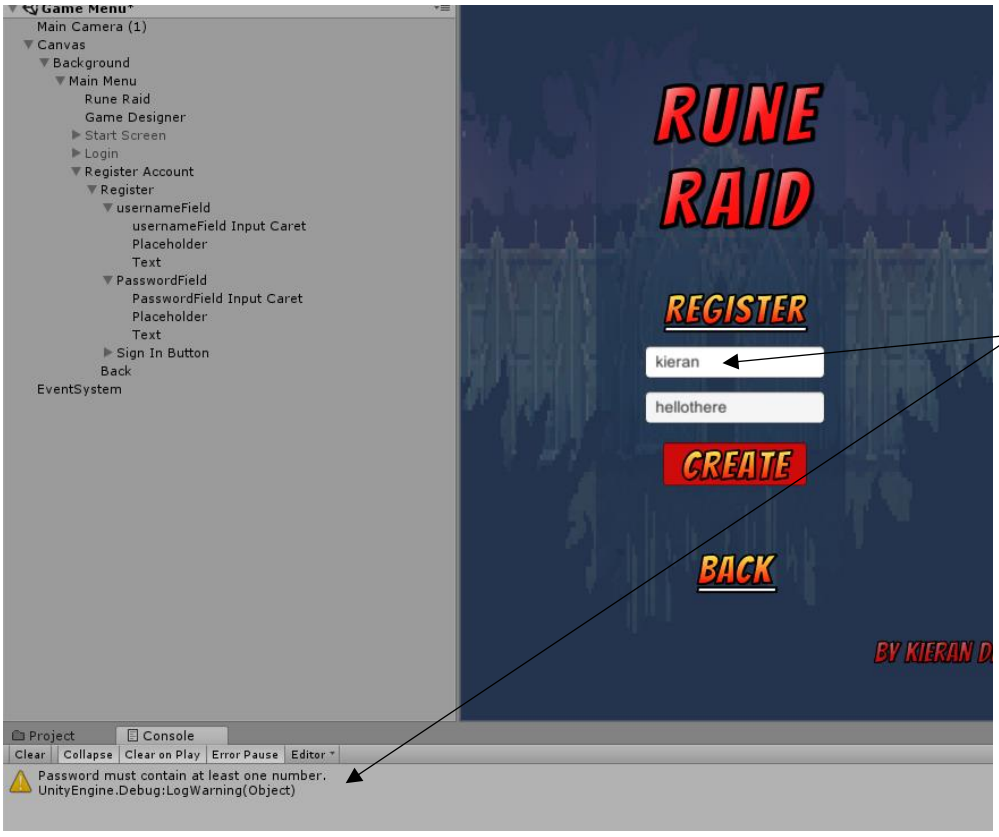
After creating all the code for registering an account I needed to test it to ensure that it all works.



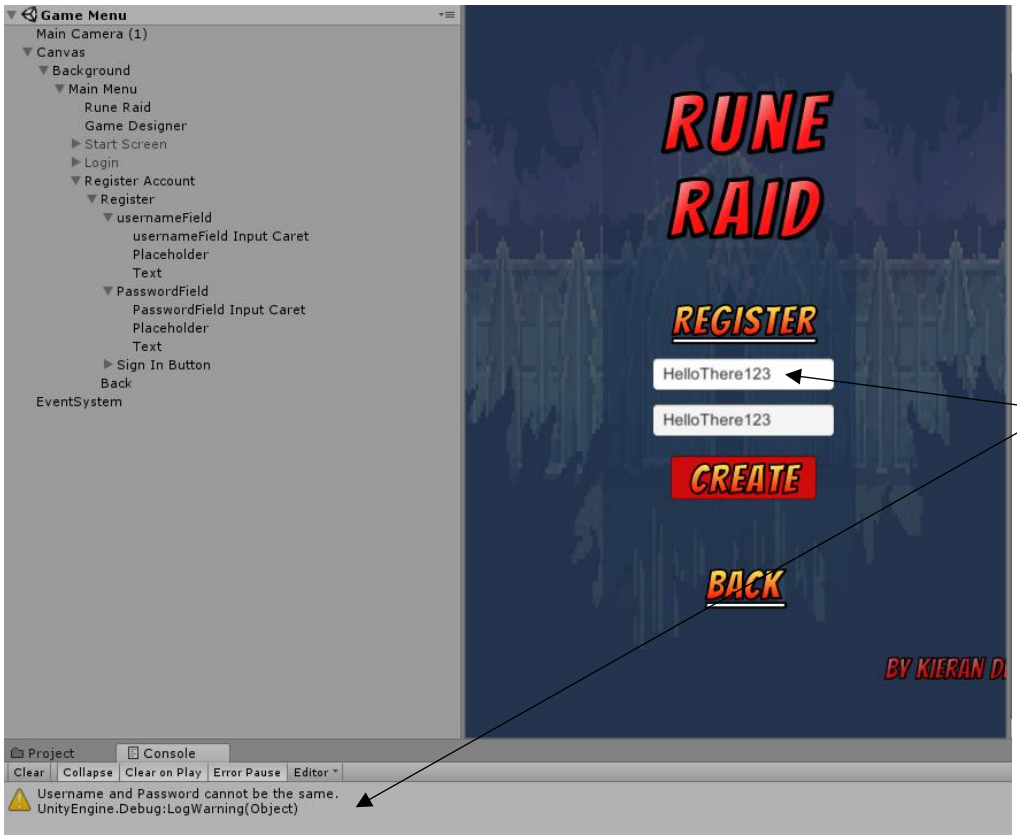
Showing that if the input fields are empty then when the user either presses the Enter button or clicks the create button, then it outputs that it is empty and prevents registration.

Showing that the Password must be between 8 and 16 characters.

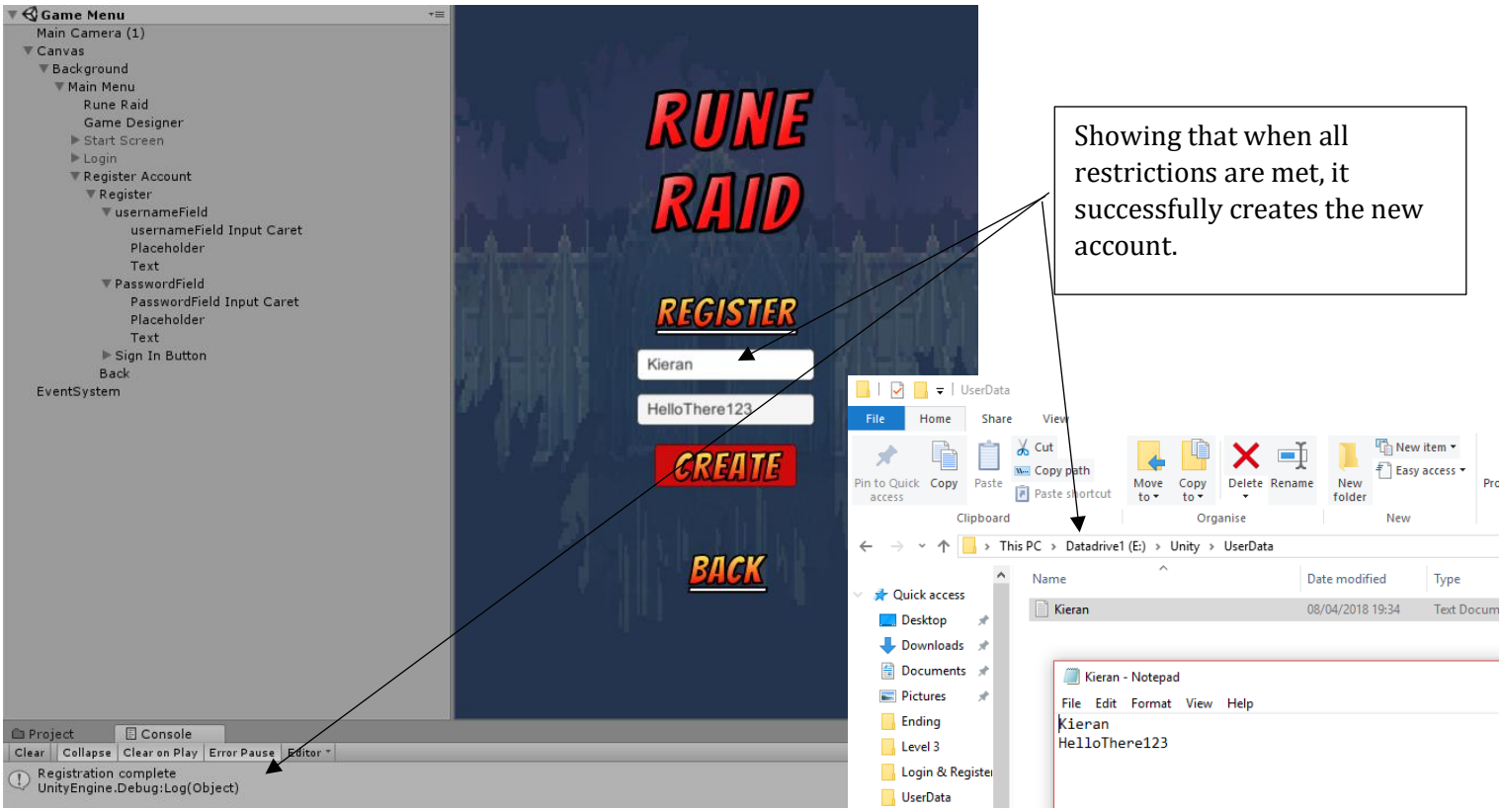
Showing that the Password cannot be longer than 16 characters.



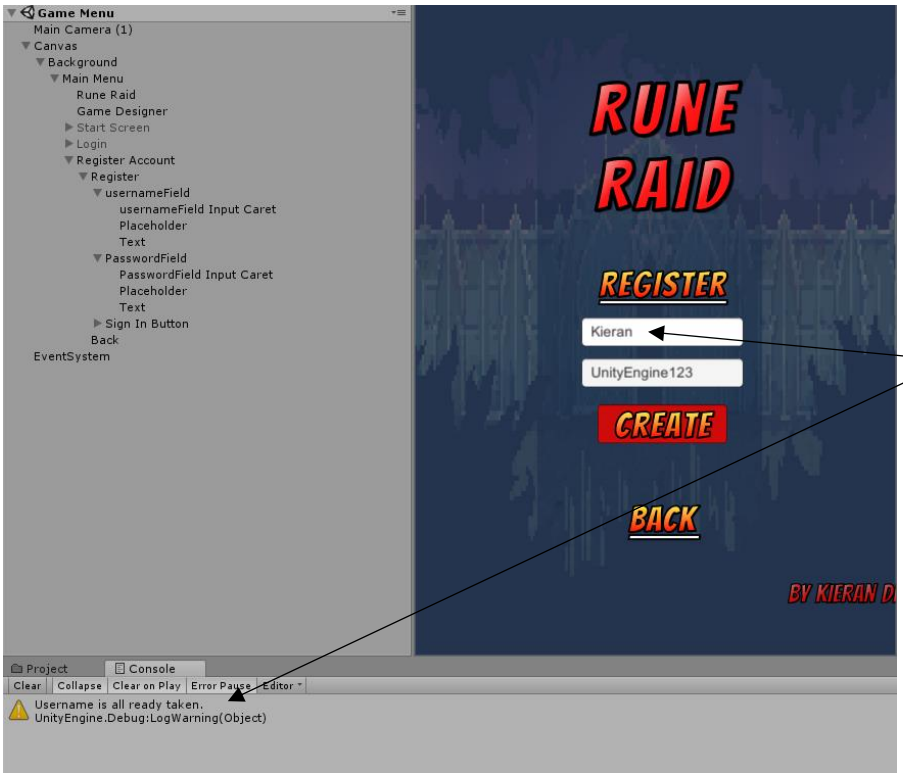
Showing that the Password must contain at least one number.



Showing that the Username and Password cannot be the same.



Showing that when all restrictions are met, it successfully creates the new account.



Showing that if another account is being made it must be unique. Here the user enters "Kieran" as a username but that is already an account username and so outputs that it is already taken.

LOGIN CODE

```
66      // Update is called once per frame
67      void Update () {
68          if (Input.GetKeyDown(KeyCode.Tab)) // if Tab key is pressed.
69          {
70              if (username.GetComponent<InputField>().isFocused) // if the username input field is selected.
71              {
72                  password.GetComponent<InputField>().Select(); // transitions to the password input field.
73              }
74          }
75          if (Input.GetKeyDown(KeyCode.Return))
76          {
77                  StartCoroutine("signInButton");
78          }
79          Username = username.GetComponent<InputField>().text; // assigning the input text to Username.
80          Password = password.GetComponent<InputField>().text; // assigning the input text to Password.
81      }
82  }
```

The code for the Login is very similar to the Register but it is just used in a slightly different way.

The Update function it identical to the Register's Update function but instead of executing the "createButton" function when the return button is pressed, it executes the "signInButton" instead.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using System;
5  using System.Text.RegularExpressions;
6  using UnityEngine.UI;
7  using System.Linq;
8  using UnityEngine.SceneManagement;
9
10 public class Login : MonoBehaviour {
11     public GameObject username;
12     public GameObject password;
13     private string Username;
14     private string Password;
15     private string[] Line; // the lines of the Username.txt
16
17
18     public void signInButton(){
19         bool userN = false;
20         bool passW = false;
21         if (Username != "")
22         {
23             if (System.IO.File.Exists(@"E:/Unity/UserData/"+Username+".txt")) // checks if Username.txt exists.
24             {
25                 userN = true;
26                 Line = System.IO.File.ReadAllLines(@"E:/Unity/UserData/"+Username+".txt"); // Reads all lines of the Username.txt and sets the lines.
27             }
28             else
29             {
30                 Debug.LogWarning("Username or Password is incorrect.");
31             }
32         }
33         else
34         {
35             Debug.LogWarning("Username field is empty.");
36         }
37
```

These are all the systems that need to be imported to allow the code to run. Without them, there would be no reference to certain library commands.

The “signInButton” function is used to check first if the user’s input is not “nul” and then if there is an existing .txt called the same as the user’s inputted Username. This is done on line 21 and 23.

If the input is “null” then it runs line 35 and outputs that the username field is empty.

If it’s not null then it checks if the inputted Username has a .txt file. If it does exists then it sets the bool variable “userN” to true and then sets the String array variable “Line” to each line as an index. Simply, it sets that the array index 0 as the first line in the .txt, the index 1 is the second and so on.

This is because Index 0 should be the Username and then the index 1 should be the Password. So now the string array “Line” will contain the username in index 0 and the password in index 2.

```
38         if (Password != "")
39         {
40             if (System.IO.File.Exists(@"E:/Unity/UserData/"+Username+".txt")) // checks if Username.txt exists.
41             {
42                 if (Line[1] == Password)
43                 {
44                     passW = true;
45                 }
46                 else
47                 {
48                     Debug.LogWarning("Username or Password is incorrect.");
49                 }
50             }
51         }
52         else
53         {
54             Debug.LogWarning("Password field is empty.");
55         }
56
```

Now it checks that the Password field is not “null”, if it is then it will output that the password field is empty. If it’s not empty then it will check if the “line” index 1 is equal to the inputted Password, as index 1 is the stored password to the username and the full account.

If the password inputted is not equal to the “Line” index 1 then the password is incorrect and outputs “Username or password is incorrect”.

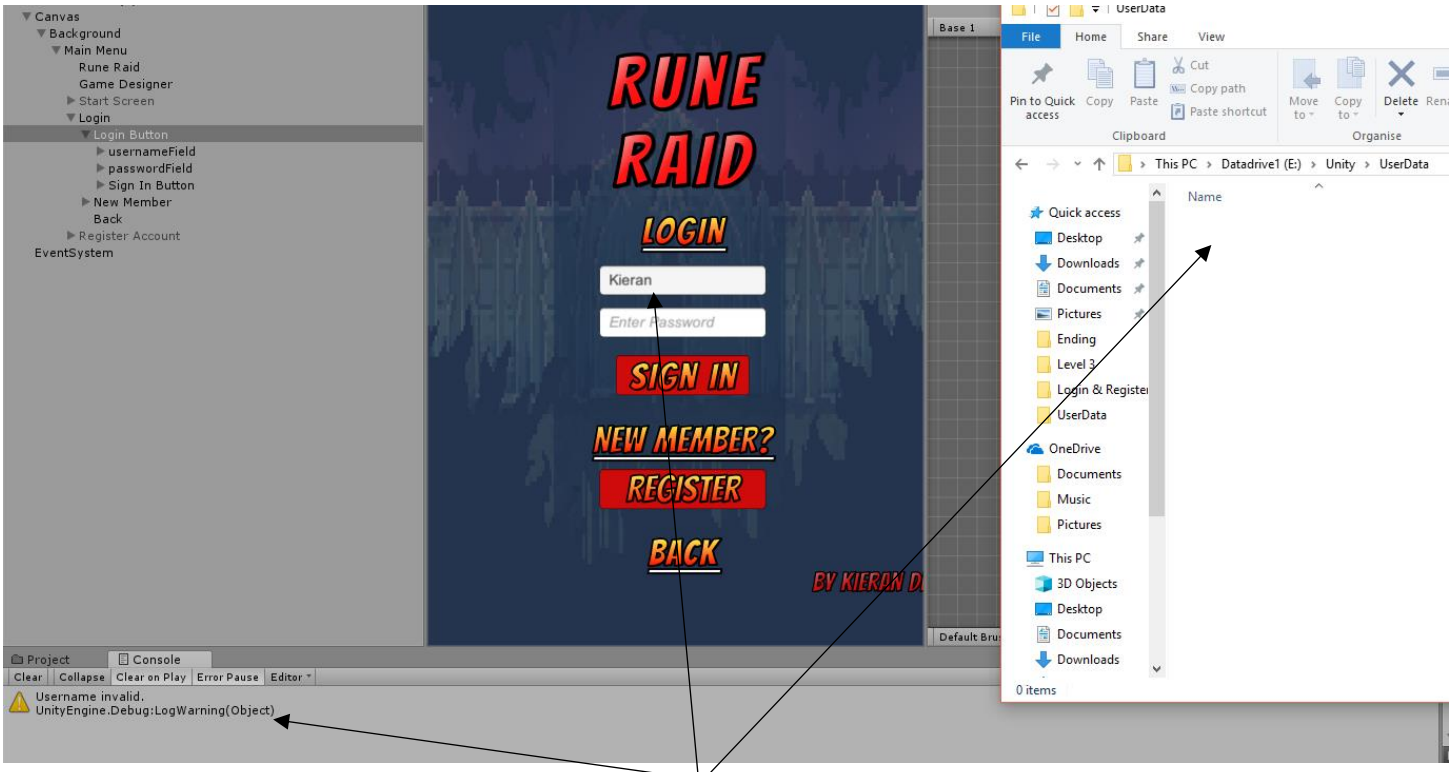
However, if the password inputted is equal to the Index 1 of “Line” then the password is correct and sets the bool “passW” to true.

```
57          if (userN == true && passW == true)
58          {
59              Debug.LogWarning("Login successful.");
60              SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1); // transitions to next scene.
61
62          }
63
64      }
```

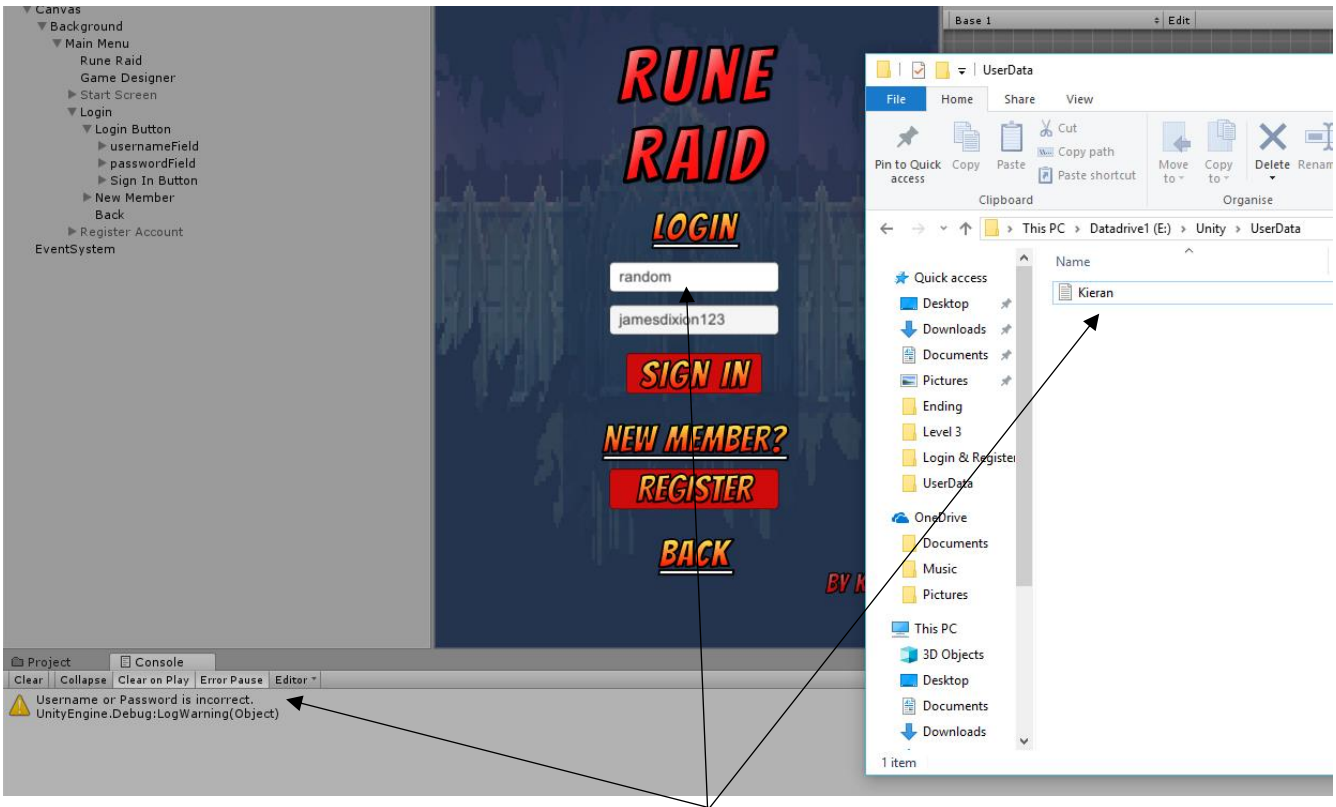
Once both bool variables are true then it runs line 59 and 60 which outputs “Login Successful” and changes the scene in build setting to +1. This is loading the opening scene.

Testing

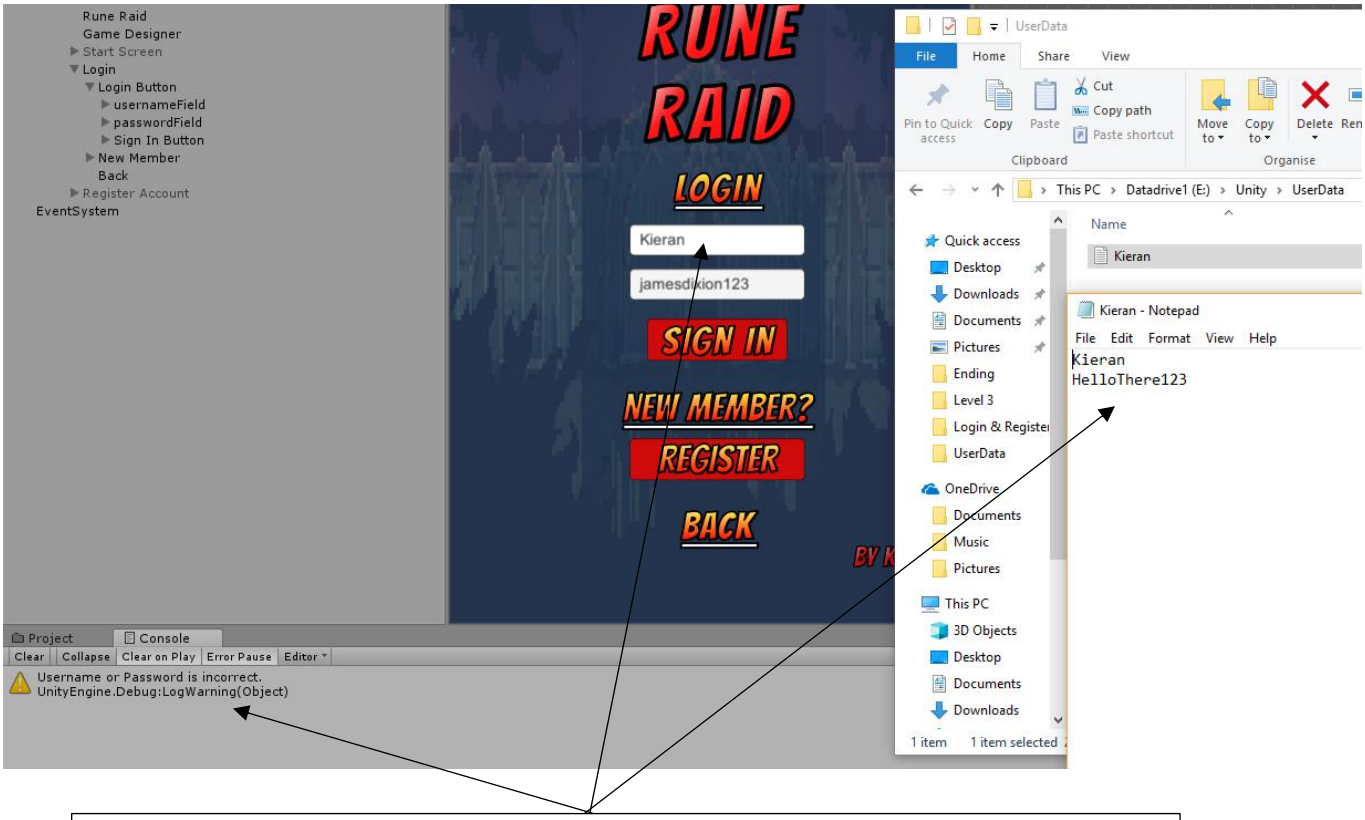
Showing that if the Username field is null then it outputs that the Username field is empty.



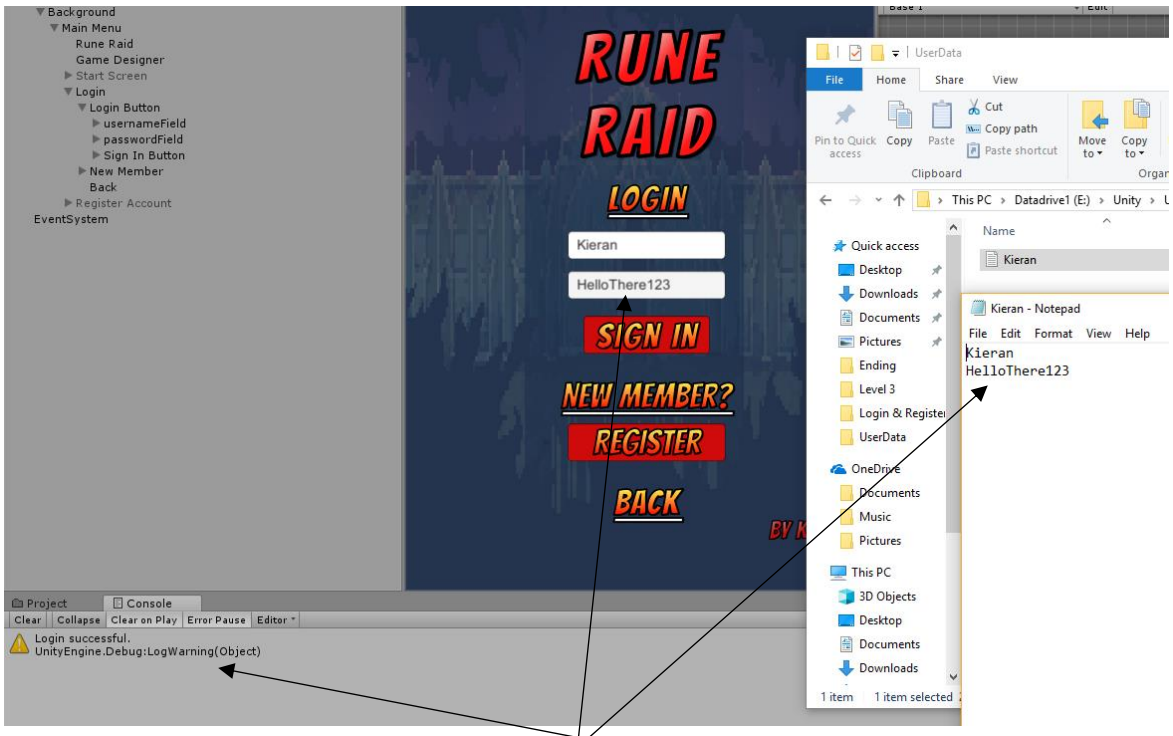
Showing that if there is no existing .txt file located in "UnityData" called the same as the inputted Username, then it outputs that the Username is invalid.



Showing that if the Username inputted doesn't match any .txt files located in the "UnityData" folder then it outputs username or password is incorrect.



Showing that if the Username inputted matches a .txt file then it still needs to have to correct password to login. Here the Username exists but the password doesn't match.



Showing that if the Username inputted exists as a .txt file and the password inputted matches the password stored then it outputs "Login successful" and then loads the next scene.

MEETING SUCCESS CRITERIA

I am confident that for the Login and Register I've met all the success criteria relevant to this section of development.

- Documentation of testing – I have clearly shown thorough testing for in range, extremes and boundary data.

- Documentation of errors – This wasn't relevant for this part of development because I just watched the YouTube video on how to code this segment and so I didn't come across any code errors.

- The player will be able to fully interact with all of Rune Raid's aspects – I've made many buttons interactive so that the user can utilise them fully by linking them to specific game objects.

- All algorithms will be as efficient as I can possibly make them – For this, as I watched a YouTube video on how to code this segment, the code was already very efficient and any extra parts I added were also to the same standard.

- All decisions shown and justified – I feel wherever I made a relevant decision I documented it and also gave my reason for it.

- All the code is split up into modules – I've definitely done this as within my scripts there are any separate functions that call each other to allow for easy debugging and allow someone else to easily understand what each part of code does. Also everything is not piled into one script, each different module has its own C# script.

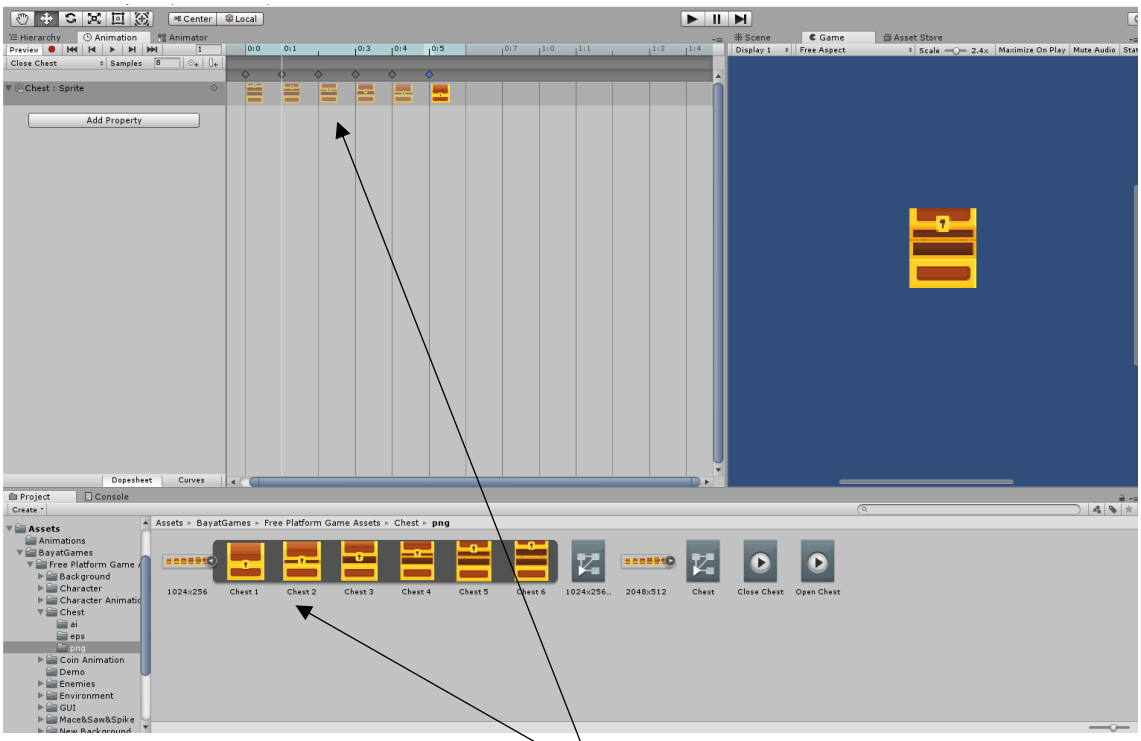
- A user friendly login interface – I believe that the Login and register interface that I've created is well presented and easy to use, such as adding in interactive big buttons, clear text and also extra things such as the return button and TAB button to work as an extra interactive tool to manipulate the interface.

Making Level 1

CHEST

First thing I wanted to do was sort out any animations, this is because I already know how to do this as I've already done it for the player movement.

I envisioned the first level to have some sort of chest that will open up when the Player gets into a certain range and presses a specific button.



This is one of the imported assets from the Unity Store which happens to contain a chest with animation positions. Therefore, it was very easy for me to animate this chest in the Animation section of Unity.

Movement.cs | Walking.cs | ButtonLink.cs | openChest.cs

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class openChest : MonoBehaviour {
6
7      public Animator chest;
8
9      private Trigger chestZone;
10
11     // Use this for initialization
12     void Start () {
13         chest = GetComponent<Animator>();
14     }
15
16     // Update is called once per frame
17     void Update () {
18         if(Input.GetKeyDown("e"))
19         {
20             chest.Play("Open Chest");
21         }
22         if(Input.GetKeyDown("r"))
23         {
24             chest.Play("Close Chest");
25         }
26     }
27 }
```

Next, was to create some code for the Chest GameObject.

First, the code sets the "Animator" to the name "chest" this is to allow the Animator part of unity to be called at any time within the code though using "chest".

Then the Start function sets the "chest" to the Animator component in the Unity Engine by getting the component.

Then the Update function checks for when the button "e" or "r" is pressed down. If "e" is pressed then it plays the animation "Open Chest" on line 20 and if "r" is pressed it plays the "Close Chest" on line 24.

```
1
2   using System.Collections;
3   using System.Collections.Generic;
4   using UnityEngine;
5
6   public class testZone : MonoBehaviour {
7       Rigidbody2D rb;
8       public Animator chest;
9
10      void Start (){
11          rb = GetComponent<Rigidbody2D>();
12          chest = GetComponent<Animator>();
13      }
14
15      void OnTriggerEnter2D (Collider2D col)
16      {
17          if (col.gameObject.name.Equals("Player"))
18              chest.Play("Open Chest");
19
20          }
21
22      }
23
24      void OnTriggerExit2D (Collider2D col)
25      {
26          if (col.gameObject.name.Equals("Player"))
27              chest.Play("Close Chest");
28      }
29  }
30
```

Now I wanted to create an area that when the player entered, it would open the chest and when they left the area the chest would close.

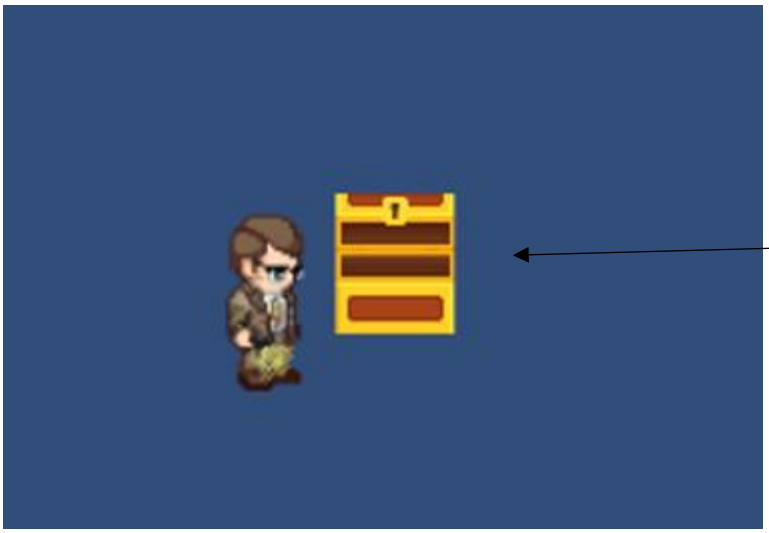
The way to do this is to add a 2D collider box to the Chest game object and then in the code user "OnTriggerEnter2D" and "OnTriggerExit2D".

"OnTriggerEnter2D" is called when a GameObject enters the 2D collider area, the "OnTriggerExit2D" is called when a GameObject leaves the 2D collider area.

[4]

Then within these "OnTrigger" functions there is each one if statement which checks if the GameObject interacting with the 2D collider is called "Player" which is the player sprite GameObject.

If the GameObject interacting with the collider is the Gamebject "Player" then it executes the next line. For "OnTriggerEnter2D" then if the "Player" GameObject enters the 2D collider's area then it plays the "Open Chest" animation. For "OnTriggerExit2D" then if the "Player" GameObject leaves the 2D collider's area then it plays the "Close Chest" animation.



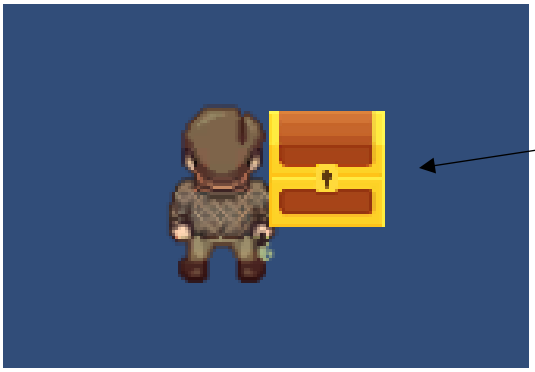
As shown, when the Player GameObject walks in the 2D collider area then the chest open animation plays.

[4] https://docs.unity3d.com/ScriptReference/Collider.html - Date accessed 05/04/18 - OnTigger functions

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

public class testZone : MonoBehaviour {

    public Animator chest;

    void Start (){
        chest = GetComponent<Animator>();
    }

    void OnTriggerEnter2D (Collider2D col)
    {
        if (col.gameObject.name.Equals("Player"))
            Debug.Log("Open Chest");

        }
    void OnTriggerStay2D (Collider2D col)
    {
        if (Input.GetKeyDown("e"))
            chest.Play("Open Chest");

        if (Input.GetKeyDown("r"))
            chest.Play("Close Chest");
    }
```

As “OnTiggerEnter” and “OnTiggerExit” only executes when the Player enter and exits the collider area, I needed to add the “OnTriggerStay” function as well. The “OnTiggerStay” function is called every frame.

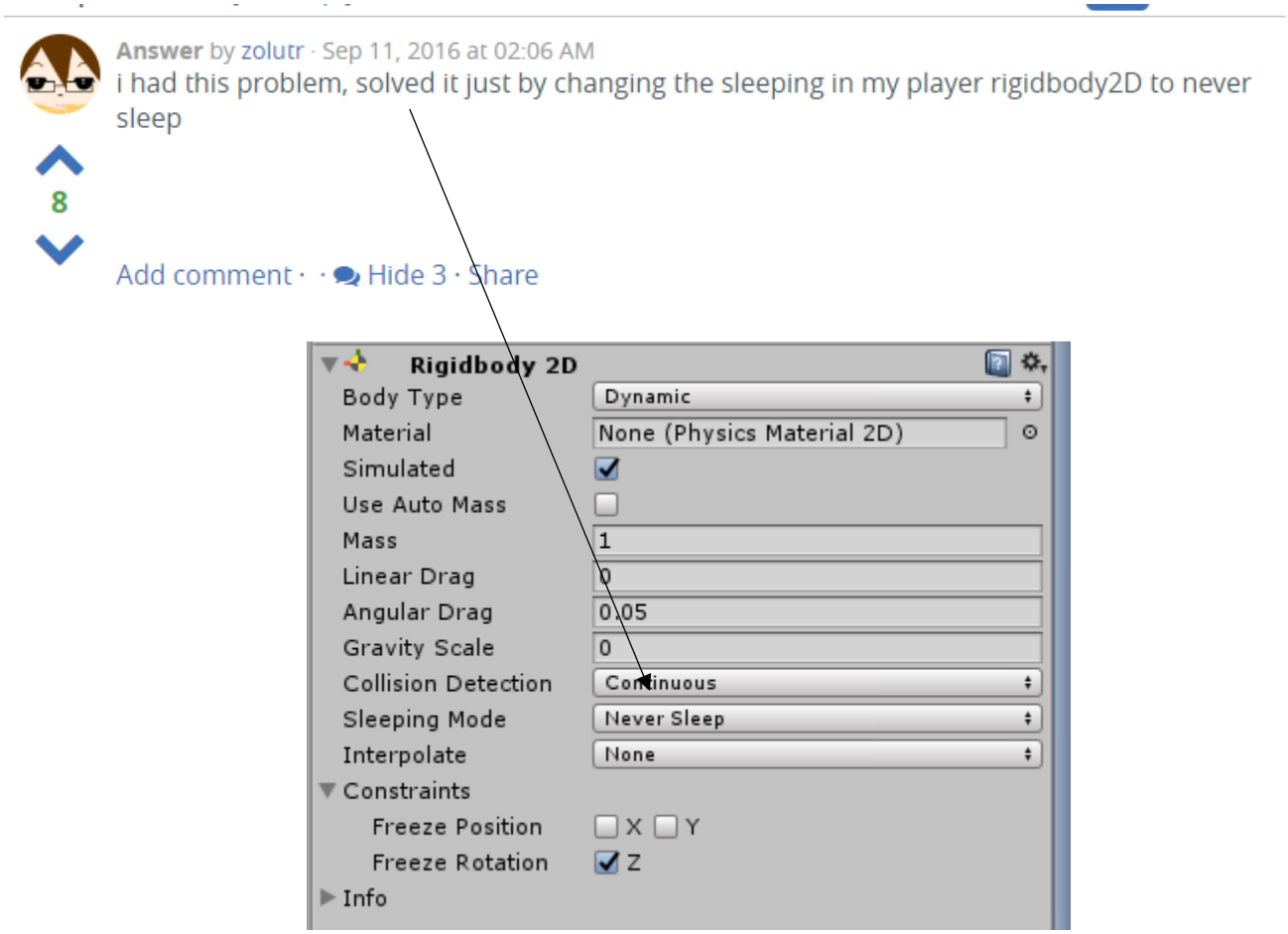
Therefore, I decided to combine the if statements for input “e” and “r” the Update function to the “OnTriggerStay2D”. This way the chest will only open if the “e” button is pressed inside the 2D collider area and the “r” button will close the chest if its inside the 2D collider area.

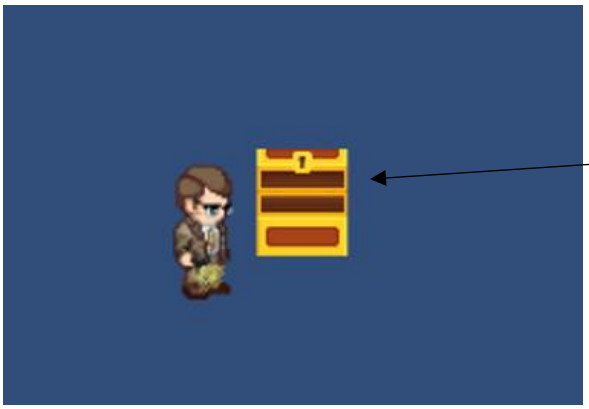


However, as shown, when I tested the code to see if it worked, it didn’t work when the Player was inside the 2D area and pressing “e” or “r”.

After researching about why the "OnTriggerStay2D" functions wasn't being called, I found out it was because of one of the setting within the "Rigidbody 2D" component of the "Chest" GameObject.

The problem was that the "RigidBody 2D" needs to be set to Never Sleep so that function is always ready to be called.[5]



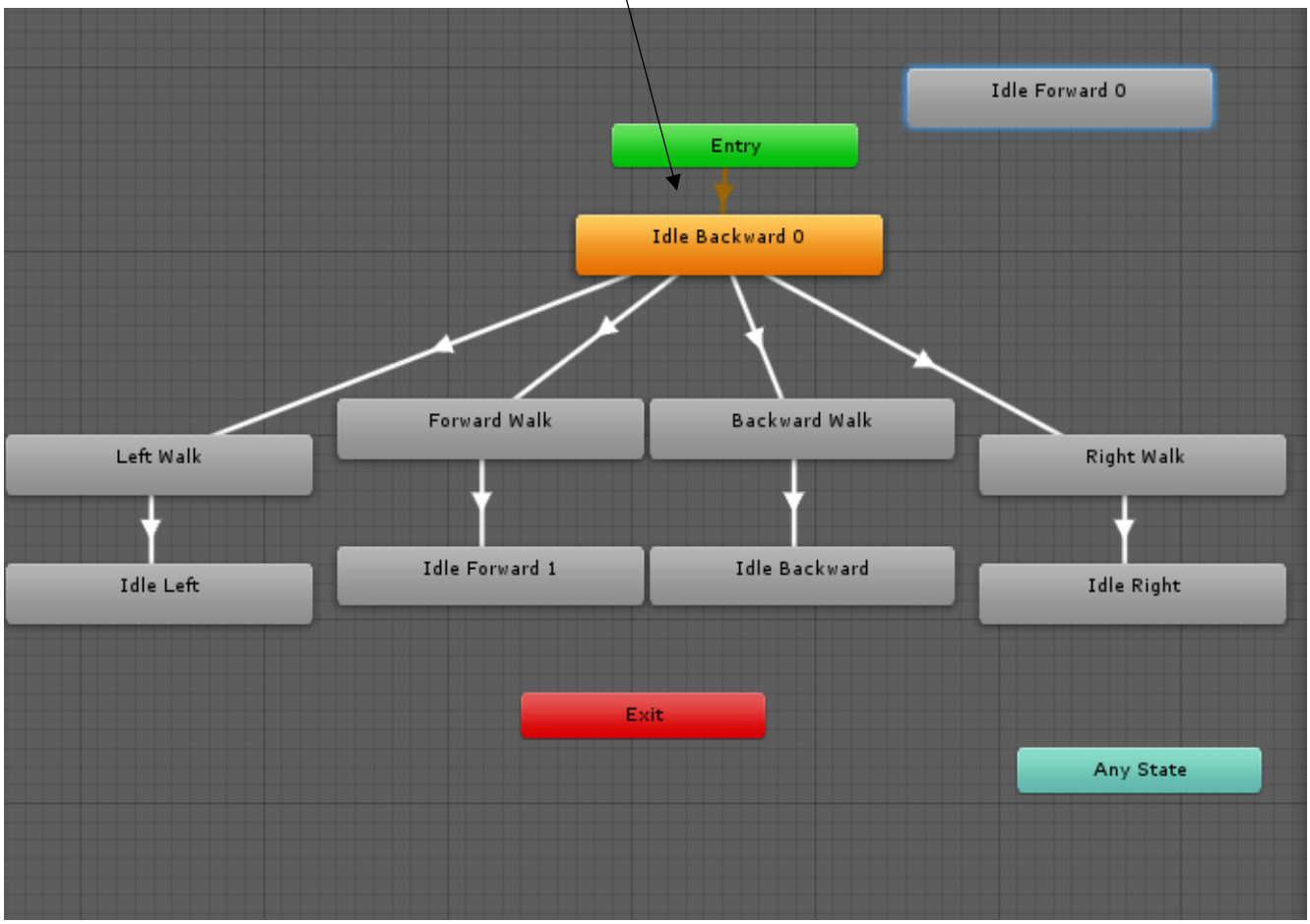


Once set to "Never Sleep" the code worked and the chest opened and closed when "e" or "r" was pressed.

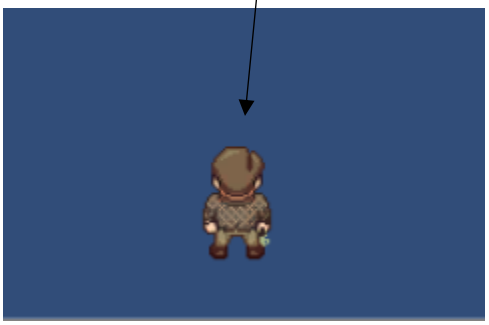
[5] https://answers.unity.com/questions/1001159/ontriggerstay2d-only-detecting-trigger-collisions.html - Date accessed 05/04/18 - RigidBody 2D never sleep.

QUICK PLAYER CHANGE

After using the Player GameObject for a while I realised that it would be better if the Player started facing backwards not forwards. To do this only required a quick change to the Animator component by swapping out the Idle Forward 0 to the Idle Backward 0 for the default layer.

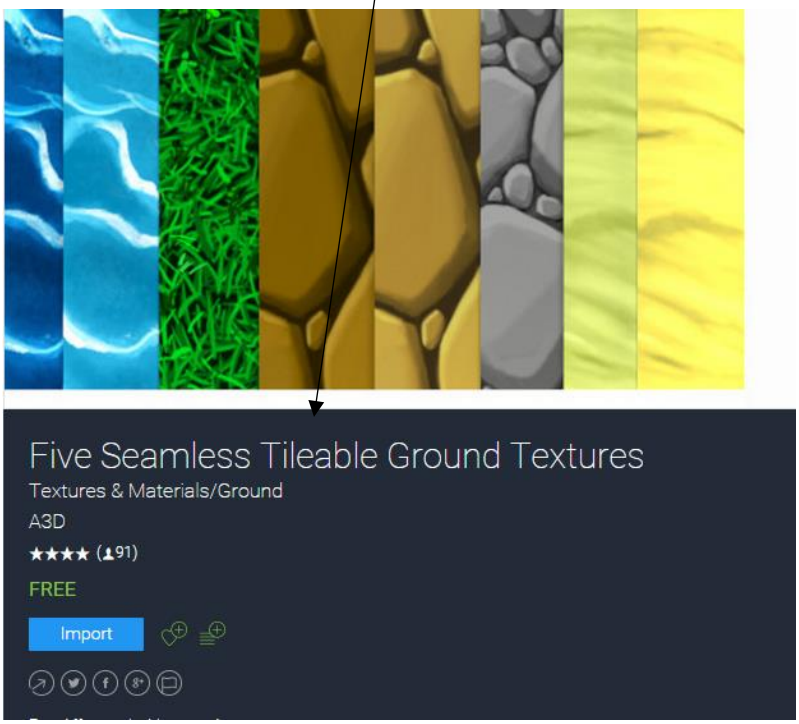


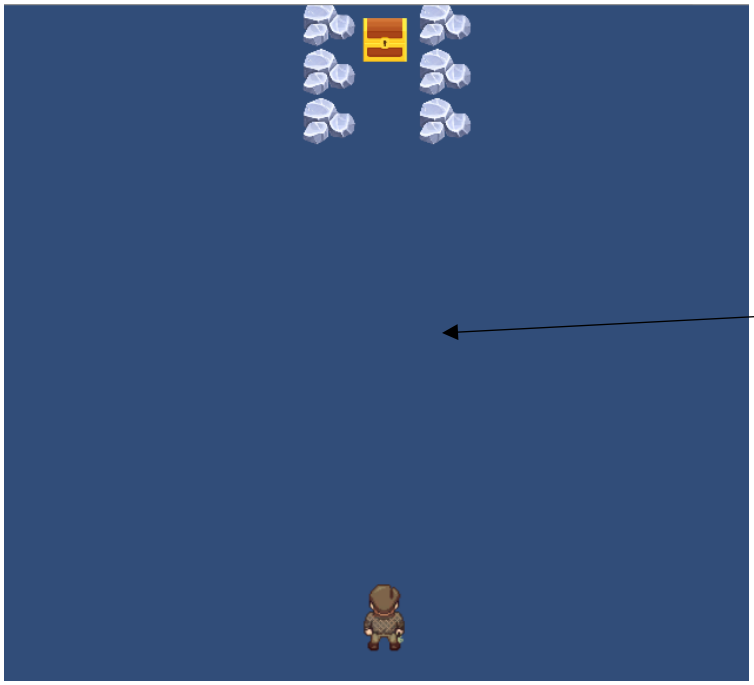
Player GameObject now starts out facing backwards



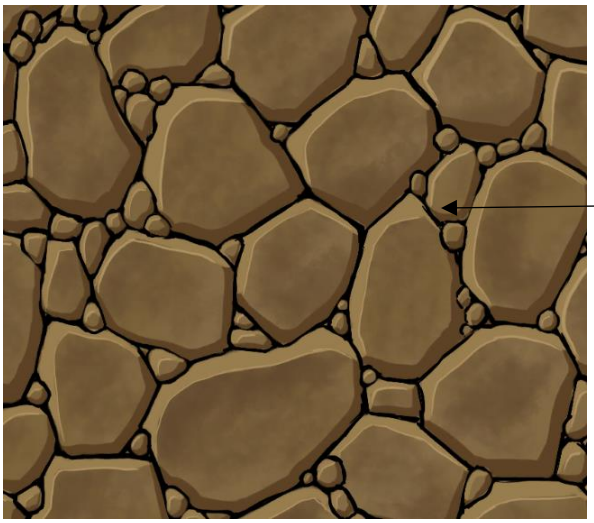
INITIAL LEVEL DESIGN

I've imported another asset from the Unity Asset Store to use for some textures





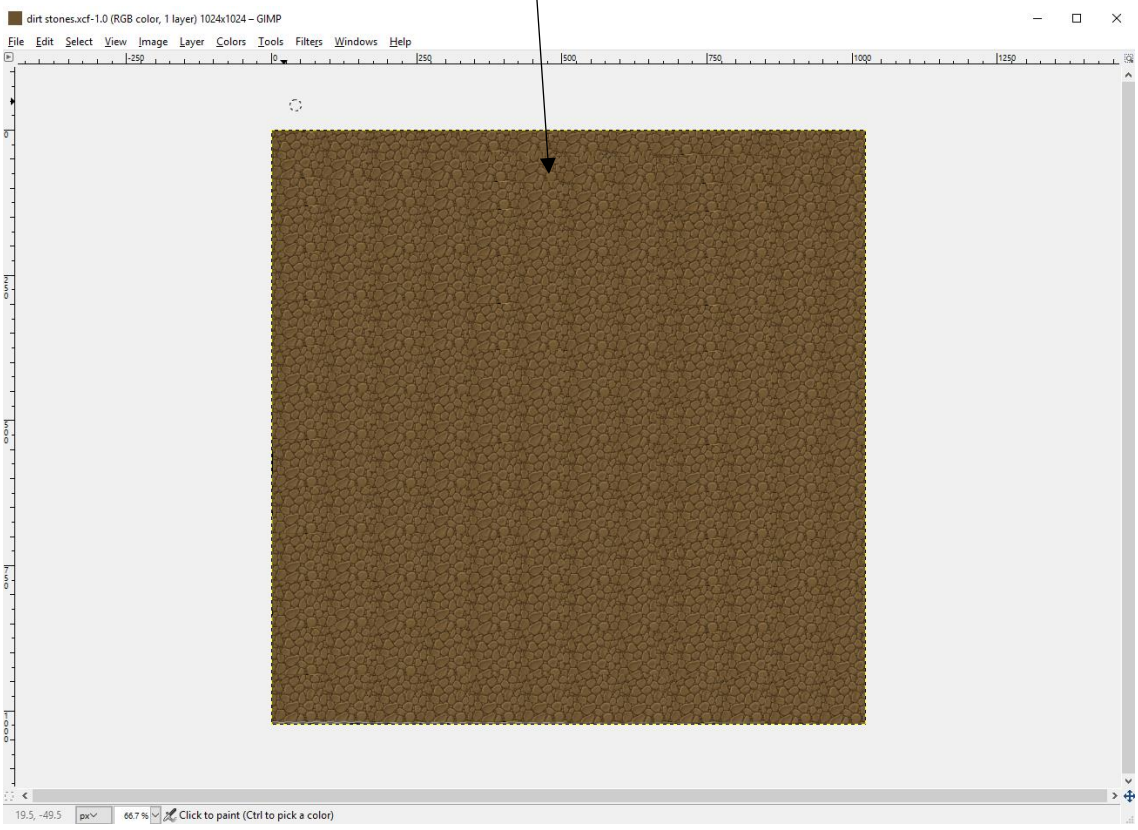
A first quick design of the level 1, this is just some imported rock GameObjects and also the Chest GameObject I configured before.



I then wanted to create the floor design from this one texture of dirt rocks

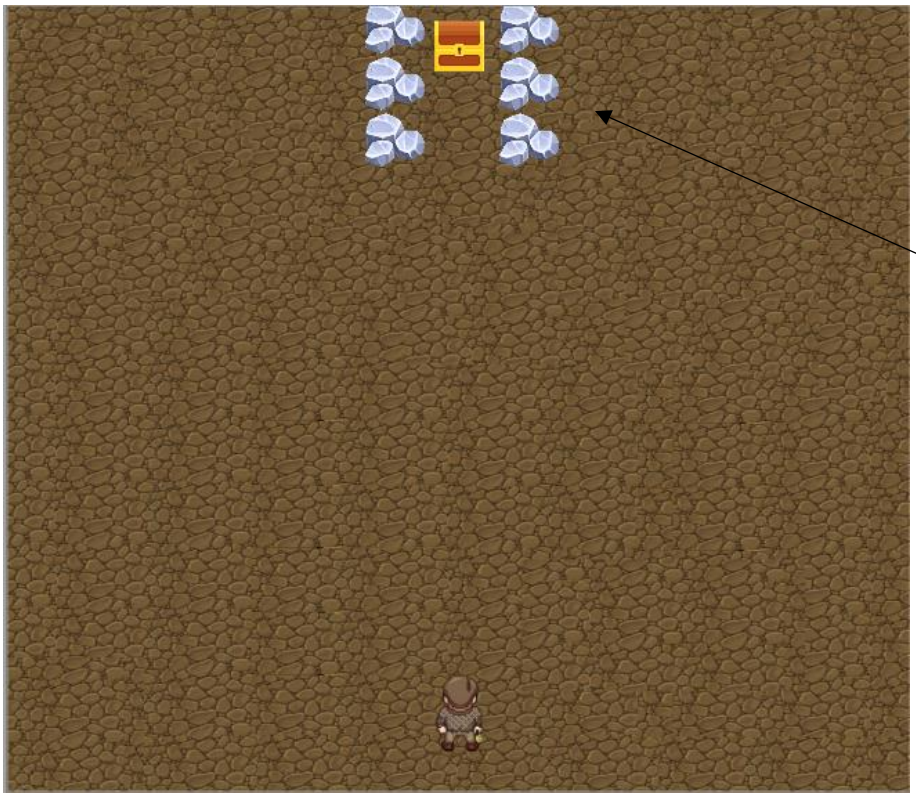
As I couldn't just use this one texture for the background as it would just be one big image of this, I needed to put it into an editor to make a large resolution image of multiple dirt rocks images.

I did this by using the Image Editor Gimp.



This is what it looks like when the final edited image is added to a Canvas GameObject.

However, the rocks no longer showed up over the floor image.

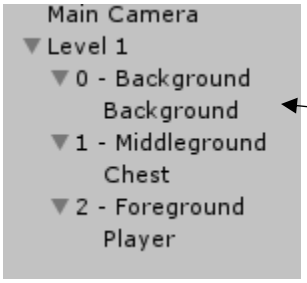


To fix this I needed to use the "sorting layer" option within the inspector components within the Unity Engine.

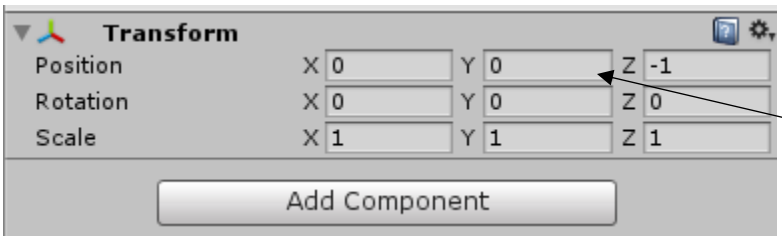
Making the rocks a higher sorting layer than the floor made the rocks appear above the floor.

[6]

[6] https://www.youtube.com/watch?v=ZzcyREamMUo – Date accessed 05/04/18 – Sorting layers.



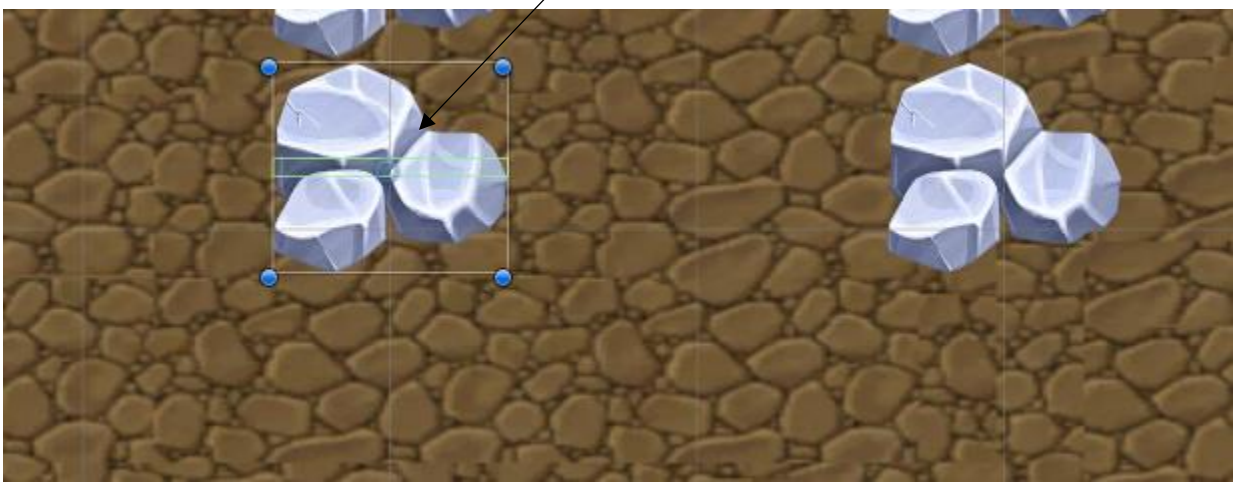
After this I decided to clean up the GameObjects to make it more organised by combining each up depending on the sorting layer.



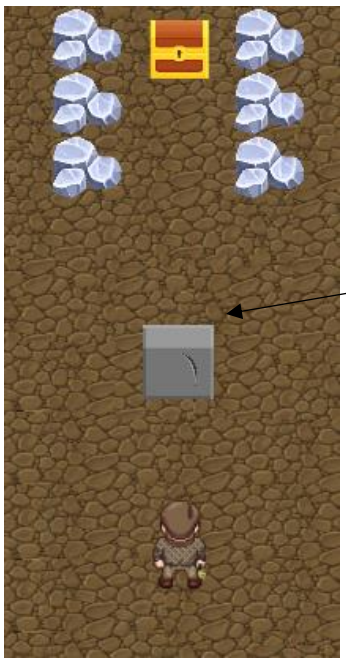
Also to edit where each GameObject was located, inside the Inspector for each GameObject there is a Transform component that allows the position to be change in all 3 axis.

Next, was to add 2D box colliders to each of the rock GameObjects so that the Player couldn't move through them.

The green rectangle is the 2D box collider.



As shown the Player GameObject cannot move through the rock anymore as the Player GameObject also has a collider on it too and so when two 2D colliders meet they cannot move through each other.



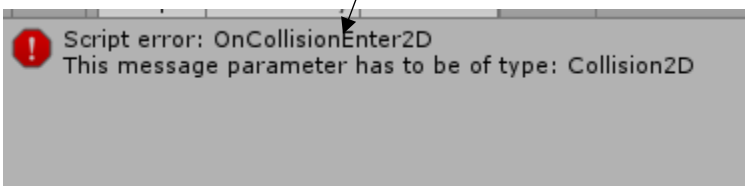
Next, I added another GameObject Called "Pushing Block" which I will use to allow the Player to push around to solve a puzzle.

The idea I had was to use the "Pushing Block" GameObject to be pushed and then for it to slide in the direction of the push until it hit another 2D box collider. This action will be part of the puzzle where the Player must push the block around a sort of "maze" to eventually be able to push it into a hole to get over to the chest.

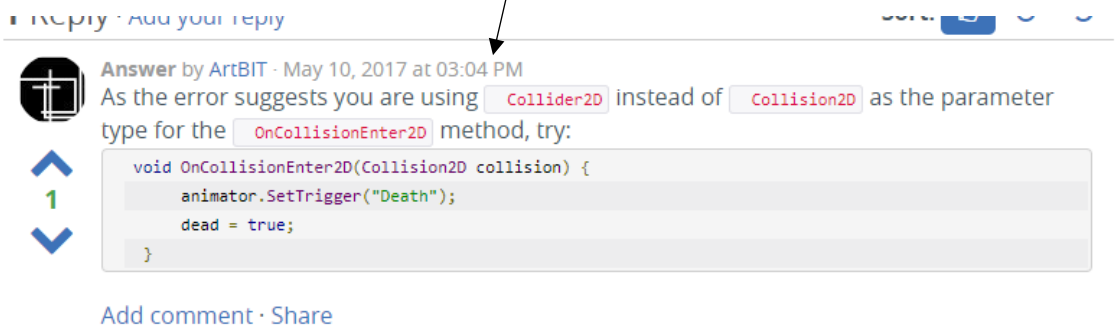
To start I added a script to the "Pushing Block" that will output Block touched" when the player collides with the pushing block. This was to be the start of the pushing code.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class PushBlock : MonoBehaviour {
6
7       void Start (){
8       }
9
10      void OnCollisionEnter2D (Collider2D col)
11      {
12          if (col.gameObject.name.Equals("Player"))
13                  Debug.Log("Block touched");
14
15
16      }
17  }
18
```

However, when I ran the code an error occurred



After googling the error code I found a forum explaining the problem and the fix.[7]



Reply · Add your reply

Answer by ArtBIT · May 10, 2017 at 03:04 PM

As the error suggests you are using `Collider2D` instead of `Collision2D` as the parameter type for the `OnCollisionEnter2D` method, try:

```
void OnCollisionEnter2D(Collision2D collision) {
    animator.SetTrigger("Death");
    dead = true;
}
```

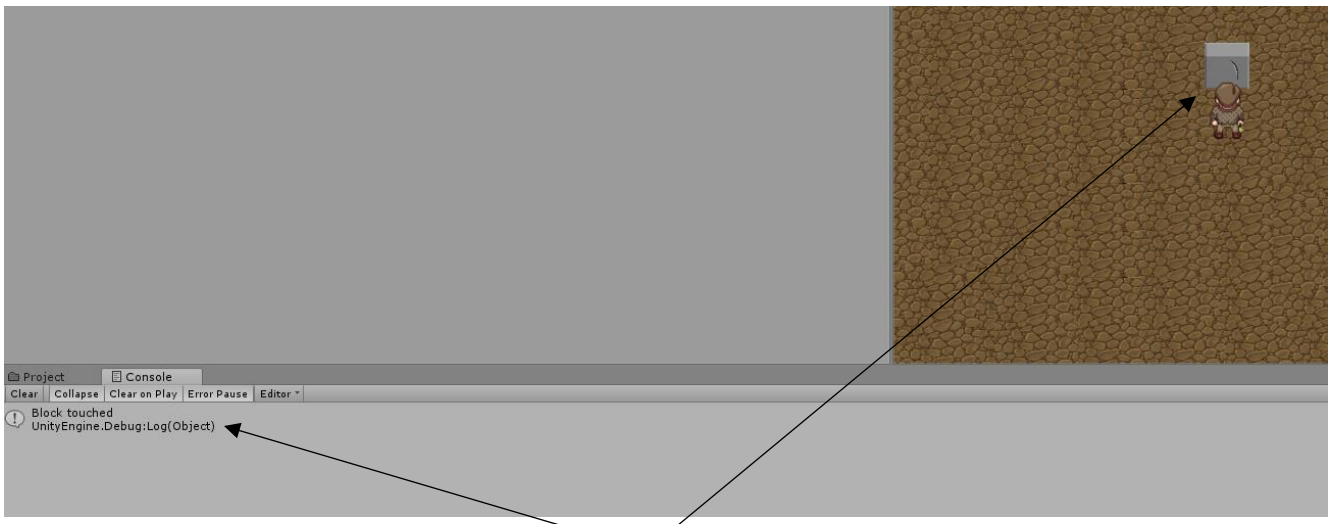
1

Add comment · Share

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PushBlock : MonoBehaviour {
6
7      void Start (){
8      }
9
10     void OnCollisionEnter2D (Collision2D col)
11     {
12         if (col.gameObject.name.Equals("Player"))
13             Debug.Log("Block touched");
14
15
16     }
17 }
18
```

The fix was to change the line 10 code "(collider2D col)" to "(collision2D col)".

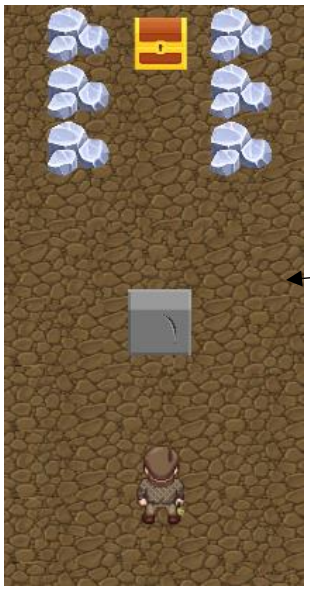
[7] https://answers.unity.com/questions/1150830/script-error-oncollisionenter2d-this-message-param.html - Date Accessed 05/04/18 - fixing the collider problem



Once changed, the code was working fine. As shown when the player collides with the pushing block, it outputs “Block touched”.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PushBlock : MonoBehaviour {
6      public float speed = 500f;
7      public int n = 0;
8      void Start (){
9      }
10
11     void OnCollisionEnter2D (Collision2D col)
12     {
13         if (col.gameObject.name.Equals("Player"))
14             //while (n < 1)
15             {
16                 transform.Translate (0,speed*Time.deltaTime,0);
17                 //if (col.gameObject.name.Equals("Chest"))
18                     //n = n+1;
19             }
20     }
21 }
```

Now I wanted to move the block in a sliding way when the player collided with the block. To try and do this I added in line 16 which moves the GameObject in the Y axis by the float “speed” and at “Time.deltaTime” which smooths out the translation.



The code does work and when the Player touches the pushing block it moves in the Y axis in a positive direction.

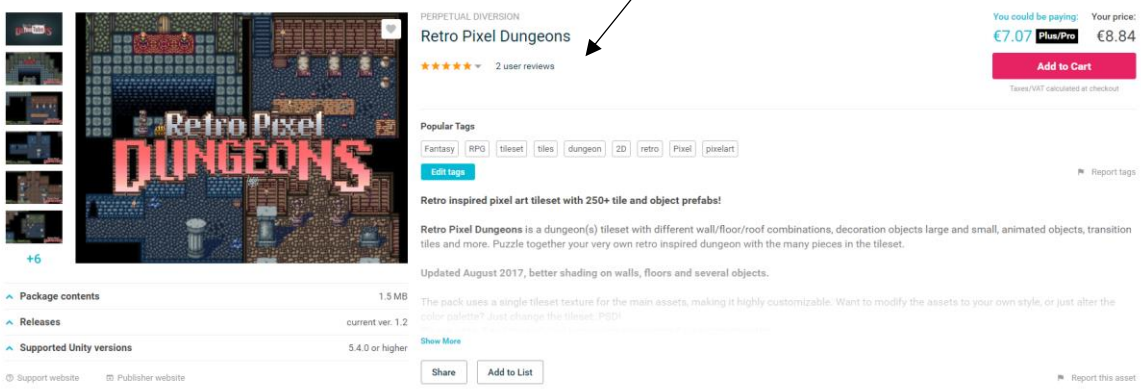
CHANGING THE LEVEL DESIGN

Before I made an more developments to my level I was thinking about my timescale and also how long it took me to just make a floor. Even though it was simple it did take a considerable amount of time which I realised was wasteful and so I took to looking at the Unity Asset store again for any assets which may speed up my visual design development.

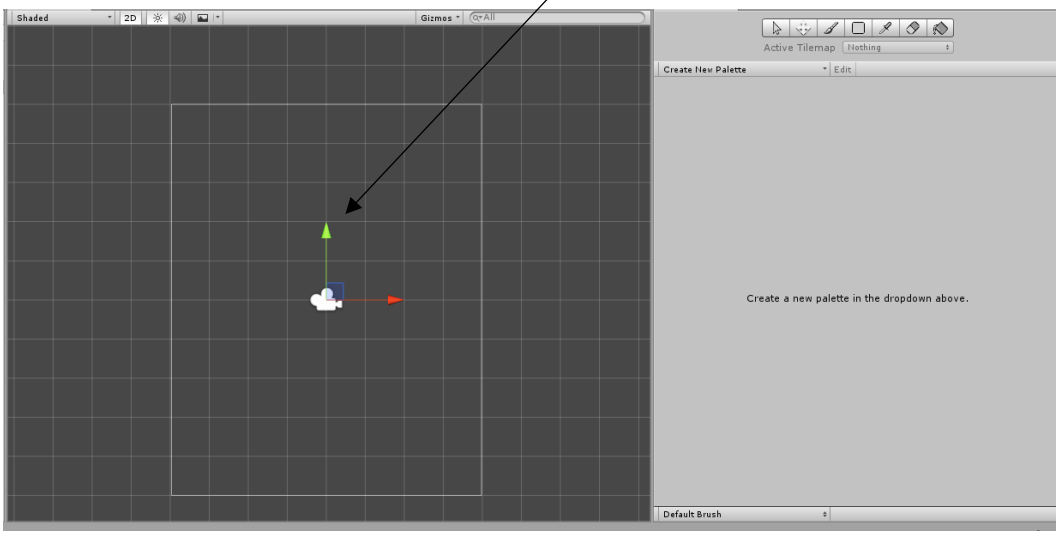
Whilst searching I came across a very well made asset which was a Tilemap asset. At first I didn't know what a Tilemap was and so I did a bit of research. I found out that a Tilemap is a texture/image that is based up of one grid block which all together can form a large grid based image, much like many block 2D platform games on the market.

Building my levels with set Tilemaps as a grid is a much more time efficient way to make the visual design of a scene in the Unity Engine. This Tilemap asset I very much liked as it was very similar to the genre and style I wanted Rune Raid to look like, as Retro and cartoon styles are very similar. Therefore, I very much wanted this asset as it look great and would significantly reduce my development time as I would no longer have to make any visual aspects myself.

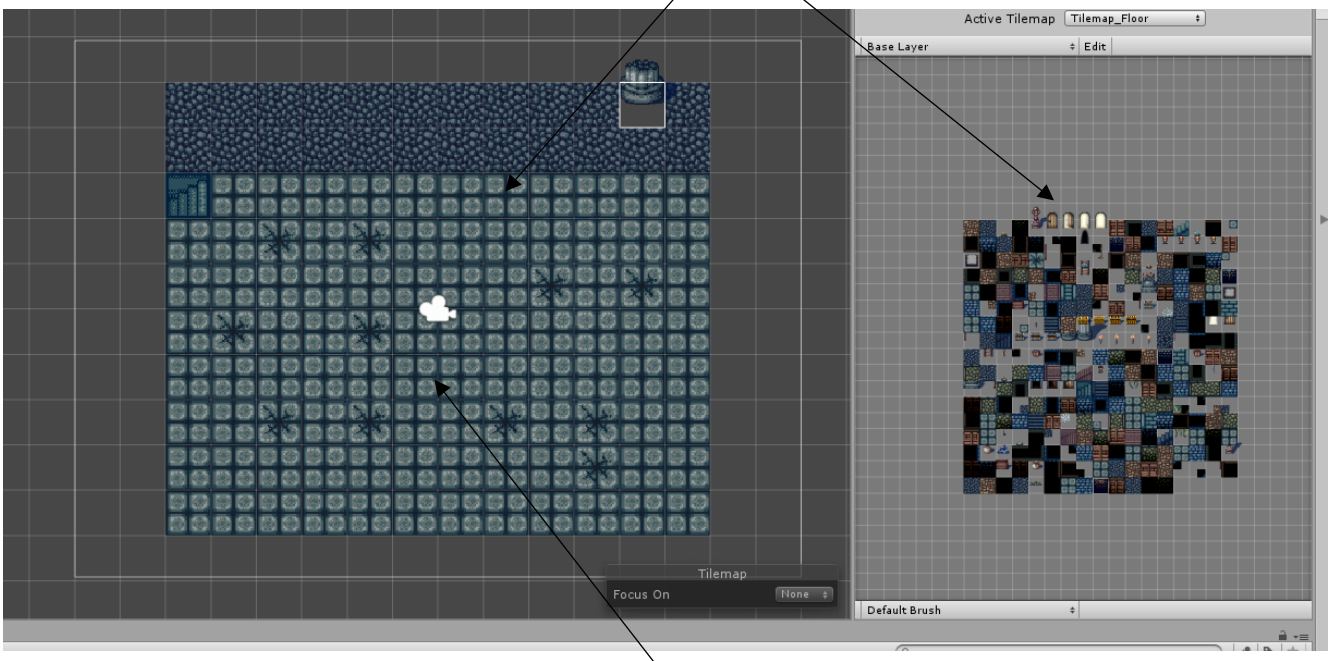
The only problem was that it cost €8.84 and from the start I planned to spend no money on this project. However, after some hard thinking I came to the conclusion that it was worth my playing for this asset as it would easily save over 10 hours of my time which is around 2 days development time and so for €0.80 an hour it was totally worth the money, I also very much liked it.



To use this new Tilemap aspect of Unity, first a Tilemap Grid GameObject needs to be created. At first it looks like this on first creation.

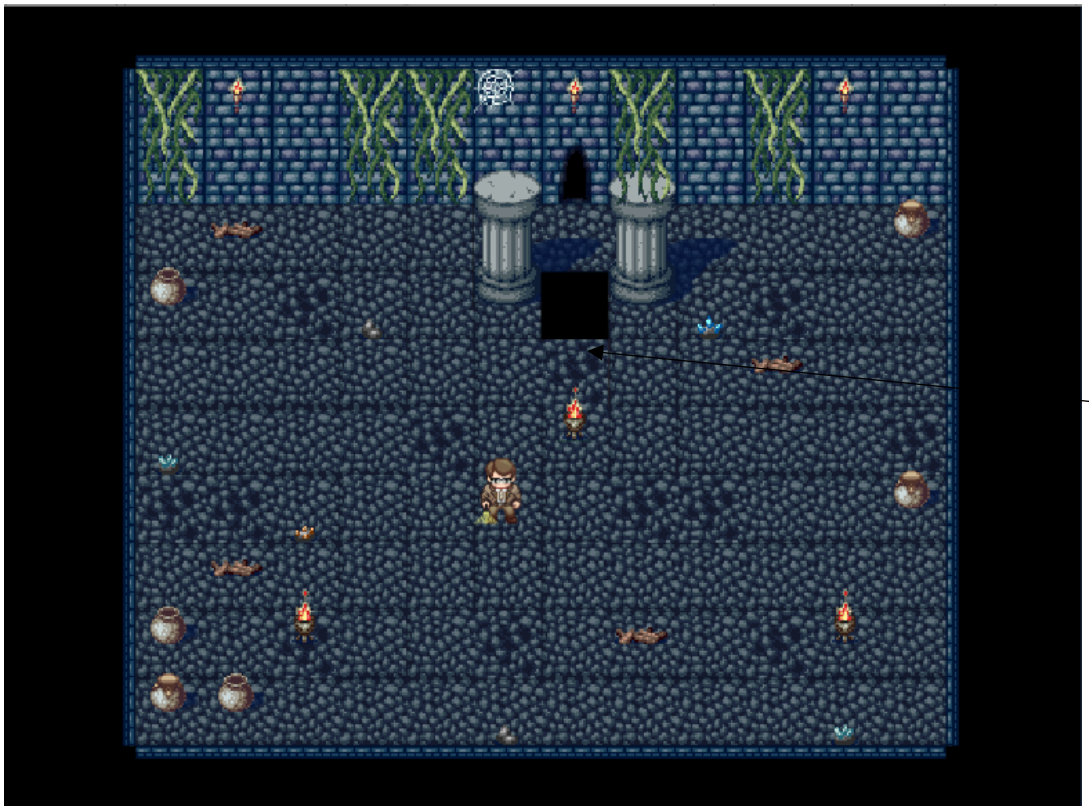


Next, I created a palette called “Tilemap_Floor”. A pallet is very similar to real life painting, it is just a place where all Tilemap images are stored and then individually selected and pasted onto the grid in the scene view.

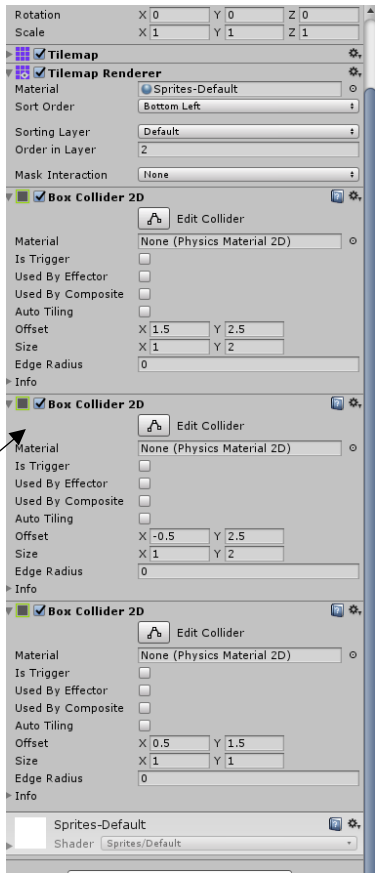


Here I was just messing around with some designs to see what I liked and to get an understanding of how everything worked.

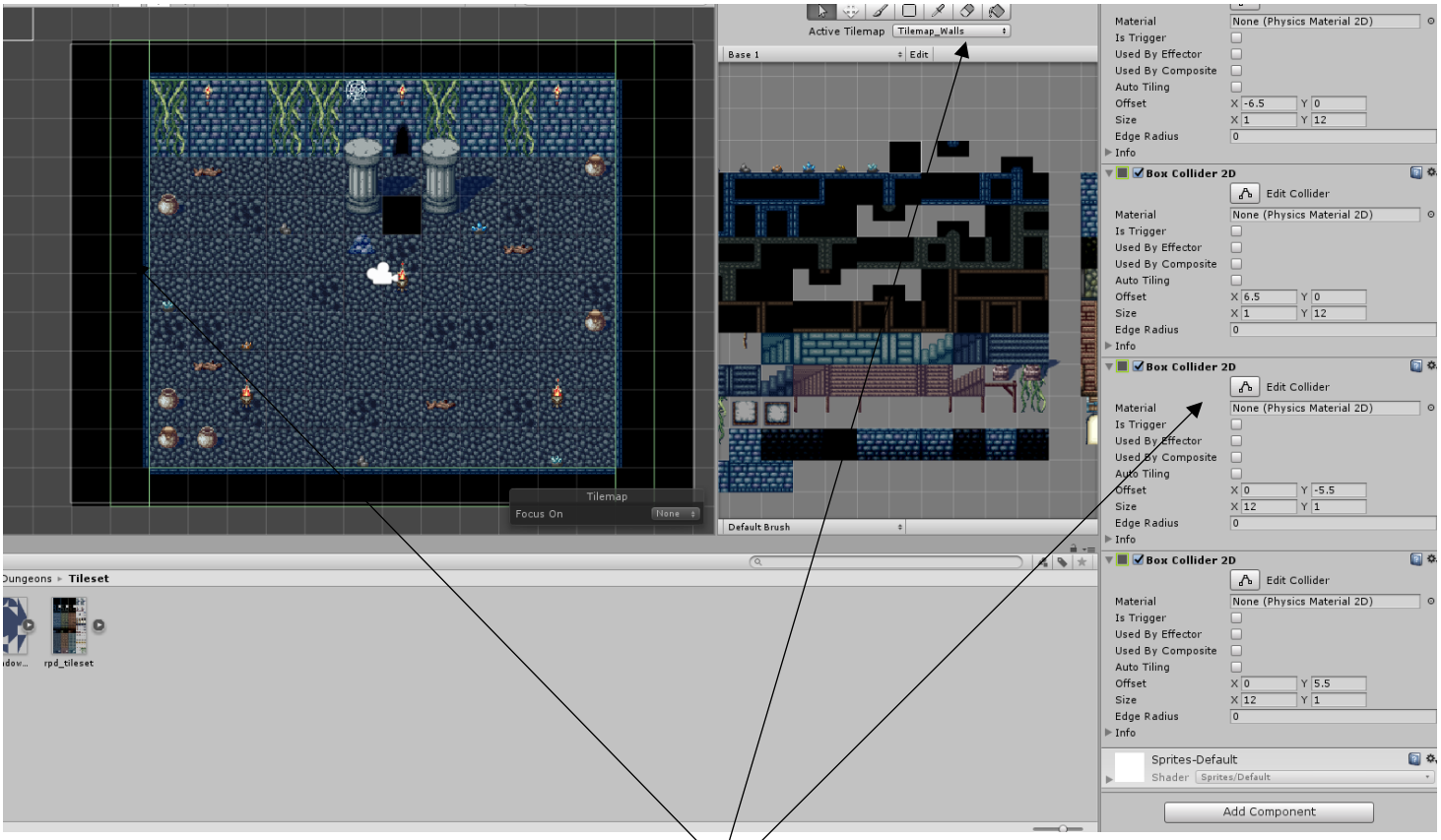
MAKING THE NEW LEVEL DESIGN



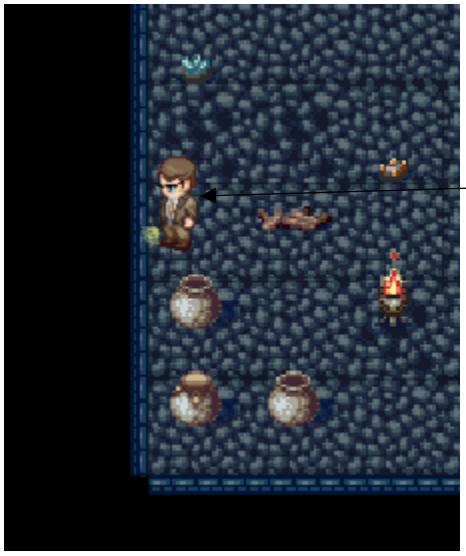
First foundation design



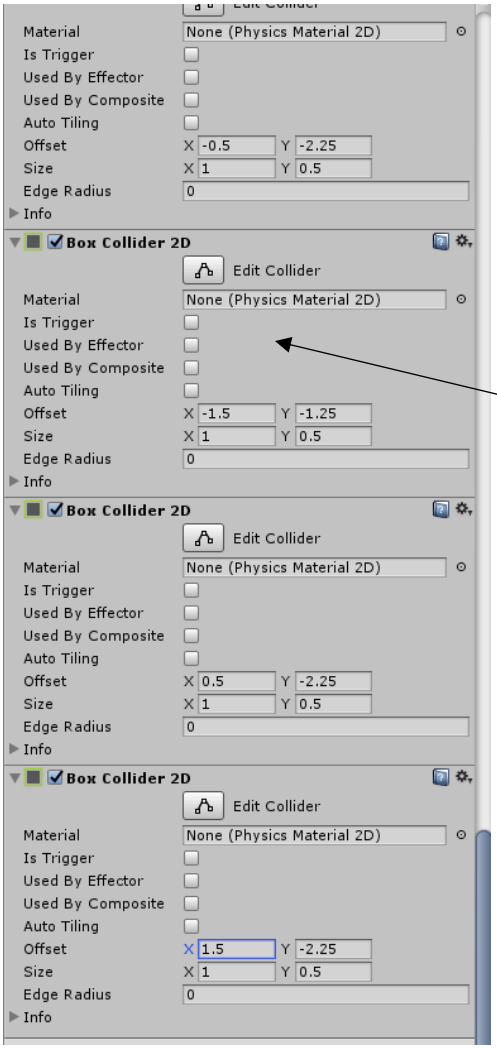
Adding in the pillar and hole colliders.



Creating a new Tilemap and calling it “Tilemap_Walls”. This is all the 2D box colliders for the wall so that the player can’t go outside the level

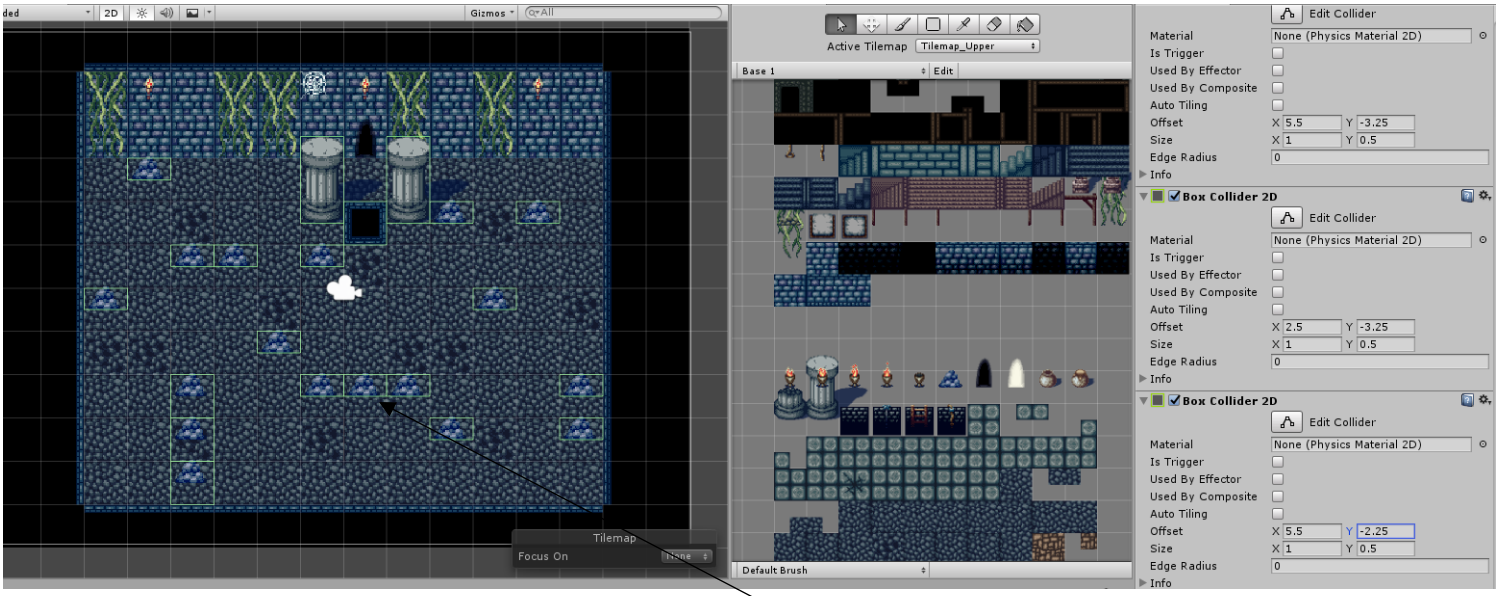


Showing that with the wall 2D colliders, the player can no longer move outside the level



Adding in a new Tilemap called "Tilemap_Upper" which contains all the rocks that will act as obstacles for the puzzle that the player must navigate the pushing block trough.

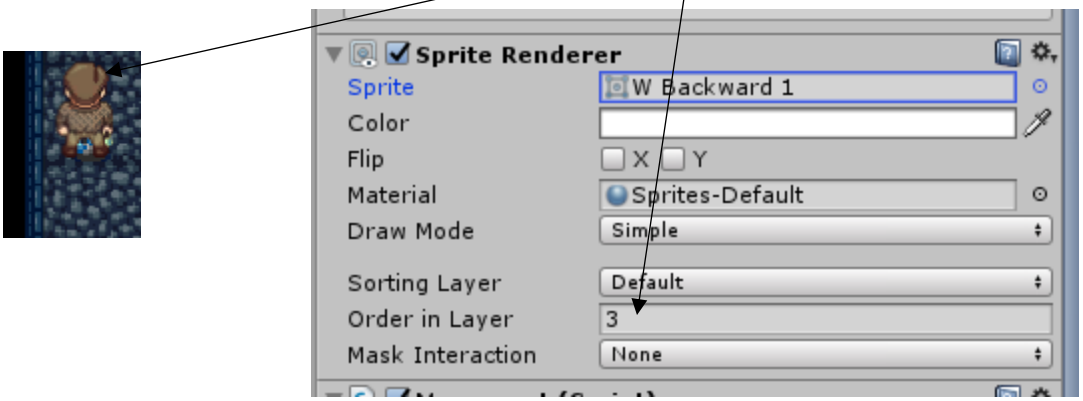
Also creating 2D box colliders for each rock to prevent the any other 2D collider from moving through them, such as the pushing block and the player GameObjects.



The final design of the rocks and their 2D colliders

I added another Tilemap GameObject called “Tilemap_Items” that is for all the other extra Tilemaps that will go over the floor and walls, such as pots, ladder, small ore etc. to make the level look more appealing.

The way I control the layers is by setting each Tilemap to a specific Order layer. Because the Player must be above every layer I've set it to Order layer 3 which is the highest layer at the moment.



I've now added in 2D colliders for each pot in the level as well because I don't want the player or the pushing block to move through them, as that would look like poor development if the player or pushing block could just move through them.

PUZZLE

Now I wanted to make it so that when the pushing block collided with the hole it removed the 2D box collider and turned off the hole to reveal the Tilemap on the layer beneath.

To start I wanted to find out the code for removing a 2D box collider.

The Tilemap underneath the hole

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class colliderBool : MonoBehaviour{
6
7      void Update()
8      {
9          {
10             if(Input.GetKeyDown("e")&&GetComponent<BoxCollider2D>())
11             {
12                 Destroy(GetComponent<BoxCollider2D>());
13             }
14         }
15     }
16 }
```

To destroy a 2D box collider the line 12 is needed. This gets the 2D box collider and then destroys it.

However, I also needed some code to destroy the GameObject hole as well and so to do this I needed the line 13. This line just completely destroys the GameObject reference. In this case the GameObject that the script is on which is why it is just "Destroy(gameobject)".

[8]

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class colliderBool : MonoBehaviour{
6
7      void Update()
8      {
9          {
10             if(Input.GetKeyDown("e")&&GetComponent<BoxCollider2D>())
11             {
12                 Destroy(GetComponent<BoxCollider2D>());
13                 Destroy(gameObject);
14             }
15         }
16     }
17 }
```

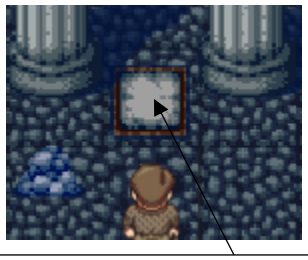
[8] https://docs.unity3d.com/2017.2/Documentation/ScriptReference/Object.Destroy.html - Date accessed 06/04/18 - Destroying GameObjects

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class colliderBool : MonoBehaviour{
6
7      void Update()
8      {
9          {
10             if(Input.GetKeyDown("e")&&GetComponent<BoxCollider2D>())
11             {
12                 Destroy(gameObject);
13             }
14         }
15     }
16 }
```

However, the line 12 makes the previous obsolete because completely destroying the GameObject means that the 2D box collider will also be destroyed.

Therefore, I just removed the previous code on line 11 and left line 12.

As shown, when "e" is pressed the Tilemap hole is destroyed.



With the other Tilemap beneath it then reveals this and allows the player to move trough.

The separate Tilemap_Hole.

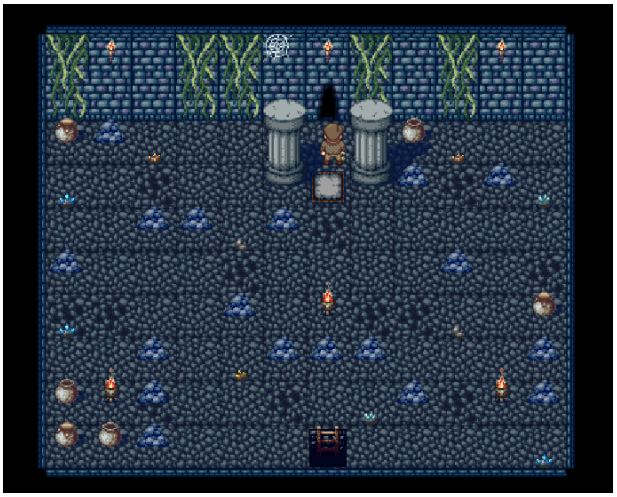
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class colliderBool : MonoBehaviour{
6
7      void OnCollisionEnter2D (Collision2D col)
8      {
9          if (col.gameObject.name.Equals("Pushing Block"))
10         {
11             Destroy(gameObject);
12         }
13     }
14
15 }
```

Now that I know how to destroy GameObjects, I've edited the code so that when the "Pushing Block" GameObjects collider meets the "Tilemap_Hole" collider it will destroy the "Tilemap_Hole" GameObject.

```
        }
        if (col.gameObject.name.Equals("Tilemap_Hole"))
        {
            Destroy(gameObject);
        }
    }
}
```

I've also added the same for the Pushing Block GameObject so that when it collides with the "Tilemap_Hole" GameObject it will destroy the GameObject.

This overall will give the look that the pushing block has been pushed into the hole and then fills it, allowing the player to move over it.



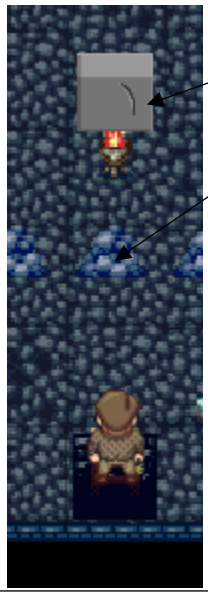
```
void OnCollisionStay2D (Collision2D col)
{
    if (Input.GetKey("a"))
        {
        transform.Translate(Vector3.left);
        }
        if (Input.GetKey("d"))
        {
        transform.Translate(Vector3.right);
        }if (Input.GetKey("w"))
        {
        transform.Translate(Vector3.up);
        }
        if (Input.GetKey("s"))
        {
        transform.Translate(Vector3.down);
        }
}
}
```

Next, I wanted to make the Pushing Block GameObject to "slide" in the direction that it was pushed in.

This is the code that I thought might work as the WASD keys decide which location the player is moving in and so also would be the direction the block has been pushed in.

Vector3.right just moves the GameObject in the right direction continuously until any collider it hit.

However, when I ran the code it didn't work the way I wanted it to. When the player pushed the Pushing Block for it to "slide", when it met a 2D box collider it just went straight through it. I didn't understand why this was the case because 2D box colliders shouldn't be able to pass tough other 2D box colliders. It goes straight though the rock collider.



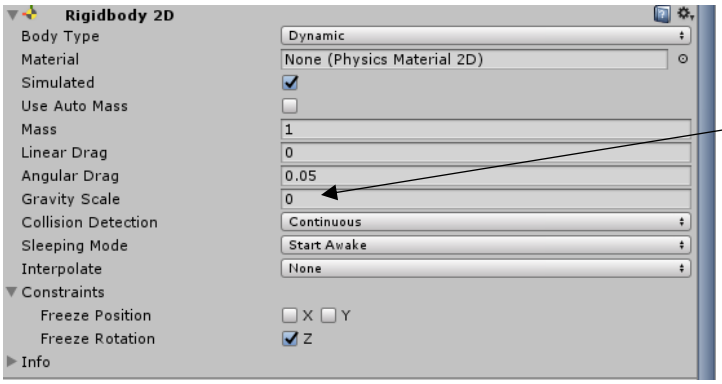
After some research, I found out that this was a big problem with 2D box colliders when a GameObject is moving too fast. Because it's moving too fast the GameObject just isn't detected and so passes straight through the 2D box collider.

There is a fix for 3D box colliders but as I'm only using 2D box colliders there is now way to fix this problem that I could find anywhere on the forums. [9]

Because of this error, I had to come up with another way I could try and represent the pushing blocks sliding movement. After some thinking and research I came out with the idea to use "Area Effectors" and the "RigidBody 2D" components to produce "sliding".

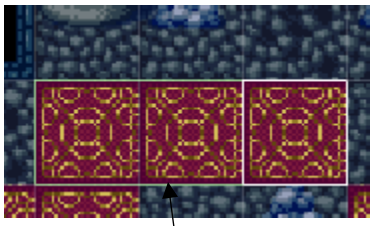
The way to do this was to set the "Rigidbody 2D" to have a "Gravity Scale" of 0 which means it won't fall down the Y axis when the game is played. As the GameObject will no longer fall, it can be moved by another Rgidbody 2D, which is the Player GameObject. Now the Player can push the Pushing Block but it doesn't slide yet. To do this I needed "Area Effectors" which apply a direction of movement for GameObjects that enter its area.

The idea is that the player will have to push the Pushing Block onto the many Area Effectors located in the level, which will then create a sliding effect, moving the pushing block in one direction depending on the specific Area Effector.



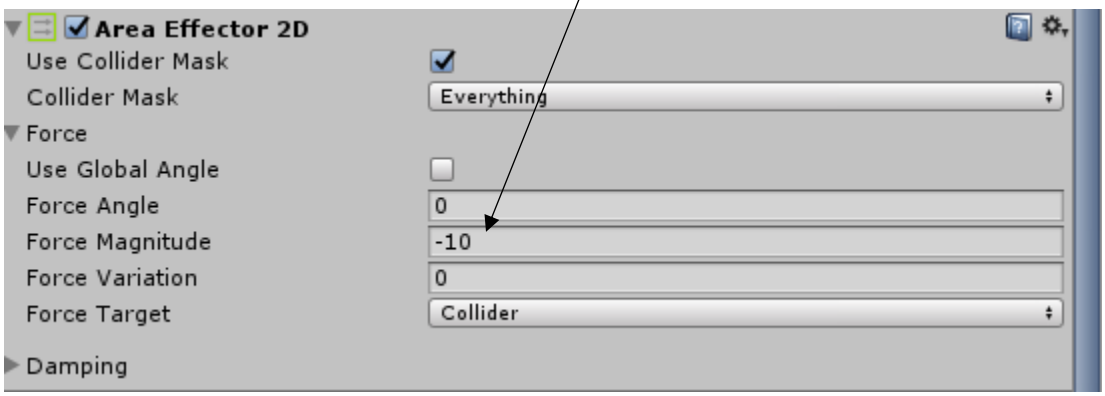
Setting Gravity Scale to 0

[9] https://forum.unity.com/threads/collider-2d-object-pass-through-when-object-move-too-fast.220480/ - Date accessed 06/04/18 – Showing that there is no known fix for the 2D box collider problem



These green boxes are the Area Effectors but for me to be able to see the, easily I've made them all carpet Tilemaps just so that I can clearly see them and build the puzzle.

Each Area Effector has some variables that you can change such as Force Magnitude, which is what I used. Depending on the direction I want them to move GameObjects in depends on the Force Magnitude. Here below, ive set the Area Effector to -10 which means it will move in a positive direction of the X axis by a force of 10 each frame, simply moving Right at a force of 10.



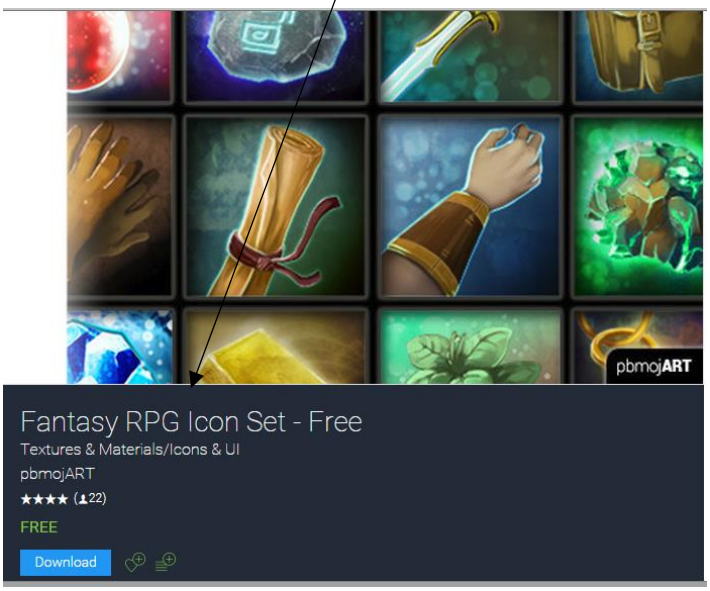
Once I set up all the Area Effectors and tested them, I found that once the Pushing Block was pushed onto the first Area Effector, it would just carry on and move onto other Area Effectors in the next direction and so complete some of the puzzle without the user's pushing input.

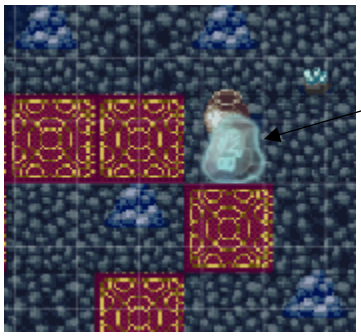
To fix this I reduced the size of the Area Effectors so that each section didn't overlap and meet any other Area Effector in another direction.

After editing the Area Effectors, the problem was sorted and now the puzzle part of the level works perfectly. The player can push the Pushing Block onto the Area Effectors which them "slide" the Pushing Block to the end of its Area, where the player must them walk and push the Pushing Block again, repeating this process until they push the Pushing Block into the Hole, where it then destroys and allows the player to cross the to the exit.

COLLECTING RUNES

As I wanted each level to contain "Runes" which the player can optionally collect, I wanted to add one into this level. To do this I imported an asset from the Unity Store which contained a nice Rune image that I used for the image of the Rune GameObject.





Look of the Rune in the level.

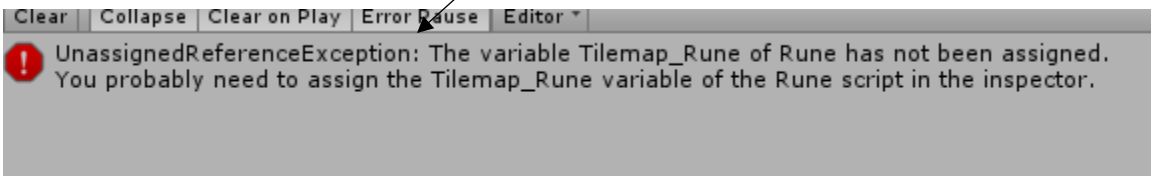
Next, I needed to make a script that when the Player GameObject collided with the Rune GameObject, it would turn off/destroy the "Tilemap_Rune" which contained the Rune GameObject. This would give the appearance that the Rune has been picked up by the player.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Rune : MonoBehaviour {
6      public GameObject Tilemap_Rune;
7
8
9
10     void OnCollisionEnter2D (Collision2D col)
11     {
12         if (col.gameObject.name.Equals("Player"))
13         {
14             Tilemap_Rune.SetActive(true);
15         }
16     }
17 }
```

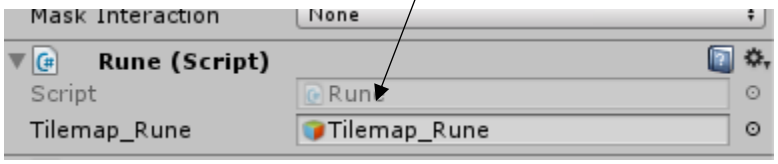
".SetActive(true)" activates the given GameObject I, in this case the Tilemap_Rune GameObject.

However, I wanted to make it so that when the level loaded, the Rune was hidden inside one of the pots and the Player must use the Pushing Block to break the pot and reveal the Rune inside it. This is just to make the level more interesting. Therefore, I also needed a script that when the Pushing Block collided with the pot, it would destroy the Pot GameObject and activate the Rune GameObject.

When I ran the code and tested to see that when the Player collides with the 2D box collider of the Rune GameObject, it just crashed and came out with an error



However, it turns out that it was just because I hadn't lined the GameObejct "Tilemap_Rune" to the script parameter

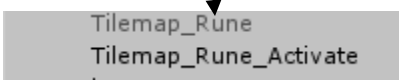


Next, I needed to create a script that when the Pushing Block collided with the pot, it would destroy the pot and activate the "Tilemap_Rune" GameObject. I've just changed the Rune script shown before to adjust so that it detects the Pushing Block instead.

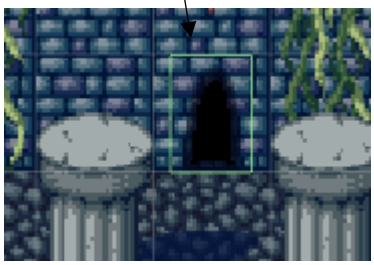
```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Rune : MonoBehaviour {
6      public GameObject Tilemap_Rune;
7
8      void OnCollisionEnter2D (Collision2D col)
9      {
10         if (col.gameObject.name.Equals("Pushing Block"))
11         {
12             Tilemap_Rune.SetActive(true);
13             Destroy(gameObject);
14         }
15
16     }
17 }
```

This script does jus this. Line 10 checks if the Pushing Block GameObject collides with the 2D box collider of the pot and if it does then it Activates thr "Tilemap_Rune" GameObject and then destroys its own GameObject which is the pot.

I found out that I can't put the Rune script onto the Tilemap_Rune GameObject because it will at the start of the level it will be turned off and so the script won't be running. Therefore, I made another GameObject that I've added the script to which contains the 2D box colliders to detect the Pushing Block GameObject collision.



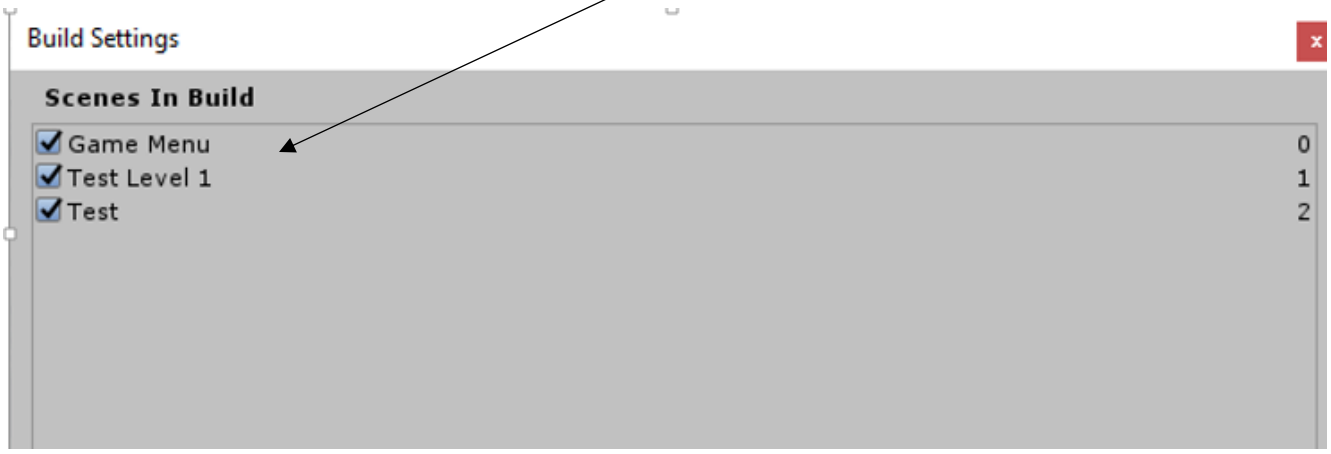
Now that the puzzle and Rune collection was working, I needed to create an exit GameObject. This will work by checking if the Player collides with its 2D box collider and if it does it will transition to the next scene.



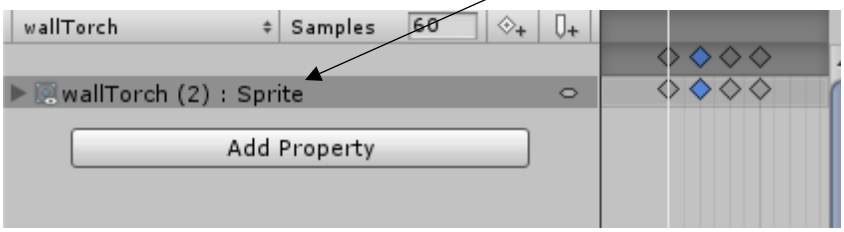
The code is very similar to the Buttons on the Main Menu scene as once the Player GameObject collides with the Exit collider it will run line 11 and load the next scene.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.SceneManagement;
5
6  public class level_1Exit : MonoBehaviour {
7
8      void OnCollisionEnter2D (Collision2D col){
9          if (col.gameObject.name.Equals("Player"))
10         {
11             SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
12         }
13     }
14 }
```

To check if it was working I added another scene into the build setting which indexes the Scenes. When line 11 runs, it should load the "Test" scene as its +1 from the "Test Level 1" scene. When I tested this it did indeed work and the next scene was loaded.

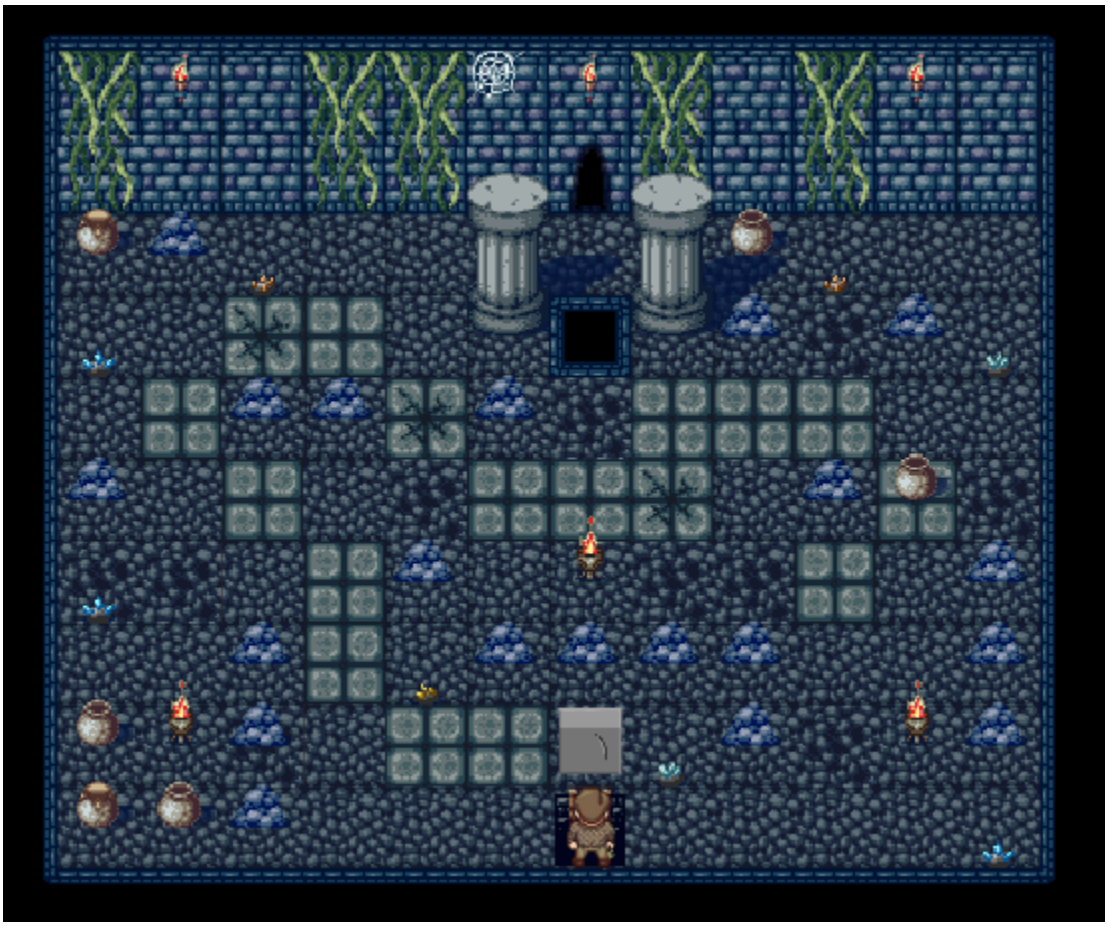


Finally, as a nice aesthetic for the level I wanted to animate the wall and floor torches that came with the asset that a bought. This was easily done as the asset pack already came with the multiple frames of a moving torch flame. Therefore, all I needed to do was make them into their own GameObject and not a Tilemap and then use the Animation component to build the frames.



Once the torches were fully animated, the majority of level 1 was complete. The only additions I needed to develop was the Hint system, a few helpful text boxes, the story and the Timer and its uses.

However, I wanted to finish all 3 levels before I started to develop these aspects because I felt it would make more sense because all these aspects will be used on the all the levels and so I would prefer to just develop them later when I can add them all to each level at the same time.



MEETING SUCCESS CRITERIA

I am confident that for Level 1 I've met all the success criteria relevant to this section of development that I have developed so far.

- Documentation of testing – I've clearly shown evidence of testing that I've done throughout the development of this level and shown any modifications I've made due to it.

- Documentation of errors whilst in development – Every error that I've come across I've documented and explained the reasoning and the solution to the problem.

- The player will be able to interact fully with every Rune Raid aspect – Every aspect I've set out to develop I've and ensured that the user can use it fully by testing them myself. However, I've not yet added a few aspects yet as I've discussed previously.

- All algorithms will be as efficient as I can possibly make them – for this level development so far there hasn't been many scripts to develop because the majority of the level is using the built In Unity Engine components which reduces the amount of extra scripts needed.

- All decisions made will be shown and justified – Every significant decision I've made to this development I've fully recorded and justified, whether they have been planned or not didn't matter.

- Run Raids puzzles must be challenging for the player – As this is the first level I didn't want to make this level that challenging because its meant to ease the player into the game but I do believe it requires some decent thought process to complete the level and I've also added a secret Rune to collect for any player who chooses to explore a bit.

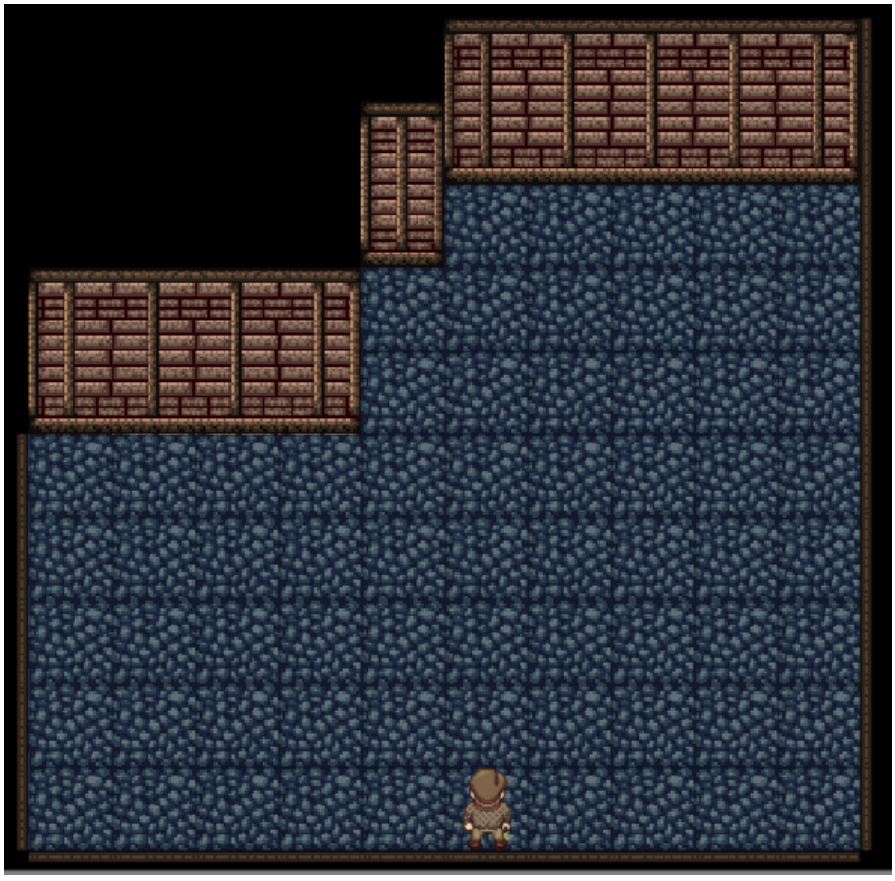
- All the code is split up into specific functions within modules – Every script has their own functions within them that are grouped effectively to by relevance and each different aspect has their own script.

Making Level 2

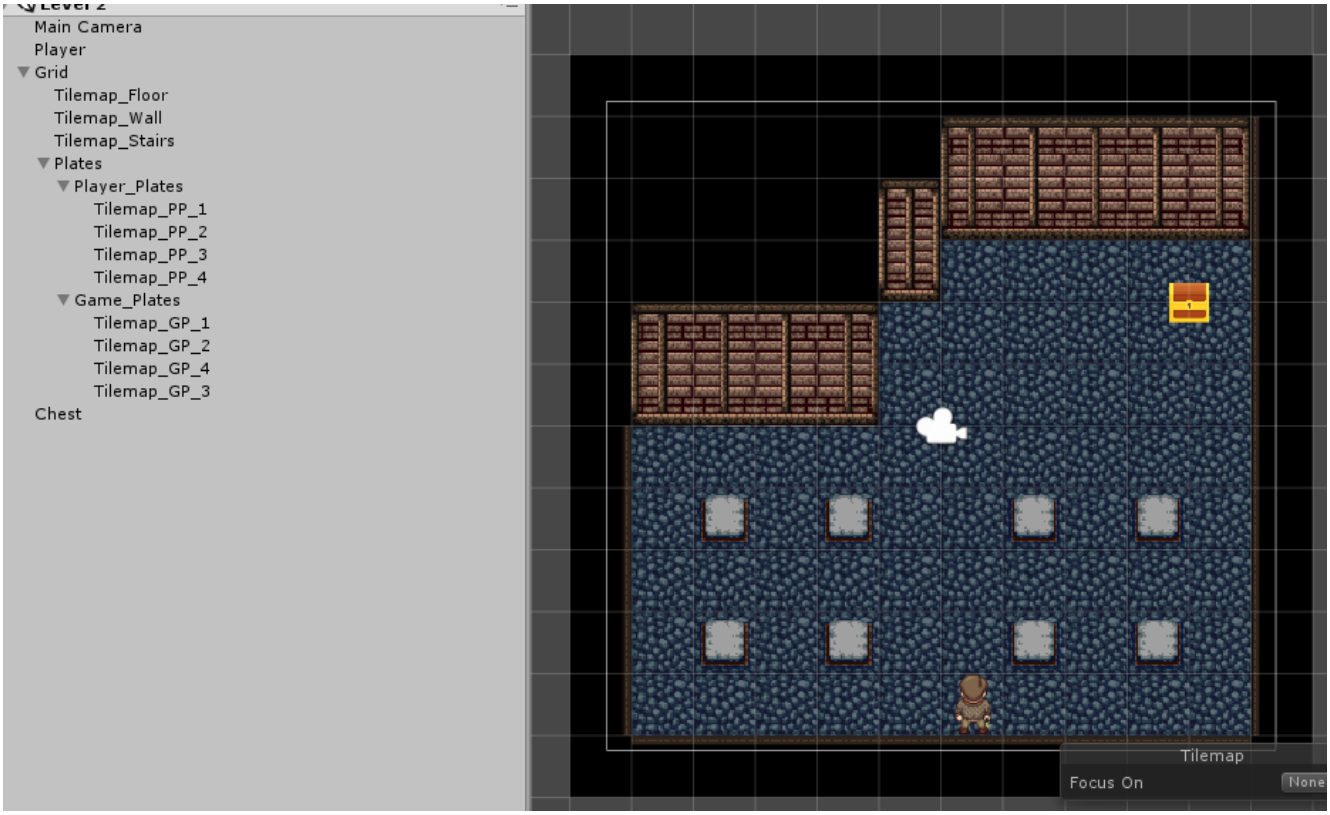
My idea for Level 2 was to make a copying puzzle, one half of the level will have 4 plates that randomly activate 4 times to create a pattern, then the player will have to copy the same pattern on the other half with pressure plates. This will be repeated 3 times and if the player gets the pattern right 3 times in a row it will activate the exit.

I also want to utilise the chest I made originally made for Level 1 and make it so that if the player manages to copy the pattern all 3 times without any mistakes then the chest will activate and give a rune to the player.

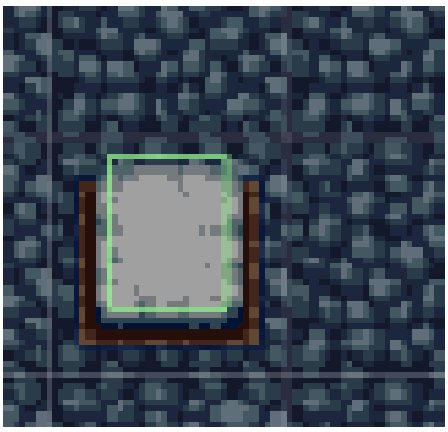
LEVEL DESIGN



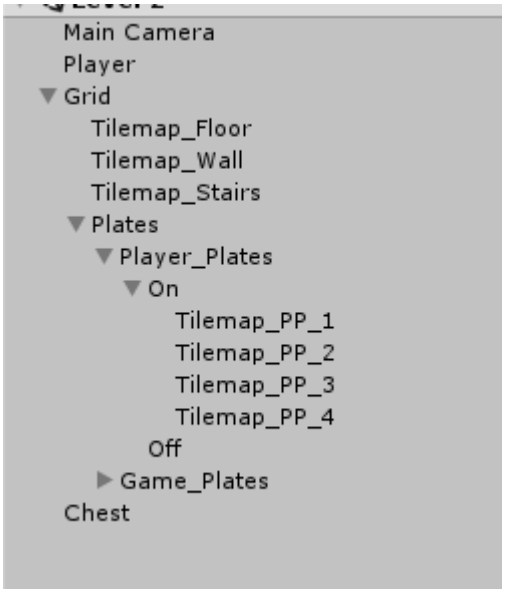
Foundations of the level with two Tilemaps, floor and walls. Also I've just copied the Player over from Level 1 and removed the irrelevant scripts that were only for Level 1



Adding in the Chest I've made before and added in two Tilemaps, the player pressure plate (PP, Left) and the puzzle plates (GP, Right).



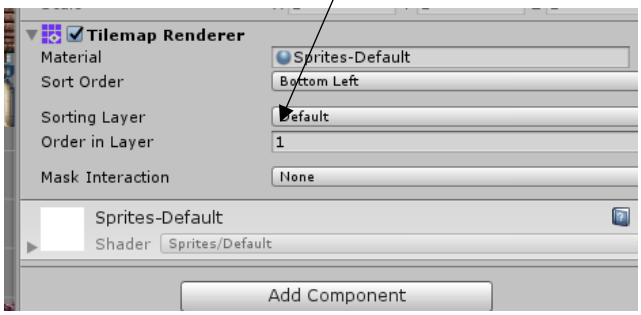
For each of the left 4 pressure plates I've added a 2D box collider to allow the Player to interact with them.

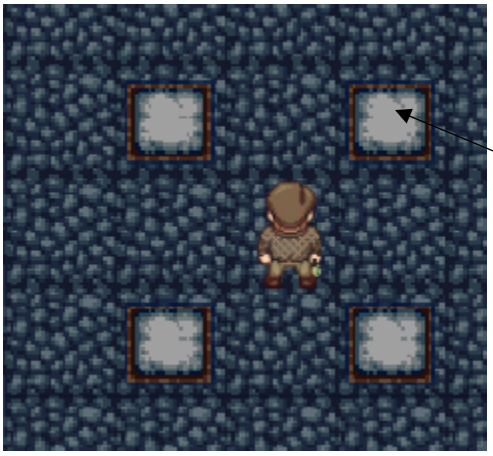


The GameObject structure so far, consisting of the Tilemaps, player and

Then, I created 4 new Tilemaps which were each a different Pressure plate but the pressed down version. This is so that when the Player walks on the off pressure plate, it will activate the pressed down version and the off version will deactivate. Then when the player walks off the plate, the off plate will be activated and the on plate will be deactivated. This is to give the look of a proper pressure plate that is being pressed down by the player and then pop back up when the player walks back off.

I also created another Tilemap that will be a layer below the other puzzle plates on the right (order layer 1), so that when they are "pressed down" for the puzzle (deactivated) then the pressed down version will be below and so show.





All 4 pressed down pressure plates on the right as another Tilemap.

PRESSURE PLATE PUZZLE CODE

Next, I needed to create a script for the Pressure Plates on the left so the player can interact with them by pressing them down when the Player walks on them.

I've already got a 2D box collider on each of the left pressure plates as shown previously.

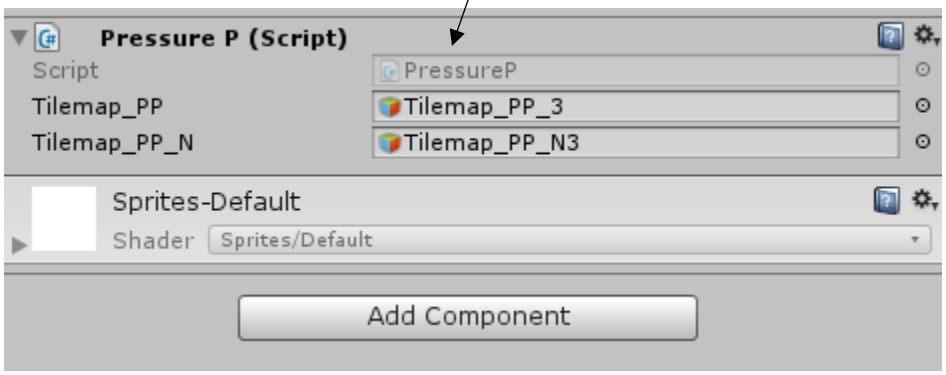
The two GameObjects that need to be called in. They are the On and Off versions of the pressure plates.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PressureP : MonoBehaviour {
6      public GameObject Tilemap_PP;
7      public GameObject Tilemap_PP_N;
8
9      void OnTriggerEnter2D (Collider2D col)
10     {
11         if (col.gameObject.name.Equals("Player"))
12         {
13             Tilemap_PP.SetActive(false);
14             Tilemap_PP_N.SetActive(true);
15         }
16     }
17     void OnTriggerExit2D (Collider2D col)
18     {
19         if (col.gameObject.name.Equals("Player"))
20         {
21             Tilemap_PP.SetActive(true);
22             Tilemap_PP_N.SetActive(false);
23         }
24     }
25 }
```

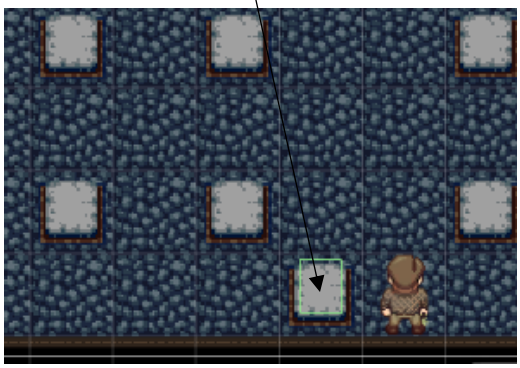
When the Player collides with the 2D box collided then, the off pressure plate (pushed up) deactivates and the on pressure plate (pressed down) activates.

Opposite as line 13 and 14 when the Player exits the 2D box collider.

Setting the parameter GameObjects for the script that is on each of the 4 plates on the left.



I also created one more pressure plate on its own which will be the start plate which the player must press down the start the puzzle sequence on the right.



Now that the pressure plates work with the 2D box colliders and the Player, I now needed to make the random puzzle pattern of 4 plates on the right.

To do this I needed some sort of delay code that will delay the code for activating and deactivating a pressure plate. This is so that the player can actually see the plates being pressed down and then popped pack up. If there was no delay then the user wouldn't be able to see the change from deactivate to activate as it would happen too fast for the human eye to see.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class pressurePlay : MonoBehaviour {
6      public GameObject Tilemap_Play;
7      public GameObject Tilemap_Play_N;
8
9      void OnTriggerEnter2D (Collider2D col)
10     {
11         if (col.gameObject.name.Equals("Player"
12         {
13             Tilemap_Play.SetActive(false);
14             Tilemap_Play_N.SetActive(true);
15             StartCoroutine("blockGame");
16         }
17     }
18     IEnumerator blockGame(){
19     yield return new WaitForSeconds(5);
20     print("waited 5 seconds");
21     }
22 }
23
24
25
```

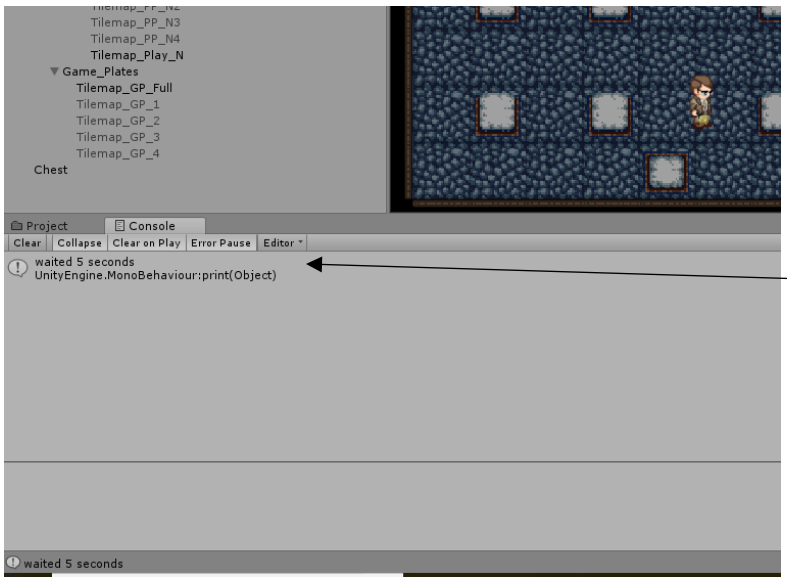
This is the script that will be used to create the random 4 plate puzzle that will play when the start pressure plate collies with the player.

This "IEnumerator blockGame" is a function that is specific to delaying line execution. The function is called "blockGame" and is called when the start pressure plate collides with the Player.

It is done by line 19 which, In this case, it waits 5 second and then prints ("waited 5 seconds").

[10]

[10] https://docs.unity3d.com/ScriptReference/WaitUntil.html - Date accessed 06/04/18 - Enumerator function

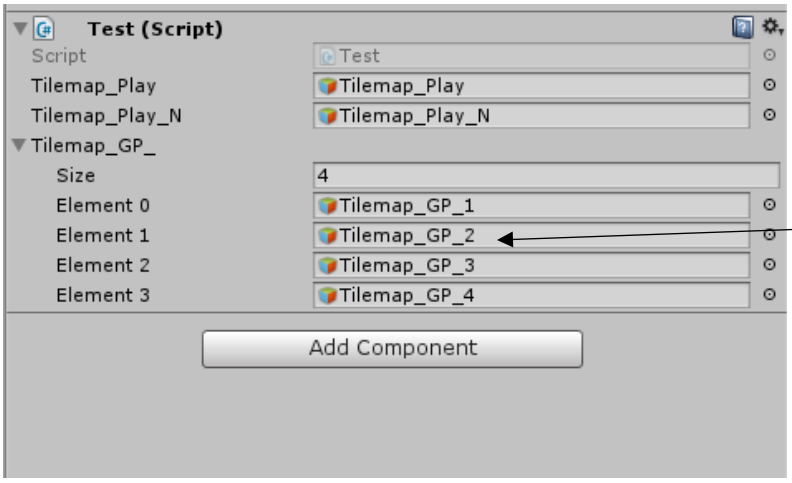


Showing that when the player collides with the start pressure plate, it runs the script and after 5 seconds the console outputs “waited 5 seconds”.

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

public class pressurePlay : MonoBehaviour {
    public GameObject Tilemap_Play;
    public GameObject Tilemap_Play_N;
    public GameObject Tilemap_GP_1;
    public GameObject Tilemap_GP_2;
    public GameObject Tilemap_GP_3;
    public GameObject Tilemap_GP_4;

    void OnTriggerEnter2D (Collider2D col)
    {
        if (col.gameObject.name.Equals("Player"))
        {
            Tilemap_Play.SetActive(false);
            Tilemap_Play_N.SetActive(true);
            StartCoroutine("blockGame");
        }
    }
    IEnumerator blockGame(){
    yield return new WaitForSeconds(1);
    Tilemap_GP_1.SetActive(true);
    yield return new WaitForSeconds(1);
    Tilemap_GP_2.SetActive(true);
    Tilemap_GP_1.SetActive(false);
    yield return new WaitForSeconds(1);
    Tilemap_GP_3.SetActive(true);
    Tilemap_GP_2.SetActive(false);
    yield return new WaitForSeconds(1);
    Tilemap_GP_4.SetActive(true);
    Tilemap_GP_3.SetActive(false);

    }
}
```

Calling in all the 4 right plates and along with the start plates on and off.

After I knew how to delay code lines, I moved onto developing the script to implement all 4 of the right plates.



Setting up the parameter GameObject for the script.

```
23      IEnumerator blockGame()
24      {
25          while (Number < 4)
26          {
27              StartCoroutine("randomObjectFalse");
28              yield return new WaitForSeconds(0.5f);
29              StartCoroutine("randomObjectTrue");
30              yield return new WaitForSeconds(0.5f);
31              Number = Number + 1;
32          }
```

I've made the IEnumerator function more efficient by using a while loop instead which has significantly reduced the lines of code needed.

It runs the lines 27 to 31 which call a the "randomObjectFalse" function, waits 0.5 seconds and then calls the "randomObjectTrue" functions, waits 0.5 seconds and then adds 1 to the Number integer. This will run through until Number equals 4 which means it will run 4 times as Number starts off as 0.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Test : MonoBehaviour {
6      public GameObject Tilemap_Play;
7      public GameObject Tilemap_Play_N;
8      public GameObject[] Tilemap_GP_;
9      private int currentIndex = 0;
10     private int Number = 0;
11
12
13     void OnTriggerEnter2D (Collider2D col)
14     {
15         if (col.gameObject.name.Equals("Player"))
16         {
17             Tilemap_Play.SetActive(false);
18             Tilemap_Play_N.SetActive(true);
19             StartCoroutine("blockGame");
20         }
21     }
22
23     IEnumerator blockGame()
24     {
25         while (Number < 4)
26         {
27             StartCoroutine("randomObjectFalse");
28             yield return new WaitForSeconds(0.5f);
29             StartCoroutine("randomObjectTrue");
30             yield return new WaitForSeconds(0.5f);
31             Number = Number + 1;
32         }
33         Tilemap_GP_[currentIndex].SetActive(false);
34     }
35
36     void randomObjectFalse()
37     {
38         int newIndex = Random.Range(0, Tilemap_GP_.Length);
39         Tilemap_GP_[currentIndex].SetActive(false);
40         currentIndex = newIndex;
41
42     }
43
44     void randomObjectTrue()
45     {
46         Tilemap_GP_[currentIndex].SetActive(true);
47     }
48 }
```

Next, I called in the Tilemap GaneObjects that I needed. Line 6 is the Start (on) plate and line 7 is the (off) plate.

Line 8 is the Tilemap for each of the Right puzzle plates. It is special because it has "[ ]" which means that I will use this to call any of the 4 plates depending on the random number generated between 0 and 3.

Line 9 just creates and sets the currentIndex integer which will be used to call on of the 4 puzzle plates.

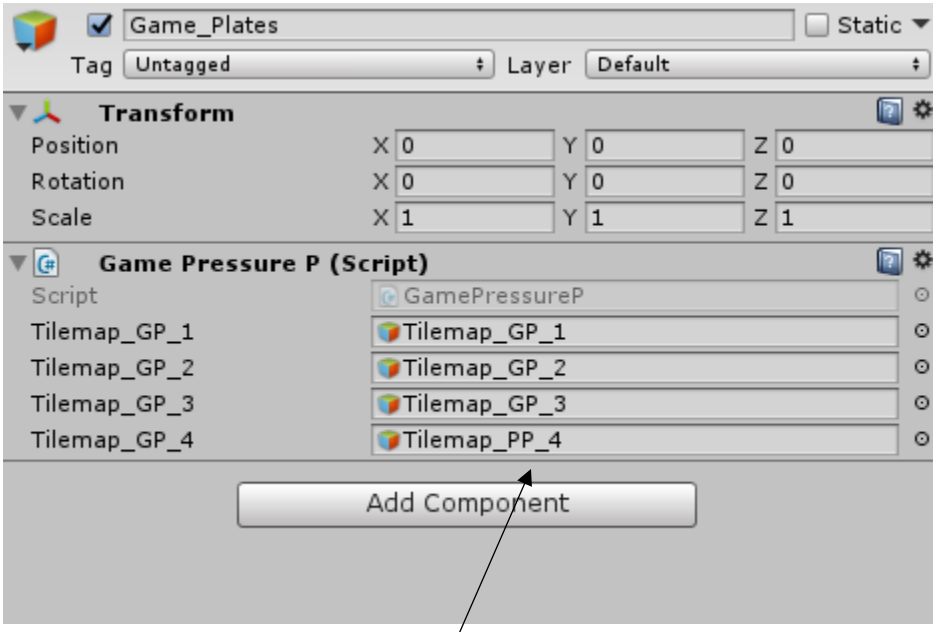
Line 10 Number integer is for the while loop in the function "blockGame"

The "randomObjectFalse" function generates a random number between 0 and 3 and then deactivates the "Tilemap_Gp_[random integer generated]". This way the puzzle plate Tilemap that is deactivated is totally random. It then sets the "currentIndex" to equal "newIndex" on line 40. This is to deactivate the last puzzle plate at the end of the while loop for the "blockGame" function, as well as to set the Tilemap to activate in the "randomObjectTrue" function.

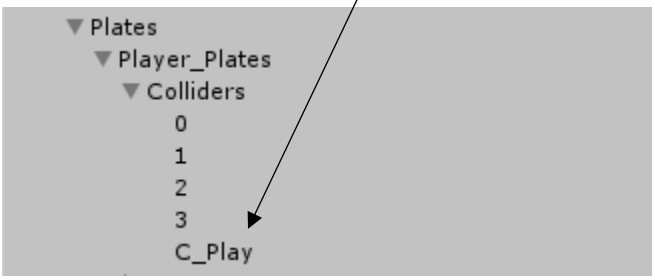
The "randomObjectTrue" function activates the "Tilemap_GP_[currentIndex]" which all depends on the randomly generated integer from "newIndex".

In simple terms, when the player collides with the Start Plate it activates the “blockGame” function. This then calls the “randomObjectFalse” function which generates a random integer between 0 and 3. Then the Tilemap_GP[currentIndex] is deactivated, then sets the currentIndex to newIndex. After, it waits 0.5 seconds and then calls the “randomObjectTrue” function which just activates the Tilemap_GP_[curentIndex] and then waits another 0.5 second and then loops again another 3 times.

The 0.5 second delays are to separate the deactivating of the Tilemap to show it has been pressed down and then 0.5 seconds later it pops back up by activating it again.



Linking in the relevant Tilemap_GP GameObjects to the Game Pressure P script (I’ve renamed the script to this). This script is on the “C_Play” GameObject.



Next, was to make it so that every Tilemap_GP involved in the puzzle was store in a list in the chronological order that they were played in. This is to later compare it with the Player's left plate list to see if they are equal.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using UnityEngine;
4
5   public class pressureGame : MonoBehaviour {
6       public GameObject Tilemap_Play;
7       public GameObject Tilemap_Play_N;
8       public GameObject[] Tilemap_GP_;
9       public int currentIndex = 0;
10      private int Number = 0;
11      public List<string> Order = new List<string>();
12
13      void OnTriggerEnter2D (Collider2D col)
14      {
15          if (col.gameObject.name.Equals("Player"))
16          {
17              Tilemap_Play.SetActive(false); // look of pressure plate being pressed down
18              Tilemap_Play_N.SetActive(true); // look of pressure plate being pressed down
19              StartCoroutine("blockGame"); // calls blockGame
20          }
21      }
22
23      IEnumerator blockGame()
24      {
25          while (Number < 4)
26          {
27              StartCoroutine("randomObjectFalse"); //calls randomObjectFalse
28              string orderList  = currentIndex.ToString();
29              Order.Add(orderList);
30              yield return new WaitForSeconds(0.5f); // waiting 0.5s
31              StartCoroutine("randomObjectTrue"); // calls randomObjectTrue
32              yield return new WaitForSeconds(0.5f);
33              Number = Number + 1; // reducing "Number" for while loop
34          }
35          Tilemap_GP_[currentIndex].SetActive(false); // at the end of the pattern it then turn false
36          Number = 0; // allows for reset
37      }
38
39      void randomObjectFalse()
40      {
41          int newIndex = Random.Range(0, Tilemap_GP_.Length); // random index for length set
42          Tilemap_GP_[currentIndex].SetActive(false); // setting "currentIndex" gameObject to false
43          currentIndex = newIndex; // Setting new currentIndex
44      }
45
46      void randomObjectTrue()
47      {
48          Tilemap_GP_[currentIndex].SetActive(true); // Setting currentIndex to true
49      }
50      void OnTriggerExit2D (Collider2D col)
51      {
52          if (col.gameObject.name.Equals("Player"))
53          {
54              Tilemap_Play.SetActive(true);
55              Tilemap_Play_N.SetActive(false);
56
57          }
```

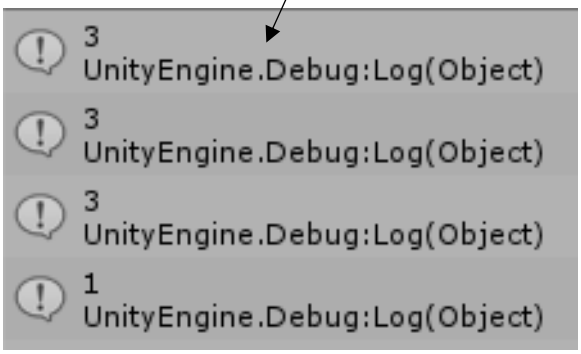
Creating a static string list called "Order"

Line 28 & 29. Creating a local string called "orderList" and setting it to the currentIndex.ToString which is just the currentIndex but converted to a string.

Then it adds "orderList" to the list "Order". As this loop 4 times it will add all the currentIndex generated by

Testing

I added temporarily some code that printed out each index in the "Order" list at the end of the while loop to check if it was working. As shown below it does add 4 strings to the list which are either 0, 1, 2 or 3, which are the currentIndex values used in the random puzzle.

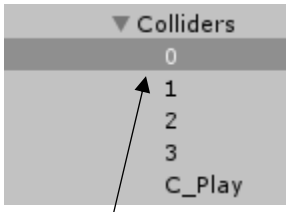


PRESSURE PLATE PLAYER CODE

Now that I fully developed the Puzzle Plates, I needed to develop the Player Plates that are on the left. To do this was very similar to the Puzzle Plates before.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class PressureP : MonoBehaviour {
6      public GameObject Tilemap_PP;
7      public GameObject Tilemap_PP_N;
8      public static List<string> userOrder = new List<string>();
9      private int i = 0;
10
11
12     void OnTriggerEnter2D (Collider2D col)
13     {
14         if (col.gameObject.name.Equals("Player"))
15         {
16             Tilemap_PP.SetActive(false);
17             Tilemap_PP_N.SetActive(true);
18             userOrder.Add(gameObject);
```

The 4 GameObjects for the left colliders



Line 18 adds the gameObject name to the list when the Player collides with its 2d box collider.

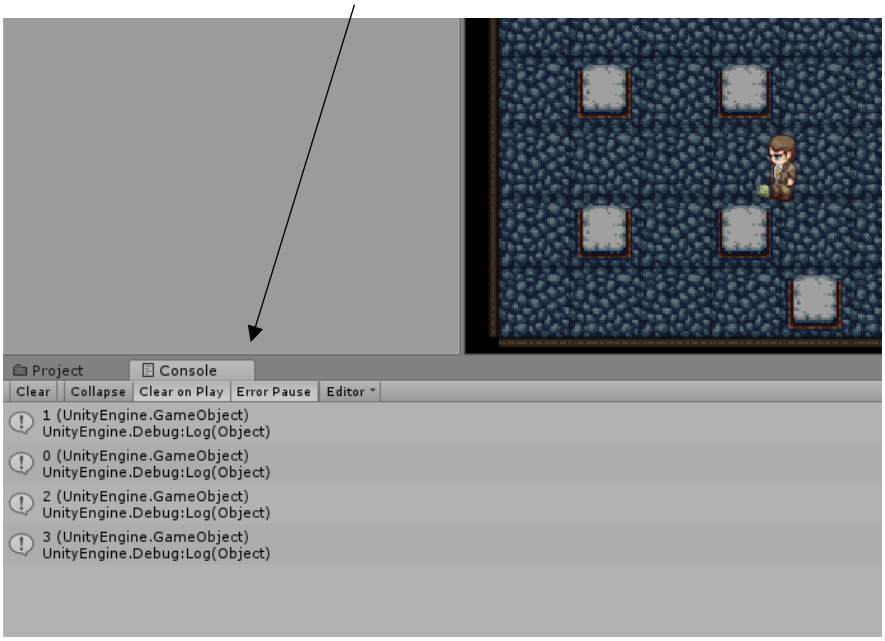
This script above has 4 parameters, line 6 and 7 are the on and off Tilemaps on the Left, line 8 is the static string list and line 9 is a local integer called "i". The reason this list is static is because I must be static to be able to be used/changed over other scripts. As this script will be on all the 4 left plate Tilemaps , for each to add to the list, it must be static, otherwise the list would be only locally added to and so there would be essentially 4 different list contents.

11

[11] https://answers.unity.com/questions/588294/anyway-to-access-a-list-from-another-script-c.html - Date accessed - 08/04/18 – Static lists

Testing

I added temporarily some code that printed out each index in the "userOrder" list at the end to see if it works. As you can see there are the first 4 index values of the userOrder lost has been printed out and so it's working.



CHECKING LISTS ARE EQUAL

Now that both the left and right pressure plates create a string list containing 4 values, I next need to add to the function to the "PressureP" script that will check to see if they are equal. If they are equal then the Player has correctly copied the puzzle pattern show to them on the right.

```
void Update() {


    if(Input.GetKeyDown("t"))
    {
        var Full = userOrder.Any(); // checking if the list is empty.
        if (Full == false)
        {
            Debug.Log("Checking");
        }
        else
        {
            var correctPattern  = userOrder.SequenceEqual(pressureGameScript.Order); // checking to see if both lists are equal.
            Debug.Log("This is"+correctPattern);
```

First checks that the list is not empty. userIrder.Any checks to see if there are any values stored in the list.

If the "userOrder" list is not empty it runs the else statement which checks if the "userOrder" list is equal to the "Order" list. The reason its "pressureGameScript.Order is because that's how to call a variable in from another script if it's not static. If the lists are equal then the local variable "correctPattern" is true, if it doesn't match, its false.

```
    public pressureGame pressureGameScript;
```

This is the parameter for calling in the Order list from the other script.

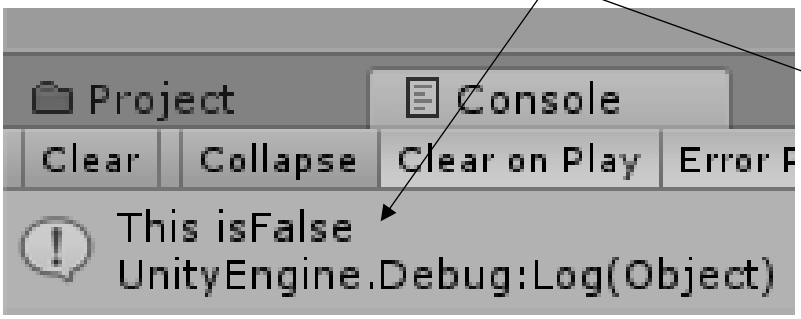
The “SequenceEqual” checks if the sequence of a two lists are equal. It is an imported function from a library which is imported by “using System.Linq; “

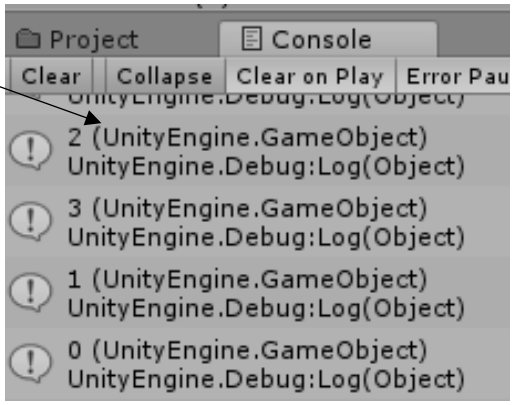
Testing

I played the level, pressed down the Start Plate with the Player GameObject, waited for the pattern on the right to play and then inputted the correct pattern on the left. This was all working fine however, when I went to check if they were equal it always outputted “This is False” even when I correctly inputted the pattern.

This confused me for a very long time as I had no idea why it wasn’t working because it outputted the same index values for both lists and so at first I thought it was a problem with the “SequenceEqual” function.

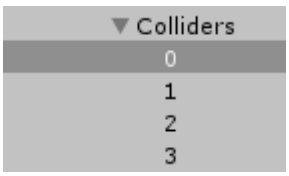
After many hours of testing what was going wrong I found out the reason. The reason was because the index values for both lists were actually not the same. This was because the “Order” list contained only 0, 1, 2 and 3 which is the right pattern plates but the userOrder list which was the players inputted plats were actually a very long string, I just didn’t notice the difference stupidly.





Instead of adding just a string of either 0, 1, 2 or 3 which is what I called the GameObjects it actually was adding 0, 1, 2 or 3 and then (UnityEngine.GameObject). I just didn’t actually notice the difference.

Therefore, to fix this I needed to find out how to remove the extra characters from the string.



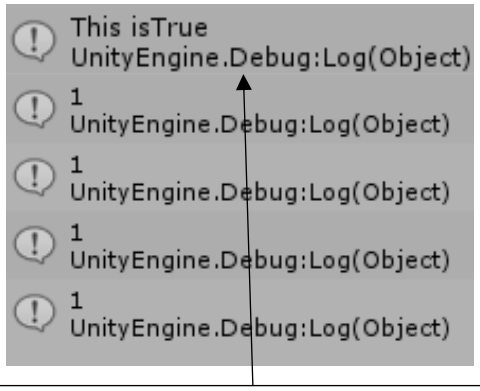
The GameObject names.

To do this I required a line that removed characters from a string which is the function "nameOfString.Remove (start, end)". The (start , end) is range index of the characters to be removed from the string.

In my case, the range was 1 to 25 as I wanted to keep the index 0 which was the number (0, 1, 2 or 3) and the end of the string is index 25. This means that all charters from the string will be removed, leaving only one character which I wanted.

```
string Rush  = gameObject.ToString();
Rush = Rush.Remove(1,25); // removes everything but index 0
userOrder.Add(Rush);
```

This is the lines of code that will convert the gameObject name to a string, remove the unwanted characters and then add the wanted one character to the "userOrder" list.



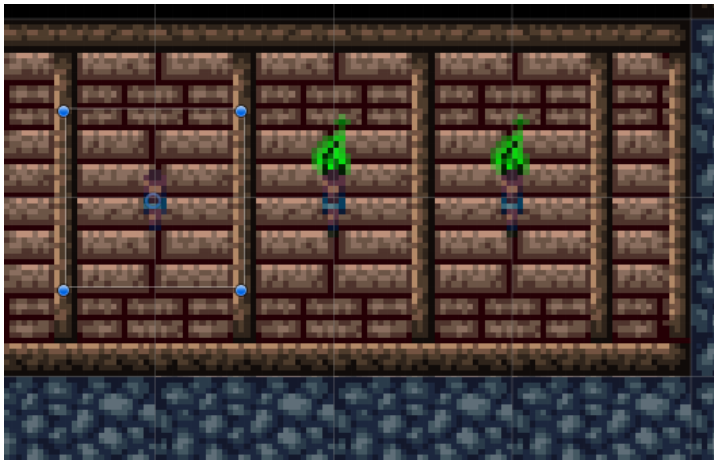
As shown above, now when I correctly input the puzzle pattern it does output "This is true"

THE THREE TORCHES

After making the lists and then be able to compare them to see if the User was correct or not, I needed to develop a way for the user to know if they were correct and what stage of the puzzle they are at.

To do this I wanted to create three green torches that will light when the player gets one puzzle pattern right and when the user manages to get all three lit, the stairway will open up and allow the user to exit the level.

However, I wanted to make it so that if the user gets the pattern wrong at any part of the three puzzles, the torches will turn all off, forcing the player start to start from the beginning again. If the user manages to get all three puzzle patterns correct without any mistakes then I want a chest to appear alongside the stairway to allow the users to collect a Rune.



The green torches, one unlit and two lit

Before I created the code for the torches I needed to actually fully develop the puzzle code to allow it to fully loop though 3 times for each puzzle.

```
21      void OnTriggerEnter2D (Collider2D col)
22      {
23          if (col.gameObject.name.Equals("Player"))
24          {
25              Tilemap_PP.SetActive(false);
26              Tilemap_PP_N.SetActive(true);
27          }
28          if (Stopper == false)
29              if (col.gameObject.name.Equals("Player"))
30              {
31                  string Rush  = gameObject.ToString();
32                  Rush = Rush.Remove(1,25); // removes everything but index 0
33                  userOrder.Add(Rush);
34                  Checker = Checker + 1;
35              }
36
37          if ((Checker == 4)&& (Stopper == false))// stops loop when 4 inputs have been inputted
38          {
39              Stopper = true;
40              var Full = userOrder.Any(); // checking if the list is empty.
41              if (Full == false)
42              {
43
44              }
45              else
46              {
47                  var correctPattern  = userOrder.SequenceEqual(pressureGameScript.Order); // checking to see if both lists are equal
48                  if (correctPattern == true)
49                  {
50                      Torch = Torch + 1; // turning on next torch.
51                      userOrder.Clear(); // resetting the list.
52                      pressureGameScript.Order.Clear(); // resetting the list.
53                      Stopper = false; // resetting for next stage.
54                      Checker = 0; // resetting for next stage.
55                  }
56                  else
57                  {
58                      firstTry = false; // user can never access the chest.
59                      correctPattern = false;
60                      userOrder.Clear(); // resetting the list.
61                      pressureGameScript.Order.Clear(); // resetting the list.
62                      Stopper = false; // resetting for next stage.
63                      Checker = 0; // resetting for next stage.
64                      Torch = 0; // resetting the torch.
65                  }
66              }
```

What I've added to the script is a few variables that are used to call other lines of the script. These are the "Checker", "Stopper", "Torch", "firstTry" and "Full". The "Checker" is an integer variable that has the value "1" added to it every time the Player collides with the Pressure Plate Player (one of them on the left) until it equals the value "4" because when this happens "Stopper" equals "True" on line 39.

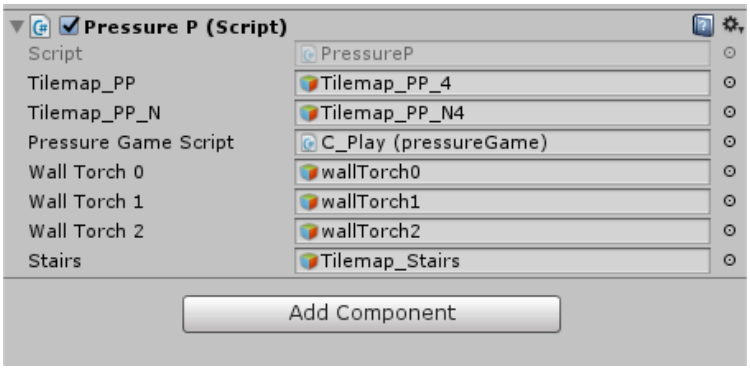
The “Stopper” function is used to allow the line 29 to 24 run when it’s “False” and when it is “True” prevent it from running. This is so that the string list only ever has a maximum of 4 characters within it, in index 0 to 3. The “Full” local variable is to make sure the List isn’t empty. Finally, the “Torch” integer has “1” added to it each time the user gets a puzzle correct but resets to “0” if at any time the “correctPattern” equals False. The “firstTry” Boolean variable starts off as True but if the user makes any mistake and the “correctPattern” equals False, it gets set to False. This is so that if the “firstTry” variable equals True when the three torches are lit, it reveals that chest.

When the “correctPattern” is set and either lines 50 to 54 or 58 to 64 are executed, the two lists, “userOrder” and “Order” are both cleared with lines 52 and 52 or 60 and 61. This is so that the next puzzle can be played and new values can be added to the lists and checked. Also, the “Stopper” becomes “False” and the “Checker” becomes “0”, to allow the code above to be executed again.

```
69      void Update() {
70          if (Torch == 0)
71              {
72                  wallTorch0.SetActive(false);
73                  wallTorch1.SetActive(false);
74                  wallTorch2.SetActive(false);
75              }
76          if (Torch == 1)
77              {
78                  wallTorch0.SetActive(true);
79              }
80          if (Torch == 2)
81              {
82                  wallTorch1.SetActive(true);
83              }
84          if (Torch == 3)
85              {
86                  wallTorch2.SetActive(true);
87                  Stairs.SetActive(true);
88                  if (firstTry == true)
89                  {
90                      Chest.SetActive(true);
91                  }
92              }
93          }
```

This is also within the same script in the Update function. It is using the “Torch” integer variable that is set above and that I’ve mentioned previously.

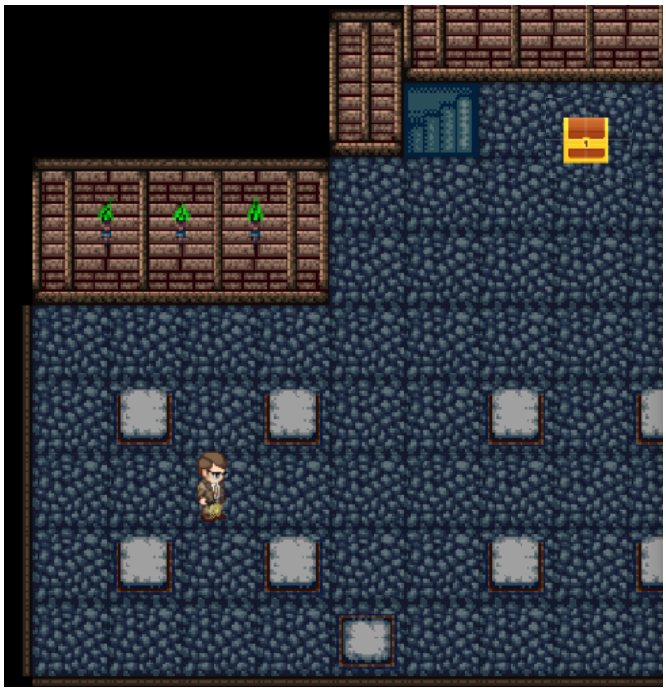
When the “Torch” equals “0” it deactivates all torches, this is when the user gets the pattern wrong. When the “Torch” equals “1” it activate the first Torch to the left. When “Torch” equals “2” it activates the middle Torch. Finally, when the “Torch” equals “3” it activates the end Torch and activate the Stairs GameObejct. It also checks to see if the “firstTry” Boolean variable equals “True” to see if the user is allowed to chest or not, if it’s “true” then the Chest GameObject activates.



Setting up the parameters of the three Torch GameObjects and the Stairs.

Testing

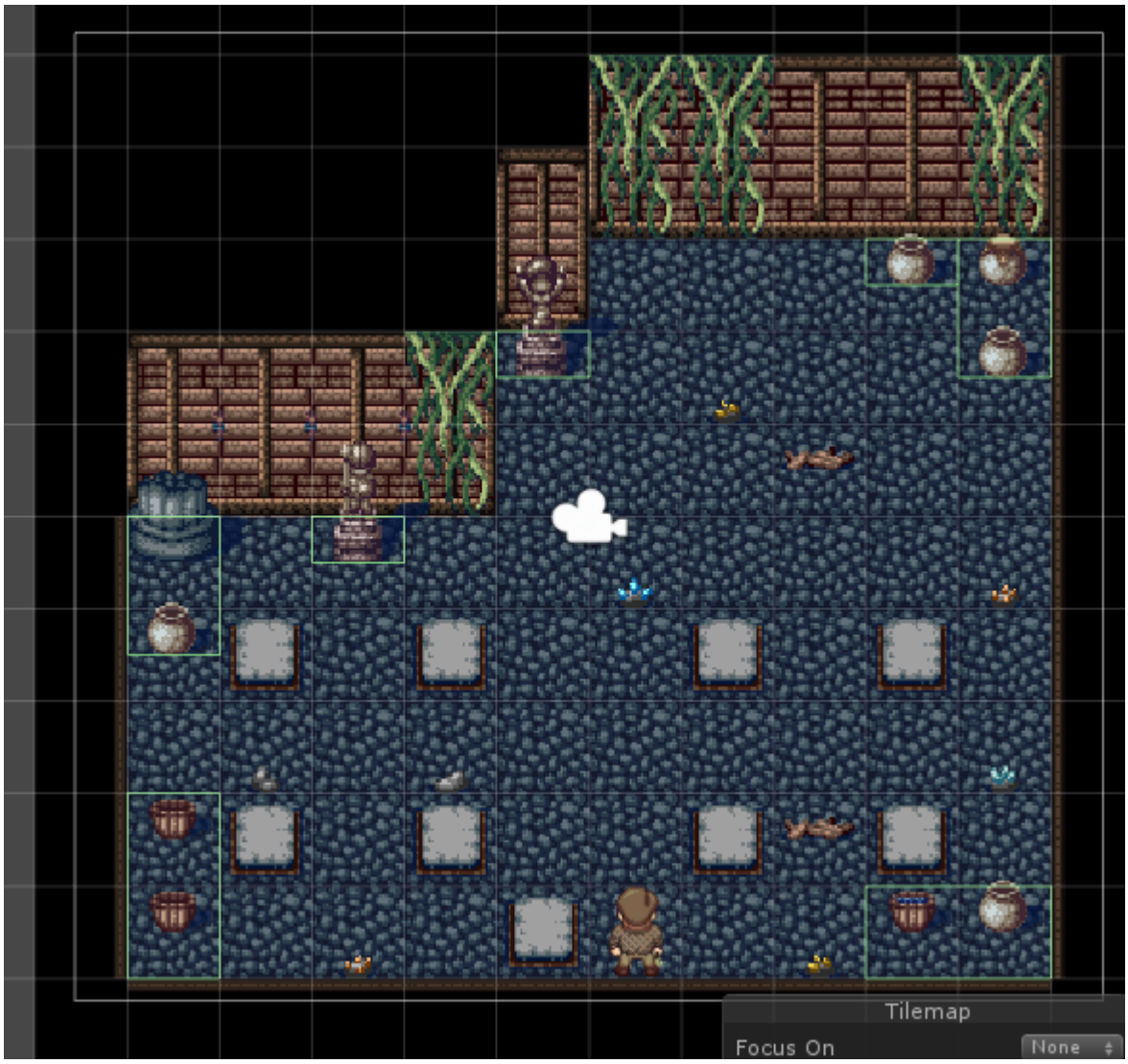
Showing that the Player got all three puzzle patterns correct in a row but didn't get it on their first try and so the chest hasn't appeared.



Showing that the Player got all three puzzle patterns correct in a row and did it on their first try and so the chest has appeared.

FINAL LOOK OF LEVEL 2

Final look of level 2 when the player loads the scene. I've just added some athletics to make the level look more appealing.



MEETING SUCCESS CRITERIA

I am confident that for Level 2 I've met all the success criteria relevant to this section of development that I have developed so far.

- Documentation of error trapping – The scripts for level 2 didn't need much error trapping as the inputs of the user are controlled, only the 4 left pressure plate GameObjects. Therefore, there wasn't any need for error trapping as the user couldn't enter other values. The only error trapping was that the index only can have 4 characters, one in each index from 0 to 3.

- Documentation of testing – I have clearly shown testing throughout the development after every major function or script, to ensure that all of my code is working fully to allow the player to play Rune Raid as smoothly as possible.

- Documentation of errors whilst in development – Any errors that I've come across I've documented and then given my explanation of the error. Then proceed to fix the error, explaining the methods I took to fix them.

- The player will be able to fully a interact with all of Rune Raid's aspects – All the aspects I've made the player can fully interact with, the Chest and Pressure Plates can all be used by the player to attempt to complete the level.

- All algorithms will be as efficient as I can possibly make them – All the algorithms that I've made, I believe are of the highest efficiently that I can make.

- All decisions made will be shown and justified – Every significant decision that I've made, I have documented my thoughts and justified the reasons for my decisions.

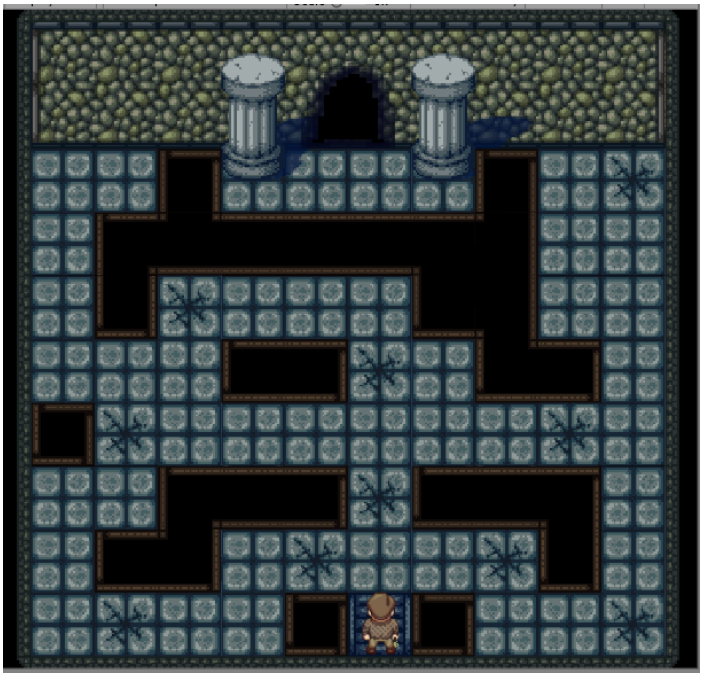
- Rune Raid's puzzles must be challenging for the player – For this level, I believe I've made the puzzle significantly harder than Level 1's puzzle and I believe the pattern puzzle actually requires a lot of attention to complete, especially if the player wants to access the special chest.

- All the code is split up into specific functions within modules – Every script has their own functions within them that are grouped effectively to by relevance and each different aspect has their own script.

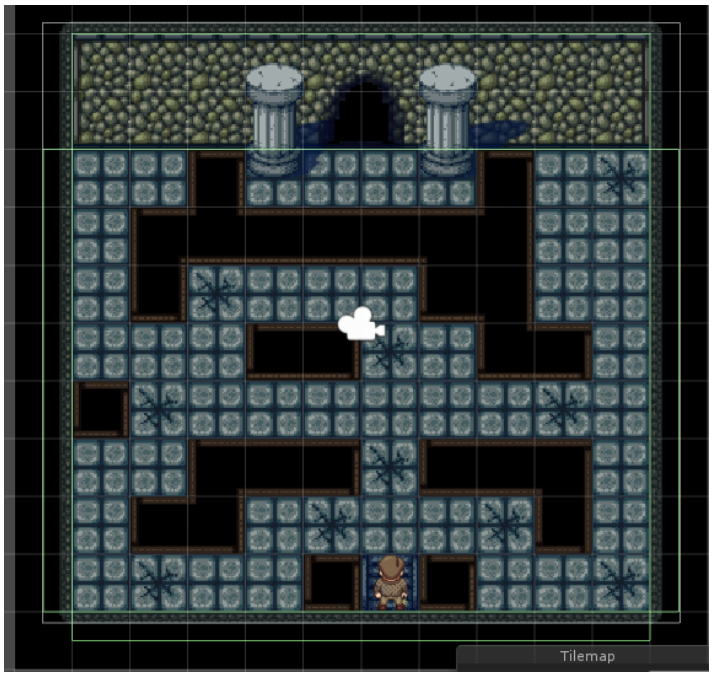
Level 3

For this level, my idea is to make a few monsters that the player must try and avoid touching whilst solving another pattern puzzle, but this time with symbols that will be on the walls instead and the user must work out the pattern to cross over the holes to get to the exit.

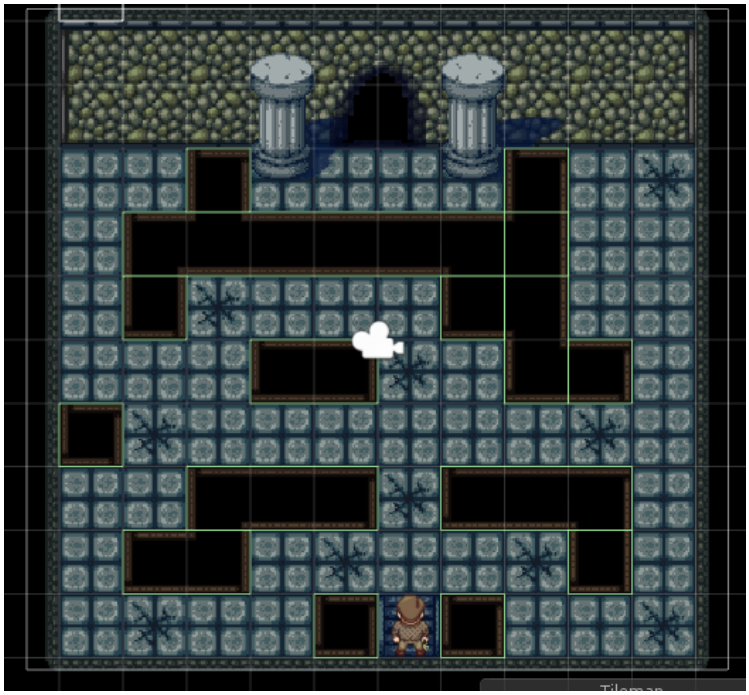
LEVEL DESIGN



The first design I made of the level, with just its foundation Tilemaps, Floor, Walls, Holes and the Pillars.

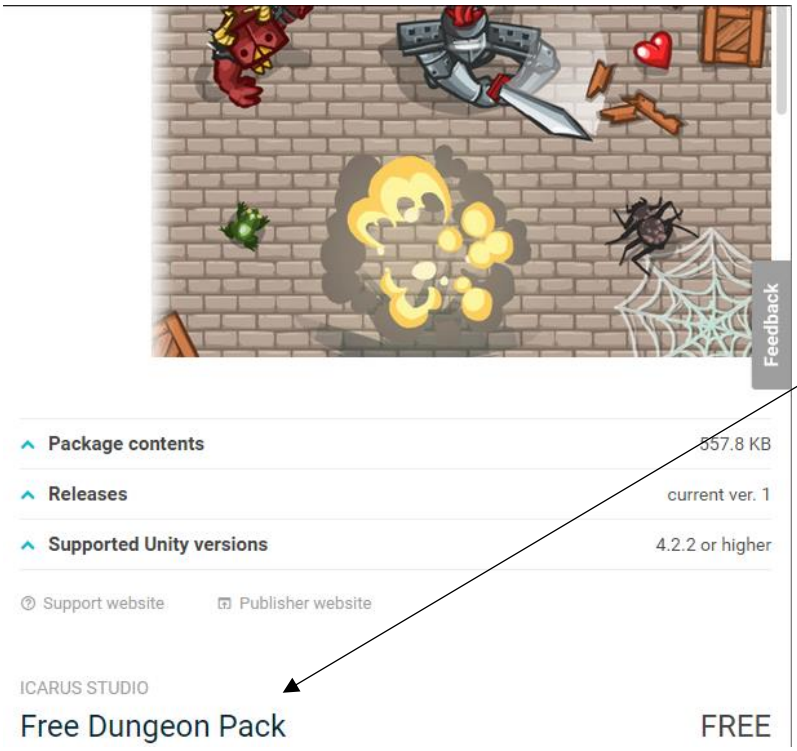


Adding in the wall 2D box colliders.



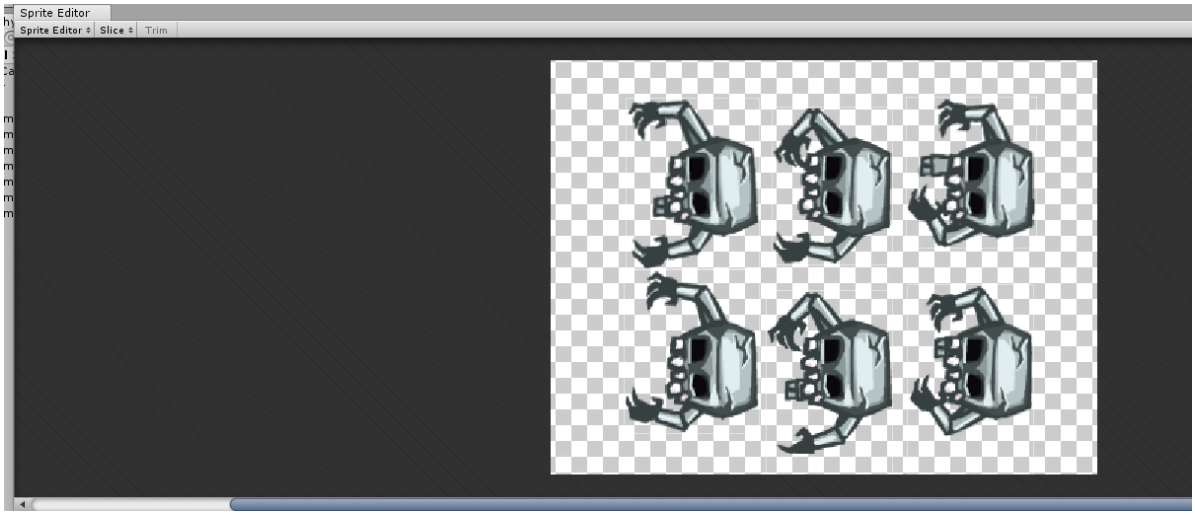
Now adding in the Hole 2D box colliders to prevent the player from moving over them.

MONSTERS



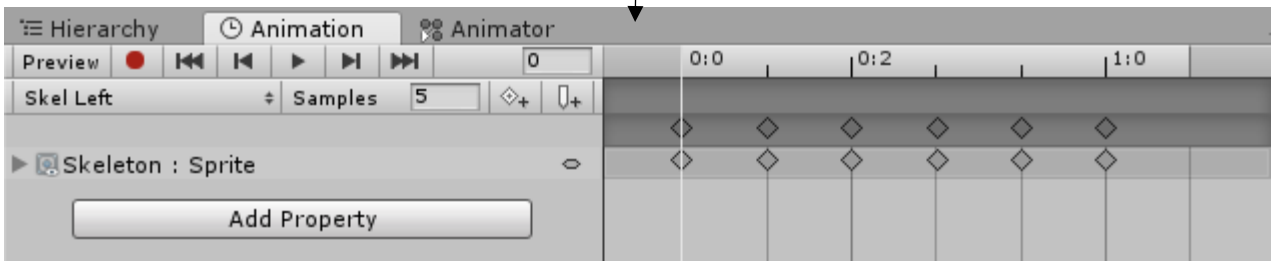
I decided to have to monster as skeletons and so I imported an asset pack from the asset store that contained a cartoon skeleton with many frames for walking animation.

First, I put the skeleton image into the sprite editor to separate each section into its own image to be used in the Animation component for the walking animation.



As the asset pack only contained the skeleton moving to the left, I needed to edit them to rotate the images so that I had a left, right, up and down skeleton image. As shown below I have made the skeleton frames in each direction.

I then added the skeleton separate images in the Animation component to build the direction movement animations. As shown below, I've made the Left animation by adding in all the image frames for the left animation.



After the animations were working, I next needed to develop some sort of looping movement which the skeletons will follow repeatedly. However, I already knew that I couldn't just use a loop in a function to move them, this is because it will run out of memory as the looping will rapidly go on forever.

Therefore, I needed a more efficient way to make a repeating movement which won't cause a crash due to running out of memory.

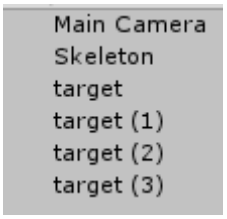
After some researching, I found out a very efficient way to do this by watching a YouTube video on the topic.

[12]

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5
6
7  public class SkeletonMove : MonoBehaviour
8  {
9      public Transform[] target;
10     public float speed;
11     private int currentPosition;
12
13     void Update (){
14         if (transform.position != target[currentPosition].position) // move until the skeleton reaches the target position.
15         {
16             Vector3 pos = Vector3.MoveTowards(transform.position, target[currentPosition].position, speed * Time.deltaTime);
17             GetComponent<Rigidbody2D>().MovePosition(pos);
18         }
19         else
20         {
21             currentPosition = (currentPosition + 1) % target.Length; // the target position is now reached and so move to the next target.
22         }
23     }
24
25 }
26
27
```

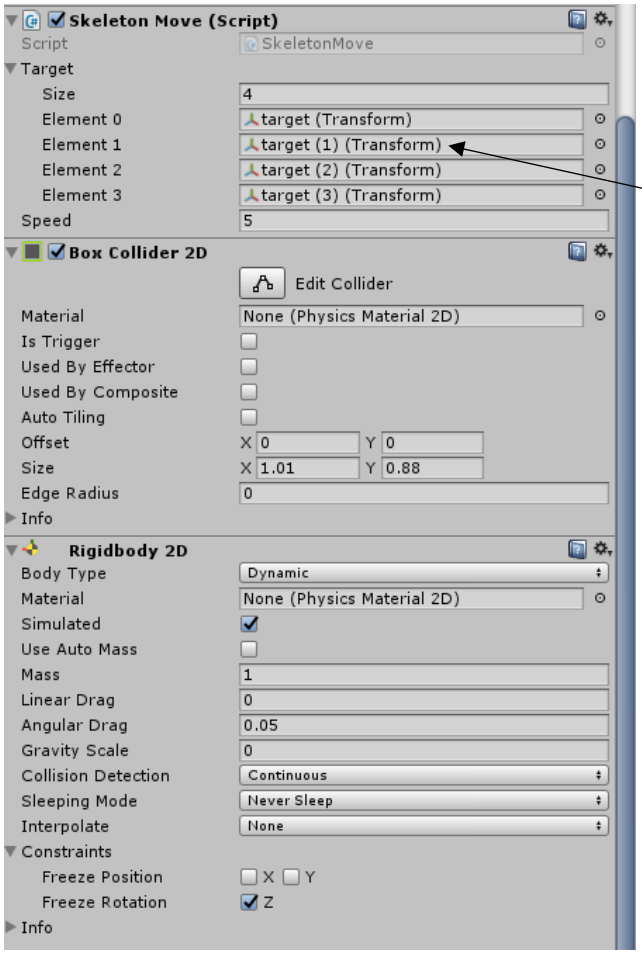
The script translates the Skeleton GameObject towards the target GameObjects position by using line 16. Line 16 moves the GameObject by Vector3, towards the Target [ ] GameObjects position.

This script utilises GameObjects to build a path for the skeleton to move towards and then when it meets the target, it changes to the target and moves to its location.

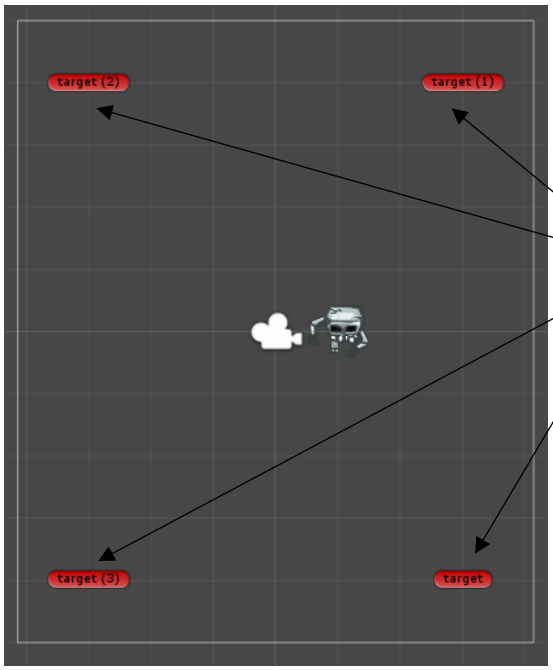


The target GameObjects.

[12] https://www.youtube.com/watch?v=fKWTpi70a_E&t=0s&list=LLDbedb7gvNzK43UXt9rTlzg&index=10 – Date accessed 08/04/18 – Target movement



Setting up the target parameters and setting the speed of the skeleton movement.



The four target GameObjects that will be used to build the square path that I want the Skeleton to follow.

Then, I positioned the targets where I want them to be in the level for the skeleton to follow.

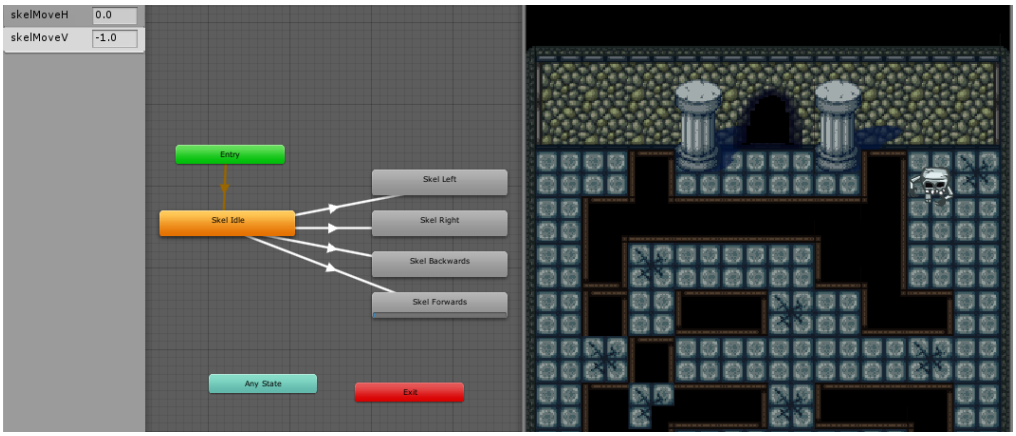
As I wanted three monster skeletons, I copied the GameObjects and just made it so that there was only 2 targets instead of 4, as the path I need is linear.

I did some testing to check that the scripts were working and as shown on the right, the Skeleton GameObject is moving towards the next target.

Now that the path movement for the Skeletons are working, I needed to add in the directional animations.

```
5   public class skelForAni : MonoBehaviour {
6   public static int skelMoveV;
7   public static int skelMoveH;
8   public Animator skel;
9
10  void Start () {
11
12      }
13
14      void OnTriggerEnter2D (Collider2D col)
15      {
16          if (col.gameObject.name.Equals("Skeleton"))
17          {
18              Debug.Log("Collding");
19
20              skelMoveV = -1;
21              skelMoveH = 0;
22              skel.SetFloat("skelMoveV",skelMoveV);
23              skel.SetFloat("skelMoveH",skelMoveH);
24          }
25
26      }
27  }
```

At first, this is the code that I developed where when the target 2D box collider collided with the Skeleton, it changed the “skelMoveV” value and the “skelMoveH” values to match the next direction and then sets the two floats for the Animation component to play the correct Animation.



Setting up the Animator so that the value of the two parameter floats decides which animation to play. If the skelMoveV = 1 then it will play the backwards animation because 1 in the Y direction is going upwards.

Making the Code More Efficient

However, after some thinking, it would be more efficient to just have one scrip that could be added to all the skeletons instead of each target.

```
7   public Animator skel;
8   public static int skelMoveV;
9   public static int skelMoveH;
10
11  void Start () {
12          skel = GetComponent<Animator>();
13
14      }
15  void OnTriggerEnter2D (Collider2D col)
16      {
17          if (col.gameObject.name.Equals("target"))
18          {
19              Debug.Log("Collding");
20
21              skelMoveV = 1;
22              skelMoveH = 0;
23              skel.SetFloat("skelMoveV",skelMoveV);
24              skel.SetFloat("skelMoveH",skelMoveH);
25          }
26          if (col.gameObject.name.Equals("target_1"))
27          {
28              Debug.Log("Collding");
29
30              skelMoveV = 0;
31              skelMoveH = -1;
32              skel.SetFloat("skelMoveV",skelMoveV);
33              skel.SetFloat("skelMoveH",skelMoveH);
34          }
35          if (col.gameObject.name.Equals("target_2"))
36          {
37              Debug.Log("Collding");
38
39              skelMoveV = -1;
40              skelMoveH = 0;
41              skel.SetFloat("skelMoveV",skelMoveV);
42              skel.SetFloat("skelMoveH",skelMoveH);
43          }
44          if (col.gameObject.name.Equals("target_3"))
45          {
46              Debug.Log("Collding");
47
48              skelMoveV = 0;
49              skelMoveH = 1;
50              skel.SetFloat("skelMoveV",skelMoveV);
51              skel.SetFloat("skelMoveH",skelMoveH);
52          }
53      }
54
```

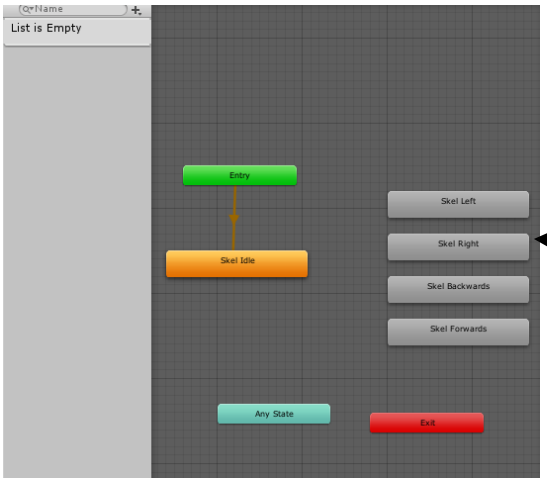
Calling in the Animator component.

When the Skeleton GameObject collides with one of the targets, it checks which one it has collided with and then sets the variables to play the specifc animation in the Animator.

However, I realised I could make it even more efficient by making it so that it directly played the animations from the code instead of using two parameters.

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

public class SkelAni : MonoBehaviour {

public Animator skel; // calling in animator


void Start () {
        skel = GetComponent<Animator>(); // setting animator

    }
void OnTriggerEnter2D (Collider2D col)
    {
        if (col.gameObject.name.Equals("target"))
        {
            skel.Play("Skel Backwards",-1,0f);
        }
        if (col.gameObject.name.Equals("target_1"))
        {
            skel.Play("Skel Left",-1,0f);
        }
        if (col.gameObject.name.Equals("target_2"))
        {
            skel.Play("Skel Forwards",-1,0f);
        }
        if (col.gameObject.name.Equals("target_3"))
        {
            skel.Play("Skel Right",-1,0f);
        }
    }
}

```

Now, when the Skeleton collides with one of the 2D box colliders it checks which target GameObject it has collided with and then directly plays the animation from the Animator.



The Animator without the transition links because they are now called directly and so no links are needed.

SYMBOL PUZZLE

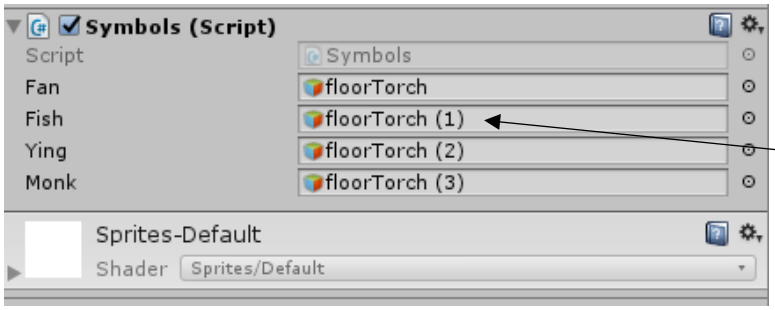
This symbol puzzle is a puzzle that the user must solve by looking at the symbols on the wall and then copying the symbols pattern on the floor. However, the twist is that the pattern is the order of the symbols on the wall right to left instead of the English reading left to right.

Firstly, I imported a collection of symbols image from the internet which I put in the sprite editor to separate each of the symbols to be used separately. These symbols will be used to from a pattern on the wall and on the floor, below 4 torches. Each torch will have one symbol below which will be linked to the torch by its 2D box collider. The player must turn off each torch in in the correct symbol pattern. If the player gets the pattern wrong, the torches all light and the player must try again.

The GameObject set up for the puzzle, there are just 4 sysmbols on the wall which are one GameObject. However, each of the floor symbols is its own GameObject and has a 2D box collider, which when the player collides with and pressed a button, it will turn off.

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

public class Symbols : MonoBehaviour {
    public GameObject Fan;
    public GameObject Fish;
    public GameObject Ying;
    public GameObject Monk;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame

    void OnTriggerStay2D (Collider2D col)
        {
            if (col.gameObject.name.Equals("floorFan")&&Input.GetKeyDown("q"))
            {
                Fan.SetActive(false);
            }
            if (col.gameObject.name.Equals("floorYing")&&Input.GetKeyDown("q"))
            {
                Ying.SetActive(false);
            }
            if (col.gameObject.name.Equals("floorMonk")&&Input.GetKeyDown("q"))
            {
                Monk.SetActive(false);
            }
            if (col.gameObject.name.Equals("floorFish")&&Input.GetKeyDown("q"))
            {
                Fish.SetActive(false);
            }
    }
}
```

This is the script which will deactivate a specific torch depending on which symbol the Player is standing on. If the Player is standing in the floorFish symbol and pressed “q”, then the Fish torch will deactivate and reveal the unlit torch on the order layer beneath.



Setting up the parameters for the script. Each of the parameters are set up to a torch GameObject.

Showing that the torches do turn off when then player stands on the symbol and pressed “q”.

The way the script will check if the player has turned off the torches in the correct way is to use a list in a similar way to the level 2 lists. First, two string lists are created, “Symbol” and “setSymbol”. “setSymbol” has 4 strings added to it when the script is first loaded, this is the pattern order for the puzzle. When the player turns off a torch, it adds a string to the “Symbol” list. This is so that when the Player turns off all four torches, it can compare the two lists and if they are equal, the player has solved the puzzle.

```
public List<string> Symbol = new List<string>();
public List<string> setSymbol = new List<string>();

// Use this for initialization
void Start () {
    setSymbol.Add("Fan");
    setSymbol.Add("Ying");
    setSymbol.Add("Monk");
    setSymbol.Add("Fish");
}

// Update is called once per frame

void OnTriggerStay2D (Collider2D col)
    {
        if (col.gameObject.name.Equals("floorFan")&&Input.GetKeyDown("q"))
        {
            Fan.SetActive(false);
            Symbol.Add("Fan");
        }
        if (col.gameObject.name.Equals("floorYing")&&Input.GetKeyDown("q"))
        {
            Ying.SetActive(false);
            Symbol.Add("Ying");
        }
        if (col.gameObject.name.Equals("floorMonk")&&Input.GetKeyDown("q"))
        {
            Monk.SetActive(false);
            Symbol.Add("Monk");
        }
        if (col.gameObject.name.Equals("floorFish")&&Input.GetKeyDown("q"))
        {
            Fish.SetActive(false);
            Symbol.Add("Fish");
        }
}
```

```
public Bucket water;


// Use this for initialization
void Start () {
    setSymbol.Add("Fish");
    setSymbol.Add("Monk");
    setSymbol.Add("Ying");
    setSymbol.Add("Fan");
}

// Update is called once per frame

void OnTriggerStay2D (Collider2D col)
    {
        if (col.gameObject.name.Equals("floorFan")&&Input.GetKeyDown("q")&&water.Water == true)
        {
            Fan.SetActive(false);
            Symbol.Add("Fan");
            puzzle = puzzle + 1;

        }
        if (col.gameObject.name.Equals("floorYing")&&Input.GetKeyDown("q")&&water.Water == true)
        {
            Ying.SetActive(false);
            Symbol.Add("Ying");
            puzzle = puzzle + 1;
        }
        if (col.gameObject.name.Equals("floorMonk")&&Input.GetKeyDown("q")&&water.Water == true)
        {
            Monk.SetActive(false);
            Symbol.Add("Monk");
            puzzle = puzzle + 1;
        }
        if (col.gameObject.name.Equals("floorFish")&&Input.GetKeyDown("q")&&water.Water == true)
        {
            Fish.SetActive(false);
            Symbol.Add("Fish");
            puzzle = puzzle + 1;
        }
```

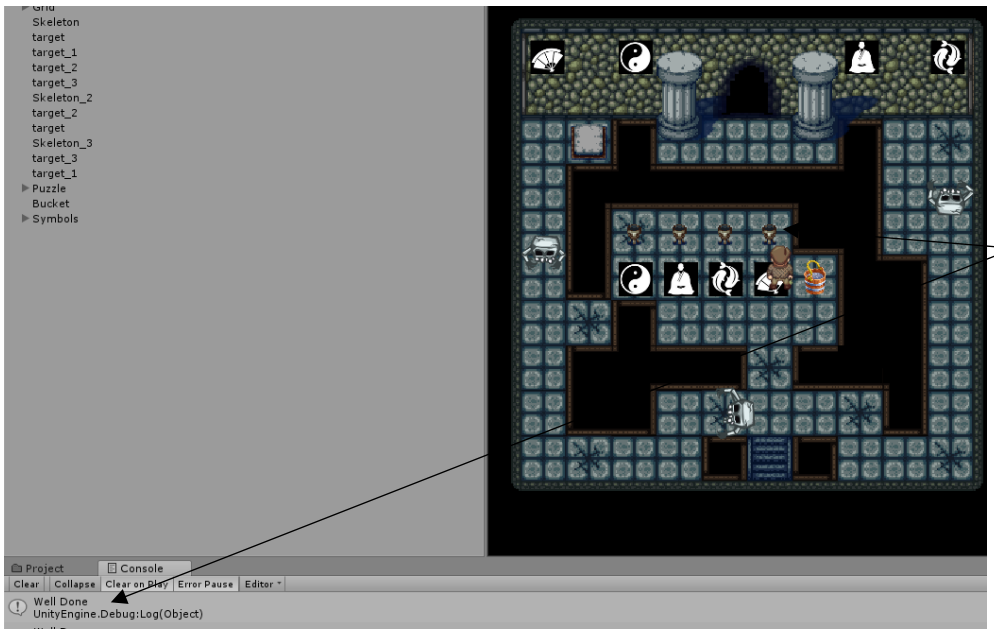
I then added a new integer variable which will be used to determine how many torches are turned off. In every if statement it now adds “1” to the “Puzzle” integer,

```
}
void Update()
{
    if (puzzle == 4)
    {
        var puzzleOrder  = setSymbol.SequenceEqual(Symbol);
        Debug.Log(puzzleOrder);
        if (puzzleOrder == true)
        {
            Debug.Log("Well Done");
        }
        else
        {
            puzzle = 0;
            Fan.SetActive(true);
            Ying.SetActive(true);
            Monk.SetActive(true);
            Fish.SetActive(true);
            Symbol.Clear();
        }
    }
}
```

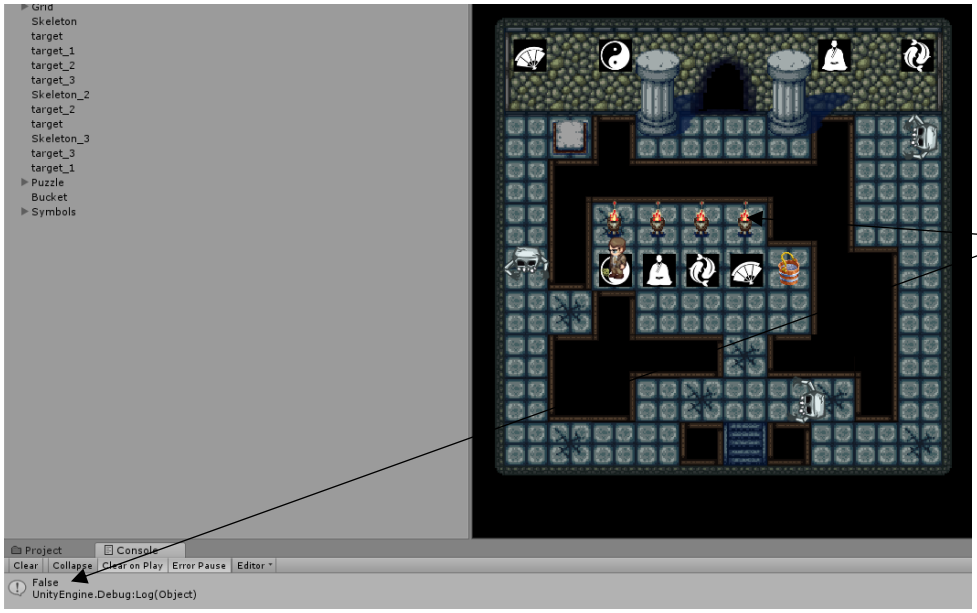
I then added some code in the Update function which will be constantly checking whether the “puzzle” variable equals 4. When it equals 4, it means that the player has turned off all the torches and so requires the two lists to be checked.

The “puzzleOrder” is a variable that will be true if the “setSymbol” lists equals the “Symbol” list and if the “puzzleOrder” equals “true” thenit prints “well done”.

However, if the “puzzleOrder” doesn’t equal “true” then else statement executes and resets the “puzzle” integer to “0” and clears the “Symbol” list, this is so that the player can try again. It also turns all the torches back on with the SetActive routine.



As shown, when the player gets the symbol order correct, the torches stay turned off and the console outputs “well done”.

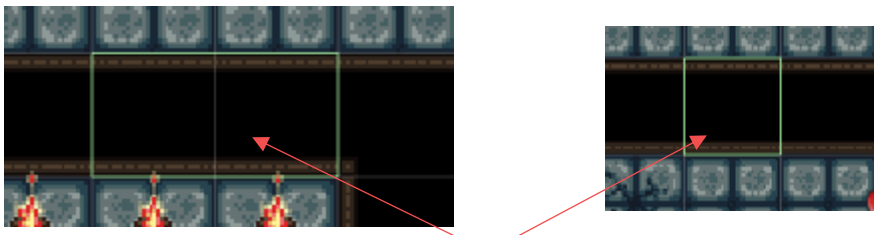


As shown, if the player gets the symbol order incorrect, it activates all the torches again and the console outputs “False”. (I added a Debug.Log(“False”) in the else statement.

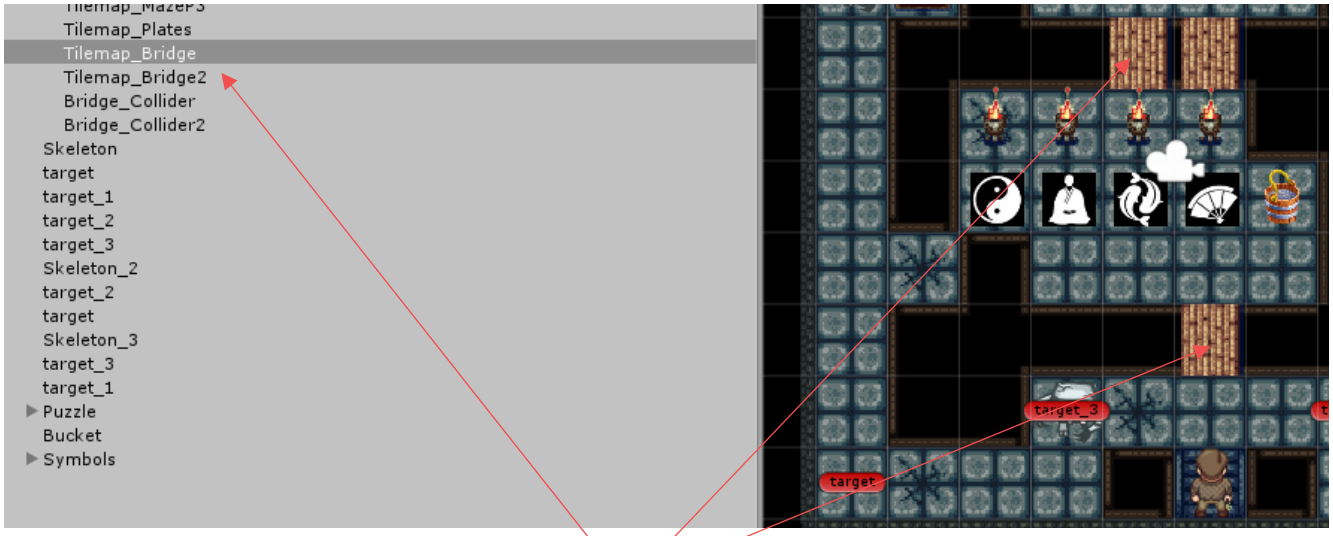
BRIDGES

Once, the symbol puzzle was working I needed to develop the GameObjects that will interact alongside the puzzle. These are a Pressure Plate which the player must collided with (step on). This will then activate a bridge and remove a 2D box collider to allow the player to walk over the bridge to the next area of the level.

Then, when the player completes the symbol puzzle, it activates a different bridge and removes its 2D box collider to allow the player to walk across to the exit.



Making the bridge 2D box colliders separate from the other “hole” colliders.

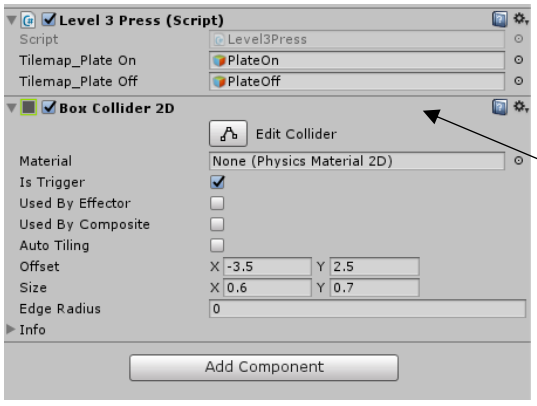


Adding in the Bridge Tilemaps as separate GameObjects

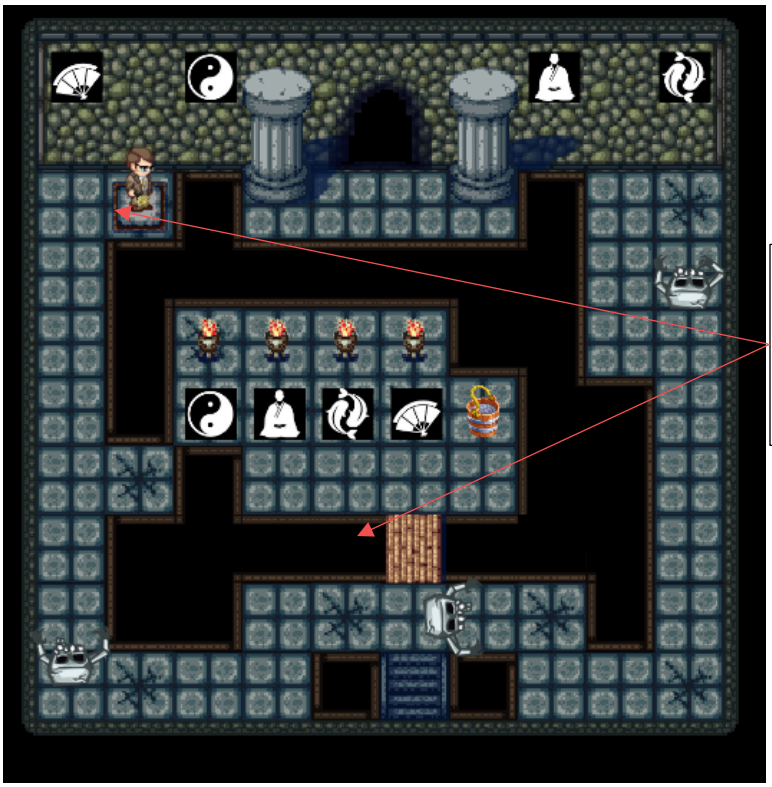
The pressure plate.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Level3Press : MonoBehaviour {
6      public GameObject Tilemap_PlateOn;
7      public GameObject Tilemap_PlateOff;
8      public GameObject Bridge;
9      public GameObject BridgeCol;
10
11
12     void OnTriggerEnter2D (Collider2D col)
13     {
14         if (col.gameObject.name.Equals("Player"))
15         {
16             Tilemap_PlateOn.SetActive(true);
17             Tilemap_PlateOff.SetActive(false);
18             Bridge.SetActive(true);
19             BridgeCol.SetActive(false);
20         }
21     }
22     void OnTriggerExit2D (Collider2D col)
23     {
24         if (col.gameObject.name.Equals("Player"))
25         {
26             Tilemap_PlateOn.SetActive(false);
27             Tilemap_PlateOff.SetActive(true);
28         }
29     }
30
31
32 }
```

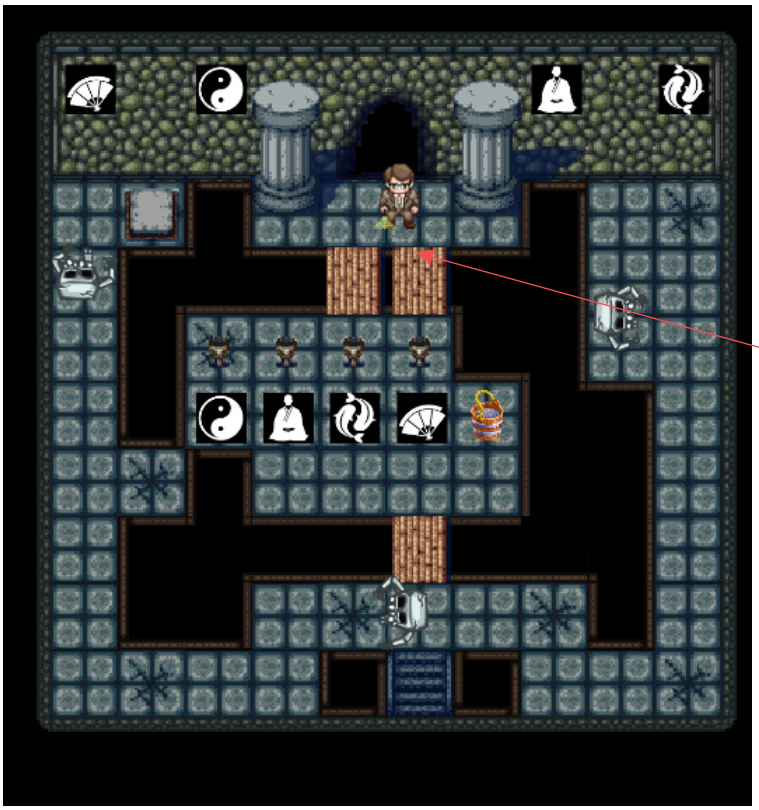
This script I've just copied from the level 2 pressure plates as it's just the same, a pressure plate that the player can push down and then pop pack up when they walk off it. All that's different is line 18 and 19 which activate the "Bridge" Tilemap and deactivates the 2D box collider that was in it's place.



Setting up the parameters for the pressure plate.



Showing that the when the Player walks on the pressure plate it does change form and activates the bridge as well as removing the 2D box collider.



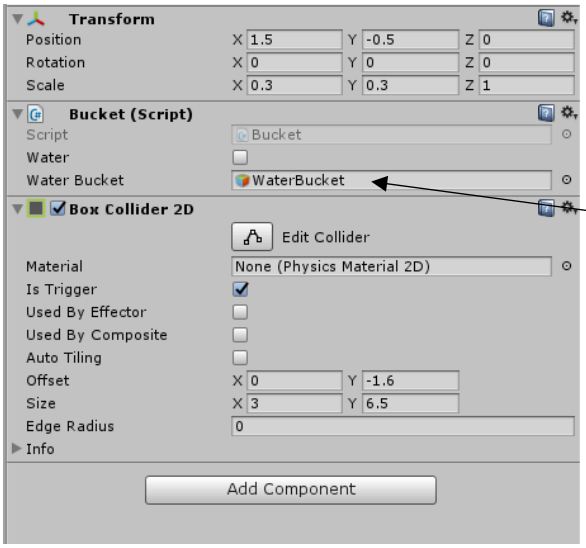
Showing that when the Player completes the puzzle, the bridge activates and the 2D box collider deactivates, allowing the Player to walk over to the exit.

WATER BUCKET

I then wanted to add a little more to the puzzle which was a water bucket GameObject. This water bucket will be placed in the level and the user must work out that they need to pick up the bucket to put out the torches. Without the bucket the torches won't go out.

Therefore, I added a water bucket GameObject and made a script that when the 2D box collider is collided with the player and the user pressed "e", the bucket GameObject is destroyed and the bool variable Water become "true". This "Water" Boolean variable will be used in the Symbol script.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class Bucket : MonoBehaviour {
6      public bool Water;
7      public GameObject WaterBucket;
8
9      void OnTriggerStay2D (Collider2D col)
10     {
11         if (col.gameObject.name.Equals("Player")&&(Input.GetKeyDown("e")))
12         {
13             Water = true;
14             Destroy(WaterBucket);
15         }
16     }
17
18 }
```



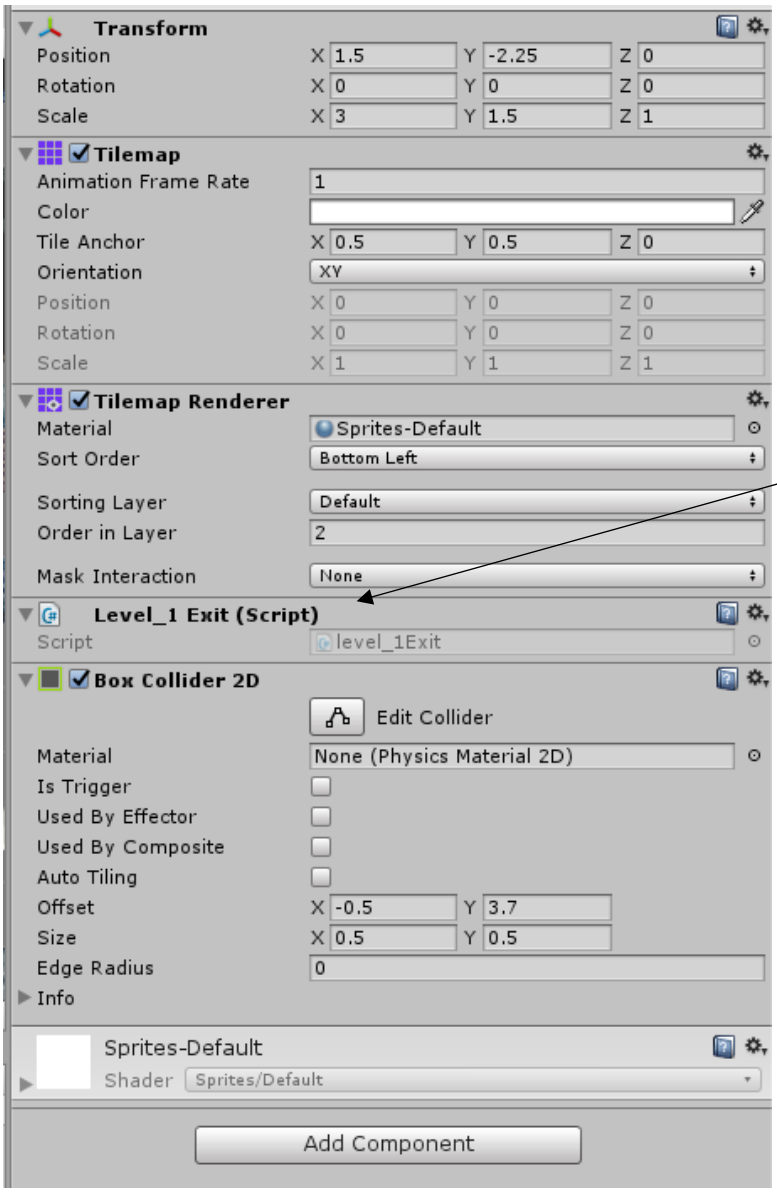
Setting up the parameters for the Water Bucket script.

```
public Bucket water;


// Use this for initialization
void Start () {
    setSymbol.Add("Fish");
    setSymbol.Add("Monk");
    setSymbol.Add("Ying");
    setSymbol.Add("Fan");
}

// Update is called once per frame

void OnTriggerStay2D (Collider2D col)
    {
        if (col.gameObject.name.Equals("floorFan")&&Input.GetKeyDown("q")&&water.Water == true)
        {
            Fan.SetActive(false);
            Symbol.Add("Fan");
            puzzle = puzzle + 1;

        }
        if (col.gameObject.name.Equals("floorYing")&&Input.GetKeyDown("q")&&water.Water == true)
        {
            Ying.SetActive(false);
            Symbol.Add("Ying");
            puzzle = puzzle + 1;
        }
        if (col.gameObject.name.Equals("floorMonk")&&Input.GetKeyDown("q")&&water.Water == true)
        {
            Monk.SetActive(false);
            Symbol.Add("Monk");
            puzzle = puzzle + 1;
        }
        if (col.gameObject.name.Equals("floorFish")&&Input.GetKeyDown("q")&&water.Water == true)
        {
            Fish.SetActive(false);
            Symbol.Add("Fish");
            puzzle = puzzle + 1;
        }
```

Slightly editing the Symbol script by adding in another condition for the IF statements which is that the Boolean variable from the Water Bucket script must be “true”, meaning the water bucket has been picked up by the player.

Showing that when the water bucket is picked up, the torches can then be put out.

EXIT



The Tilemap Exit which I made previously, I just added the pre-existing exit script to it so that when the Player collides with it, it loads the next scene.

MEETING SUCCESS CRITERIA

I am confident that for Level 3 I've met all the success criteria relevant to this section of development that I have developed so far.

- Documentation of testing – I've clearly shown evidence of testing that I've done throughout the development of this level and all the normal, boundary an extreme testing when applicable.

- Documentation of errors whilst in development – Every error that I've come across I've documented and explained the reasoning and the solution to the problem. However, I didn't come across many errors during the development of this level, this is probably due to me having already gained knowledge and experience on C# and Unity from developing the other two levels.

- The player will be able to interact fully with every Rune Raid aspect – Every aspect I've set out to develop I've and ensured that the user can use it fully by testing them myself. However, I've not yet added a few aspects yet as I've discussed previously such as the timer and the Rune.

- All algorithms will be as efficient as I can possibly make them – The most complex algorithm that I have developed for this level is the Skeleton movement algorithm that builds a pathway for a GameObject to follow by utilising targets which the algorithm finds, moves towards and then repeats for the next target when it reaches its current targets position. However, due to me finding this code from YouTube all I did was slightly modify it to adapt to my level environment and therefore, it kept the same high efficiency. It also show's it's efficiently because it creates a endless in the level loop but doesn't cause a Memory overload problem.

- All decisions made will be shown and justified – Every significant decision I've made to this development I've fully recorded and justified such as, any extra additions that I decided to add into the level, like the Water Bucket.

- Run Raids puzzles must be challenging for the player – As this is the final level I wanted this puzzle to be the hardest and I believe I achieved that. This is because not only does the player have to avoid the Skeletons because it will add extra time on their timer (I haven't developed the timer yet) but the puzzle itself I believe is relatively difficult. It may seem very simple once the player knows the pattern but just reading the symbols backwards is not always a thought that comes to mind when people are trying to solve an order puzzle and so it may take some time for players to work out.

- All the code is split up into specific functions within modules – Every script has their own functions within them that are grouped effectively to by relevance and each different aspect has their own script such as the Skeleton target finding script and the Animation script.