P11011180 – Luke O Brien

Design Document for:

Headwear Lass

Game Overview

Philosophy:

The core goal of this game was to capture what made the early 3D platformers like 'Super Mario 64' and 'Banjo Kazooie' so well received and create a game using some new techniques that weren't around back in the heyday 3D platformers.

This involved looking at what made those games great and mimicking them. The inspiration that I took from these games can be seen with my simple and colourful art style.

Focus:

One thing that all successful 3D platformers have in common is that they all have great and tight controls because of this, the focus of the game was to fine tune the controls until they just felt right.

Features:

- Linear levels as they allow more fine tuning than an open world.
- Fluid 360-degree movement.
- Multiple levels.
- Simple yet colourful art style.

Story:

The games story component is basic. The player was playing in her parent's bank when a rift in space and time opened up and sucked up all the bank's money. It is up to the player to go into the rift and recover her parent's money before they get back.

Gameplay:

The gameplay is fairly easy to get your head around. You can only move and jump so the challenge comes with you being able to master the simple control set to overcome obstacles within the levels. Along the way through the levels the player will be able to collect coins and aim for a high score. I decided against requiring the player to collect a certain amount of coins before they can beat a level because I didn't want players to feel pressured into playing the game in a way that they didn't want to.

The Game World:

The game will take place over three levels that are inside a rift. The levels will be a series of blocks floating in the middle of nowhere. Each level will introduce a new mechanic for the player to experience and overcome. This will allow the layer to slowly get better without being overwhelmed by different obstacles.

Art:

The art style that I chose was one of the earliest aspects that I had decided on. I wanted the game to be colourful and fore it to appeal to a younger audience.

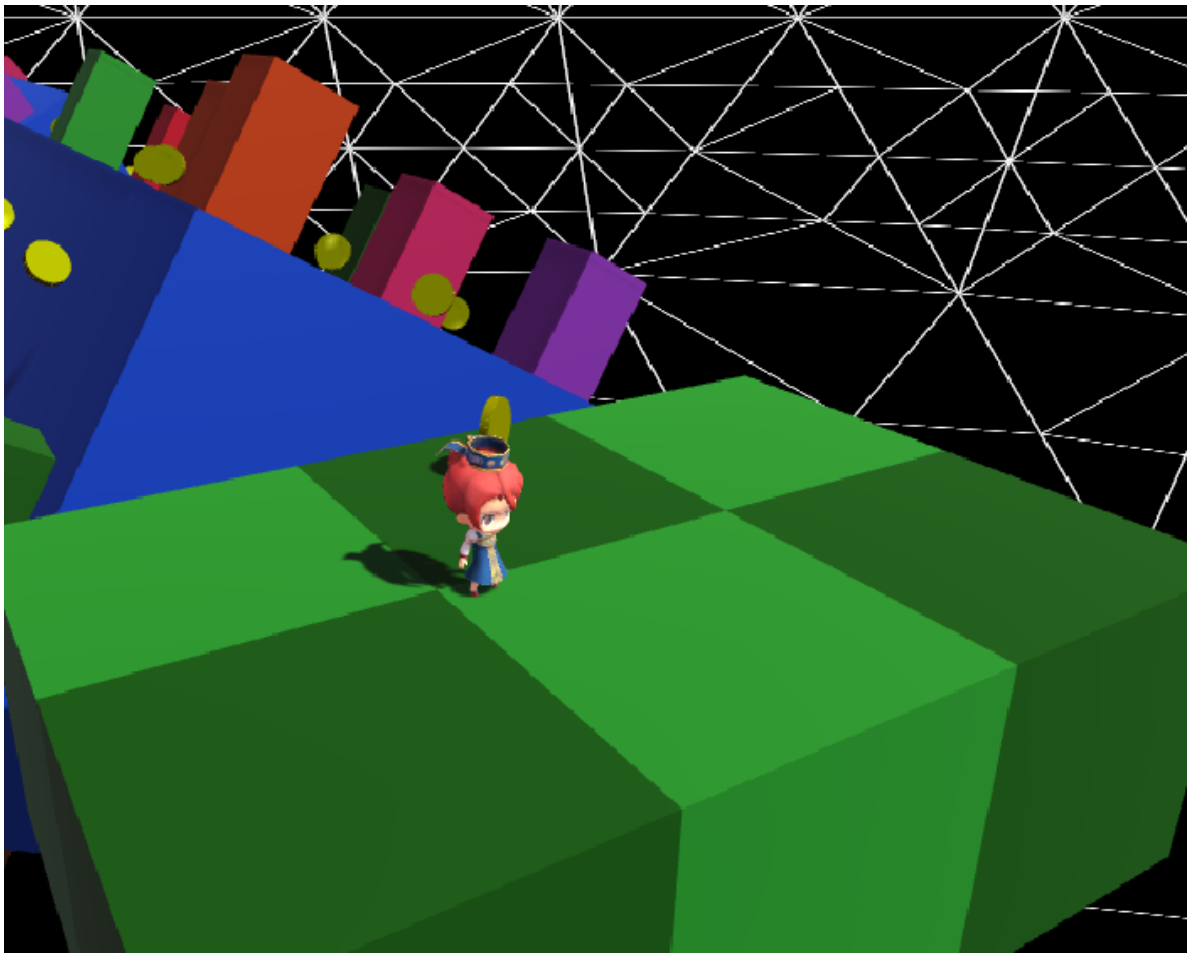

Figure 1 - Art Style

As you can see, I created the levels out of colourful blocks that contrast against the black skybox within the game. The dark background helps to make the player feel like they really were sucked into a rift and fits the story of the game.

The player model that was used in the game is a free asset on the Unity store. This came with animation for running and jumping. I had to edit some of the animations (removing the wind up from the jump animation) so that they were more in line with my players movement.

All the other art in the game was created by me. I used the basic Unity 3D shapes to create all the platforms and coins.

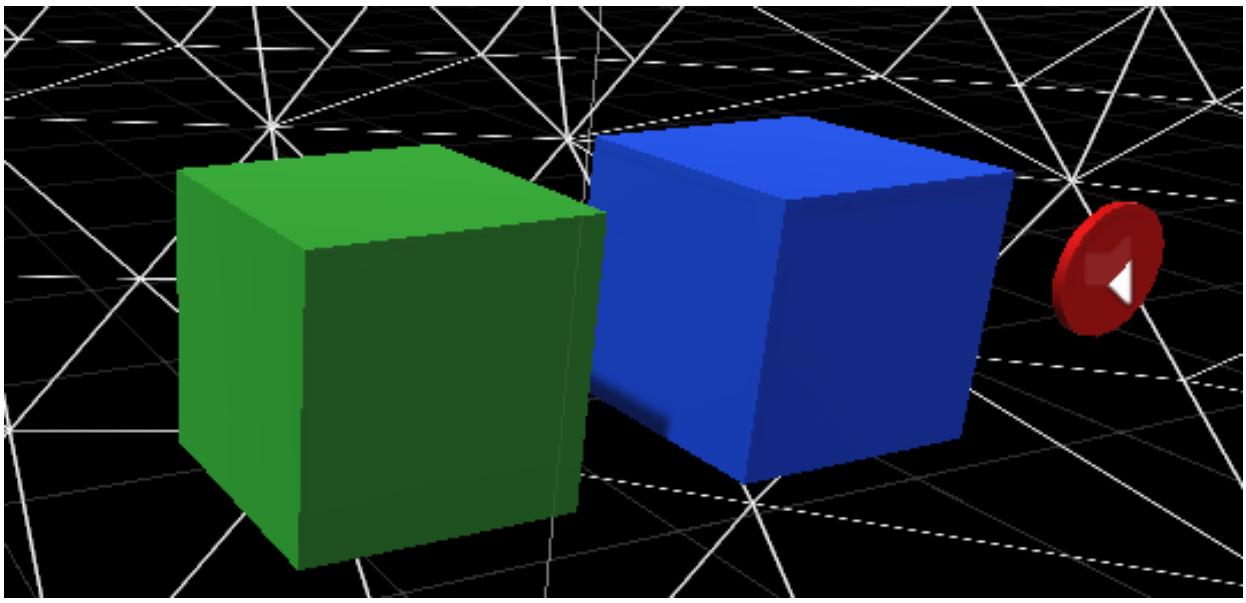

Figure 2 - 3D shapes

All the coins are just cylinders that I lowered the height of so that they looked like coins. The large rotating objects found on level two are just lots of cubes and coins as child objects to an empty game object.

Sounds:

All of the sounds and music that I used in this game are royalty free and were gotten online. Most of the sound effects and music that I used can be found on:

https://www.dl-sounds.com/

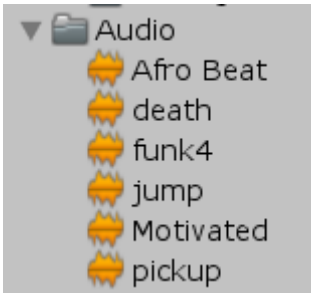

Figure 3 - Audio list

Scripts

In this section I will explain the code that's used in the game. I will go script by script and use a screenshot of code followed by the explanation.

CameraController

Start()

```
16      // Use this for initialization
17      void Start ()
18      {
19          if (!useOffsetValues)
20          {
21              offset = target.position - transform.position;
22          }
23
24          pivot.transform.position = target.transform.position;
25          pivot.transform.parent = null;
26
27          Cursor.lockState = CursorLockMode.Locked;
28      }
```

Figure 4 - CameraController Start Function

The first thing that happens is that we check the Boolean that decides whether we use a pre-set offset or just take the cameras position in relation to the player as the offset. Because we don't want the players rotation to change with the camera, we use another transform that gets set to the players position. This will stop the player from rotating. On line 27, I lock the mouse's position so that as the player is playing the game, they don't have to worry about the mouse cursor moving around.

LateUpdate()

```
30      // Update is called once per frame
31      void LateUpdate ()
32      {
33          pivot.transform.position = target.transform.position;
34          //get the x position of the mouse
35          float horizontalRotation = Input.GetAxis("Mouse X") * rotateSpeed;
36          pivot.Rotate(0, horizontalRotation, 0);
37
38          //get the y position of the mouse
39          float verticalRotation = Input.GetAxis("Mouse Y") * rotateSpeed;
40          pivot.Rotate(-verticalRotation, 0, 0);
41
42          float desiredYAngle = pivot.eulerAngles.y;
```

Figure 5 - CameraController LateUpdate function part 1

I use LateUpdate here because I found that if the camera updates weren't after others in the game, then the camera would sometimes stutter and not look smooth. Using LateUpdate allows me to wait until all of the other objects in the game have updated before the camera.

The first line of this function is setting the pivot to the players position. Lines 34 to 36 are using Unity's built in input manager to read the mouse's horizontal movement. I then use this value to rotate the pivot by an amount. Lines 38 to 40 do the same thing but for the Y axis.

```
44          //limit up/down camera rotation
45          if (pivot.rotation.eulerAngles.x > maxViewAngle && pivot.rotation.eulerAngles.x < 180f)
46          {
47              pivot.rotation = Quaternion.Euler(maxViewAngle, desiredYAngle, 0);
48          }
49          if (pivot.rotation.eulerAngles.x > 180f && pivot.rotation.eulerAngles.x < 360f - minViewAngle)
50          {
51              pivot.rotation = Quaternion.Euler(360f - minViewAngle, desiredYAngle, 0);
52          }
53
```

Figure 6 - CameraController LateUpdate Function part 2

These if statements are responsible for limiting the range that the camera can move in on the Y axis. I do this because if the camera goes too high it will eventually cause the camera view to suddenly flip which is unpleasant to look at. The lower limit was used so that the camera couldn't be moved into the ground.

```
58          Quaternion rotation = Quaternion.Euler(desiredXAngle, desiredYAngle, 0);
59          transform.position = target.position - (rotation * offset);
60
61          if(transform.position.y  < target.position.y)
62          {
63              transform.position = new Vector3(transform.position.x,
64                                               target.position.y - 0.5f, transform.position.z);
65          }
66
67          //makes the camera look at the player
68          transform.LookAt(target);
69      }
```

Figure 7- CameraController LateUpdate Function part 3

Now that we have the rotations stored as Euler angles, we need to apply them to the camera. To do this I first translated the Euler angles into a quaternion which is much better suited for this scenario. I then apply the rotation and the offset to the camera. Finally, we tell the camera to look at the player.

CoinPickup

Start()

```
13          void Start ()
14          {
15              pickup = GetComponent<AudioSource>();
16          }
```

Figure 8 - CoinPickup Start Function

All the Start function is doing in this script is initializing the sound effect that plays when the coin is picked up by the player.

Update()

```
19      void Update ()
20      {
21          transform.Rotate(90 * Time.deltaTime, 0, 0);
22
23      }
```

Figure 9 - CoinPickup Update Function

This update function just makes the coin rotate around the X axis. I use delta time so that it doesn't matter what frames per second my game is running at, the coins will always rotate at the same speed.

OnTriggerEnter(Collider other)

```
25      private void OnTriggerEnter(Collider other) //when the player touches th
26      {
27          if(other.name == "Player")
28          {
29              //play sound
30              pickup.Play();
31
32              //spawn particles
33              Instantiate(particles, transform.position, transform.rotation);
34
35              // increment coins and print out
36              FindObjectOfType<GameManager>().coinsCollected++;
37              FindObjectOfType<GameManager>().coinText.text = "Coins: "
38                      + FindObjectOfType<GameManager>().coinsCollected;
39
40              // disable coin once collected.
41              GetComponent<Collider>().enabled = false;
42              GetComponent<MeshRenderer>().enabled = false;
43          }
44      }
```

Figure 10 - CoinPickup OnTriggerEnter Function

This function is called whenever an object collides with a coin. The first thing we check is if the object that collided is the player as it's the only collision we care about. It will play the pickup sound and then spawn a particle effect at its location.

Lines 35 to 38 are responsible for updating the coins collected variable inside the Game Manager object, it then updates the text on screen that shows the coin count. Using 'FindObjectOfType' allows us to easily access any public variables or functions inside other objects.

Lastly the code is turning off the collider and the mesh renderer for the coin that's picked up. I don't destroy the coin here because the pickup sound won't have time to play. As my levels are short, its okay to just let unity clean up the coins in memory when the scene changes.

CoinSingleton()

Awake()

```
 9      static CoinSingleton instance;  //creat
10
11      public int coins = 0;
12      public int deaths = 0;
13      private void Awake()
14      {
15          if(instance != null)    //making su
16          {
17              Destroy(gameObject);
18          }
19          else
20          {
21              instance = this;
22              DontDestroyOnLoad(gameObject);
23          }
24      }
25
```

Figure 11 - CoinSingleton Awake function

As I want to use this singleton to store variables that persist between scenes, I first create a static instance of it. In the Awake function, I am telling the compiler to destroy any duplicates of this singleton and also telling Unity's garbage collection to not destroy it when moving to a new scene.

AddCoins() and ResetCoins()

```
26      public void AddCoins()  //adding the coins collected in the cu
27      {
28          coins = FindObjectOfType<GameManager>().coinsCollected;
29      }
30
31      //resets the coins back to 0 in the event that the player clic
32      public void ResetCoins()
33      {
34          coins = 0;
35      }
```

Figure 12 - CoinSingleton AddCoins and ResetCoins functions

The AddCoins function is called at the end of each level and just adds the coins collected on the level to the total that's stored here.

I call ResetCoins at the end of the game if the user hits the 'Play Again' button.

GameManager

Start()

```
17      void Start ()
18      {
19
20          coinsCollected = FindObjectOfType<CoinSingleton>().coins;
21          music.Play();
22          Cursor.lockState = CursorLockMode.Locked;
23          Cursor.visible = false;
24          coinText.text = "Coins: " + coinsCollected;
25      }
```

Figure 13 - GameManager Start Function

The game manager is created at the start of every level. The first thing it does is retrieve the amount of coins collected on previous levels. It then starts playing the background music. Next, it locks the mouse cursor and makes it invisible before finally printing the coins to the screen.

LevelComplete() and LevelEnd()

```
35      public void LevelComplete()
36      {
37          coinsCollected += 10;
38          if (!isComplete)
39          {
40              StartCoroutine("LevelEnd");
41          }
42
43      }
44
45      //this is called when the player touches the big coin at the end of a level
46      // it starts a wait function for 2 seconds on a different thread than every
47      public IEnumerator LevelEnd()
48      {
49          isComplete = true;
50
51          yield return new WaitForSeconds(2);
52
53          isComplete = false;
54
55          SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
56      }
```

Figure 14 - GameManager LevelComplete and LevelEnd functions

The LevelComplete function gets called whenever the big coin at the end of a level gets picked up. It first increments the amount of coins collected by 10. It then calls the ‘LevelEnd’ coroutine. It is considered a good programming practice to call coroutines in this way instead of calling them directly.

A coroutine runs on a different thread than the rest of the game. This allows us to call wait methods without everything else in the game needing to wait. After this function waits for 2 seconds, it then loads the next scene in the build order.

LevelEndCoin and RespawnCoin

The code for these scripts is the same as the basic coin pickup script the only difference is that they both have an added call when the player collides with them.

When the player collides with the LevelEndCoin, it calls the LevelComplete function within the GameManager.

When the player collides with the RespawnCoin, it calls the SetRespawn function that's in the PlayerController.

I understand that using 3 scripts for the coins is poor programming, but I didn't have time to rectify it. If my game gets selected for GamesFleadh, I will combine the 3 coin scripts into 1.

MagicBlocks

There are two scripts here that both perform the same function but have inverted Boolean values. What these scripts do is to turn on/off the blocks on a 2 second loop.

```
20      void Update ()
21      {
22          if (!magicBlocksActive)
23          {
24              StartCoroutine("MagicBlocks");
25          }
26      }
27
28      public IEnumerator MagicBlocks()
29      {
30          magicBlocksActive = true;
31
32          //disable the block
33          GetComponent<Collider>().enabled = false;
34          GetComponent<MeshRenderer>().enabled = false;
35
36          yield return new WaitForSeconds(2); //wait for 2 seconds
37
38          //enable the block
39          GetComponent<Collider>().enabled = true;
40          GetComponent<MeshRenderer>().enabled = true;
41
42          yield return new WaitForSeconds(2); //wait for 2 seconds
43
44          magicBlocksActive = false;
45
```

Figure 15 - MagicBlocks code

In the Update function, all that happens is that we call a coroutine. Inside the coroutine we first disable the block by turning off the collider and mesh renderer. We then wait for 2 seconds on a different thread before enabling the block again. Then we wait for another 2 seconds.

MainMenu

Start() and Update()

```
10      public void Start()
11      {
12          //make the mouse visible
13          Cursor.lockState = CursorLockMode.None;
14          Cursor.visible = true;
15
16          menuMusic.Play();
17
18      }
19      public void Update()
20      {
21          if (Input.GetButtonDown("Jump"))
22          {
23              PlayGame();
24          }
25      }
```

Figure 16 - MainMenu Start and Update Functions

In the Start function we first give the user control of the mouse before playing the menu music. In the Update function we are checking to see if the spacebar (or A on a controller) is pressed. If it is then we call the PlayGame function.

```
26      public void PlayGame()  //when the play button is pressed
27      {
28          SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
29      }
30
31      public void QuitGame()  // when the quit button is pressed
32      {
33          Debug.Log("Quit");
34          Application.Quit();
35      }
36
37      public void PlayAgain() //when play again is pressed
38      {
39          FindObjectOfType<CoinSingleton>().ResetCoins();
40          SceneManager.LoadScene("LevelOne");
41      }
42  }
```

Figure 17 - MainMenu Button functions

The PlayGame function is called whenever the player clicks on the play button. What it does is progress the game into the next scene in the build order.

The next function will first print out ‘Quit’ to the debug log in Unity before terminating the application. I tell it to print out to the debug because it is the only way to test while inside the Unity editor.

Finally, the PlayAgain function is called if the player hits the ‘Play Again’ button after they have beaten the game. It first calls the ResetCoins function inside the coin singleton so that the next playthrough starts with 0 coins and then it loads level one.

MovingPlatform

OnTriggerEnter() and OnTriggerExit()

```
17      private void OnTriggerEnter(Collider other) /
18      {
19          if(other.gameObject == Player)
20          {
21              Player.transform.parent = transform;
22          }
23      }
24
25      private void OnTriggerExit(Collider other) //
26      {
27          if (other.gameObject == Player)
28          {
29              Player.transform.parent = null;
30          }
31      }
```

Figure 18 - MovingPlatform OnTriggerEnter & OnTriggerExit functions

On trigger enter is called when an object collides with the moving platform. It first checks to see if the object is the player. If it is then we make the player a child of the moving platform so that it is also affected by the transform. This stops the player from just slipping off the platform when it moves.

On trigger exit is then called when the player jumps off the platform. This function just removes the player as a child of the moving platform.

Start()

```
33      private void Start()
34      {
35          startingPosition = transform.position;
36      }
```

Figure 19 - MovingPlatform Start function

Here we are just saying that when the moving platform is spawned, store its starting position.

Update()

```
38      private void Update()   //platforms movement. general and allows different directions and amounts to m
39      {
40          // move in the amount specified in the editor
41          if(reverseMovement == false)
42          {
43              transform.position = new Vector3(transform.position.x + movementAmountX * Time.deltaTime,
44                  transform.position.y + movementAmountY * Time.deltaTime,
45                  transform.position.z + movementAmountZ * Time.deltaTime);
46          }
47          else if (reverseMovement == true) // moves back to the starting position
48          {
49              transform.position = new Vector3(transform.position.x - movementAmountX * Time.deltaTime,
50                  transform.position.y - movementAmountY * Time.deltaTime,
51                  transform.position.z - movementAmountZ * Time.deltaTime);
52          }
```

Figure 20 - MovingPlatform Update function part 1

I programmed this script to take in the directions and amount(speed) that you want to move in through the editor. I then use a Boolean to make sure that the platform moves in the correct direction as it needs to be reversed at some point.

```
54          //decide when to reverse the movement of the platform
55          if (transform.position.z > (startingPosition.z + movementTotal) ||
56              transform.position.y > (startingPosition.y + movementTotal))
57          {
58              reverseMovement = true;
59          }
60          if (transform.position.z < (startingPosition.z) ||
61              transform.position.y < (startingPosition.y))
62          {
63              reverseMovement = false;
64          }
65      }
66
```

Figure 21 - MovingPlatform Update Function part 2

I need a method of deciding when its time for the platform to reverse direction. To do this I use an if statement that says if the platforms position is higher than its starting position + the total amount its supposed to move, then we reverse its direction.

Now that it is reversed, I also need to tell it to reverse again when it reaches the starting position again.

RotatingObstacle

Update()

```
16      void Update ()
17      {
18          transform.Rotate(rotationX * Time.deltaTime, 0, rotationZ * Time.deltaTime);
19      }
```

Figure 22 - RotatingObstacle Update Function

The only thing in this script is the Update function. All it does is rotate whatever game object that its attached to by an amount specified inside the Unity editor.

PlayerController

Start()

```
26      void Start ()
27      {
28          // when the player is spawned, this line will get t
29          controller = GetComponent<CharacterController>();
30          respawnPoint = transform.position;
31      }
```

Figure 23- PlayerController Start function

The first thing that the start function does is fetch the Character Controller component from the play. It then sets the respawn point of the player to its current location.

Update()

```
34  void Update ()
35  {
36
37      // setting the players movement
38      float yStore = moveDirection.y;
39      moveDirection = (transform.forward * Input.GetAxisRaw("Vertical"))
40          + (transform.right * Input.GetAxisRaw("Horizontal"));
41
42      moveDirection = moveDirection.normalized * moveSpeed;
43      moveDirection.y = yStore;
44
45      //check if the player is on the ground
46      if (controller.isGrounded)
47      {
48          moveDirection.y = -0.3f;
49          //if 'spacebar' is pressed, then the player jumps
50          if (Input.GetButtonDown("Jump"))
51          {
52              jump.Play();
53              moveDirection.y = jumpForce;
54          }
55      }
56      moveDirection.y = moveDirection.y + (Physics.gravity.y * gravityScale * Time.deltaTime);
```

Figure 24 - PlayerController Update function part 1

Lines 38 to 43 are responsible for the players movement. First, we store the movement on the Y axis because we don't want to affect it every update. We then get the movement direction before normalizing it and multiplying it by the players move speed. Then it sets the Y movement to the value we just stored.

Lines 45 to 55 is where we deal with the players jump. First, we check to see if the player is grounded. This stops the player from being able to jump while in mid-air. Then we check to see if the user has pressed the jump button. If they have then we play the jump sound and apply the jump force to the Y move direction.

The last line of code in the image is applying gravity to the player.

```
59      controller.Move(moveDirection * Time.deltaTime);
60
61      //basing the players movement on the cameras rotation
62      if(Input.GetAxisRaw("Horizontal") != 0 || Input.GetAxisRaw("Vertical") != 0)
63      {
64          transform.rotation = Quaternion.Euler(0f, pivot.rotation.eulerAngles.y, 0f);
65
66          //stopping the player model from snapping to a rotation using Slerp
67          Quaternion newPlayerRotation = Quaternion.LookRotation(new Vector3(moveDirection.x,
68                                                                  0f, moveDirection.z));
69          playerModel.transform.rotation = Quaternion.Slerp(playerModel.transform.rotation,
70                                          newPlayerRotation, cameraRotateSpeed * Time.deltaTime);
71      }
```

Figure 25 - PlayerController Update function part 2

The first line in the image above is using the Character Controllers Move function to move the player. The if statement is required so that the players movement takes the cameras rotation into account. We do this by setting the players rotation to the Y rotation of the

pivot. We want the player model to move realistically to the new rotation instead of just snapping to it. To do this we use Slerp to fluidly rotate to the new players rotation.

```
73          //respawning
74          if(transform.position.y < -20)
75          {
76              if(!isRespawning)
77              {
78                  StartCoroutine("RespawnTime");
79              }
80
81          }
82
83          //values for animation choosing
84          anim.SetBool("isGrounded", controller.isGrounded);
85          anim.SetFloat("speed", (Mathf.Abs(Input.GetAxis("Vertical"))
86                                  + Mathf.Abs(Input.GetAxis("Horizontal"))));
```

Figure 26 - PlayerController Update function part 3

The if statement is calling the respawn function whenever the players Y position is less than -20 (fallen off level). Lines 84 and 85 are just telling Unity when to use the players animations.

RespawnTime()

```
91      public IEnumerator RespawnTime()
92      {
93          isRespawning = true;
94          death.Play(); //play death sound
95
96          //this block will disable the players child player model and renderer
97          Component[] a = GetComponentsInChildren(typeof(Renderer));
98          foreach (Component b in a)
99          {
100             Renderer c = (Renderer)b;
101             c.enabled = false;
102         }
103
104         yield return new WaitForSeconds(2); //wait for 2 seconds
105
106         isRespawning = false;
107         transform.position = respawnPoint; // move the player to the respawn point
108
109         //re-enable the players renderer and collision
110         foreach (Component b in a)
111         {
112             Renderer c = (Renderer)b;
113             c.enabled = true;
114         }
115         GetComponent<MeshRenderer>().enabled = false;
116         FindObjectOfType<CoinSingleton>().deaths++; //increment death counter
117     }
```

Figure 27 - PlayerController RespawnTime function

First it plays the death sound. Lines 97 to 102 are responsible for disabling the player. Because the players model is in multiple parts and is only a child of the player game object, I needed a way to disable each child of the player. To solve this, I looped though all the

players children and disabled the renderer and collider for each of them. We then wait for 2 seconds before teleporting the player back to the respawn point. Now we go through the loop again and re-enable everything. Because the player is just a capsule, we need to disable the mesh renderer for the capsule. Finally, we increment the amount of deaths.

SetRespawn()

```
119     public void SetRespawn(Transform newRespawn) //set
120     {
121         respawnPoint = newRespawn.transform.position;
122     }
```

Figure 28 - PlayerController SetRespawn function

This function is called whenever the player collides with one of the red respawn coins. All it does is set the players new respawn point to the transform of the coin that was collided with.

IsGrounded()

```
128     private bool IsGrounded()
129     {
130         if (controller.isGrounded)
131         {
132             return true;
133         }
134
135         //get center of player - height/2 to get the bottom of the player
136         Vector3 bottom = controller.transform.position - new Vector3(0, controller.height / 2, 0);
137
138         RaycastHit hit; //create raycast
139
140         // if the player is within .2 of the ground then it is considered grounded
141         if (Physics.Raycast(bottom, new Vector3(0, -1, 0), out hit, 0.2f))
142         {
143             controller.Move(new Vector3(0, -hit.distance, 0));
144             return true;
145         }
146         return false;
147     }
```

Figure 29 - PlayerController IsGrounded function

This function uses raycasts to determine if the player is grounded. This is an attempt to fix an issue in Unity when running down a slope the player will lift off of the ground thus preventing the ability to jump.

First, we check the Character Controllers isGrounded function because we don't need to do anything if it returns true. After this we get the center of the bottom of the player. The next if statement then says that if we shoot out a raycast straight down and it hits the ground within 0.2f, then we move the player down by that amount. Otherwise we just return false.

I had issues getting this to work with both the moving platforms and the rotating objects on level two. Because of this, I chose not to use it as I didn't have time to refine it.

ScoreBoard

Start()

```
// Use this for initialization
void Start ()
{

    //using PlayerPrefs so that the stats are kept across launches
    highScore = PlayerPrefs.GetInt("highScore");
    totalCoins = PlayerPrefs.GetInt("totalCoins");
    totalDeaths = PlayerPrefs.GetInt("totalDeaths");

    //retrieve deaths and coins from the singleton
    coins = FindObjectOfType<CoinSingleton>().coins;
    deaths = FindObjectOfType<CoinSingleton>().deaths;

    totalDeaths = totalDeaths + deaths;
    SetDeaths(totalDeaths);

    totalCoins = totalCoins + coins;
    SetTotal(totalCoins);

    if(coins > highScore)
    {
        SetHighScore(coins);
        highScore = coins;
    }
    else
    {
        SetHighScore(highScore);
    }

}
```

Figure 30 - ScoreBoard Start Function

Unity has a useful component called PlayerPrefs. It can store basic data types that will persist forever even if you close the game. This was a perfect fit for storing a scoreboard without needing to worry about file I/O.

So, the first thing I do in start is to retrieve the high score, total coins and total deaths from PlayerPrefs. Once I have these, I now need to get the current playthroughs coins and deaths from the singleton.

If the current playthroughs collected coins is higher than the highscore, I set it as the high score in PlayerPrefs

Update()

```
// printing out the stats
private void Update()
{
    TMscore.text = "High Score: " + highScore.ToString();
    TMcoins.text = "Coins Collected: " + coins.ToString();
    TMtotal.text = "Total Coins Collected: " + totalCoins.ToString();
    TMdeath.text = "Total Deaths: " + totalDeaths.ToString();
}
```

Figure 31 - ScoreBoard Update function

In the update function, all we are doing is printing out values to a canvas in the scene.

Setters for PlayerPrefs

```
64      // storing the total coins in the PlayerPrefs
65      public void SetTotal(int value)
66      {
67          PlayerPrefs.SetInt("totalCoins", value);
68      }
69
70      // storing the high score in the PlayerPrefs
71      public void SetHighScore(int value)
72      {
73          PlayerPrefs.SetInt("highScore", value);
74      }
75
76      // storing the total deaths in the PlayerPrefs
77      public void SetDeaths(int value)
78      {
79          PlayerPrefs.SetInt("totalDeaths", value);
80      }
```

Figure 32 - ScoreBoard Setter functions

These functions are pretty self-explanatory and are just setting values in the PlayerPrefs.

Planned Improvements

While working on this game, there were many features and improvements that I didn't have time to add in. I also learned a lot while creating the game that I wish I had known from the beginning.

Prefabs and Nesting Objects:

It wasn't until a few days before the deadline that I realised how useful prefabs were. I was able to create large obstacles by creating an empty game object and using lots of child objects to combine into a larger more intricate platforming challenge. I was then able to save it as a prefab and just drag it into levels to make the level building easier.

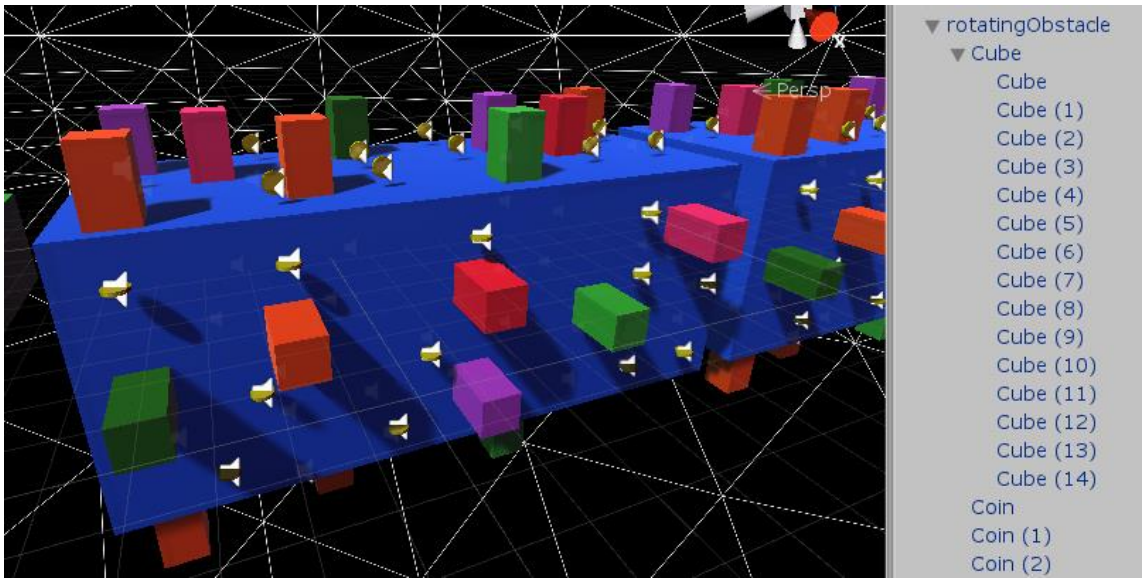

Figure 33 - Prefab and nested objects

Ray casting:

I didn't have time to implement my ray casting function properly as it was only a problem that I noticed in the last few days before the upload date. The function almost works and with a couple of hours of tweaking and testing I'm sure that I will come up to a solution to getting it feeling right.

Scoreboard Improvements:

While I am happy with my scoreboard, there are improvements to be made. At the moment, if you do not complete a playthrough then the scoreboard will not be updated. This is an easy fix and shouldn't take long.

Pause Screen:

Currently my game lacks any means to pause or exit the game mid playthrough. I plan to add in a pause screen

Local Multiplayer:

I would like to add in local multiplayer. I don't know what form it will take yet, maybe a 2 player race or a cooperative effort to collect all of the coins.