Virtual Ring Rally: Virtual Reality Flying Game

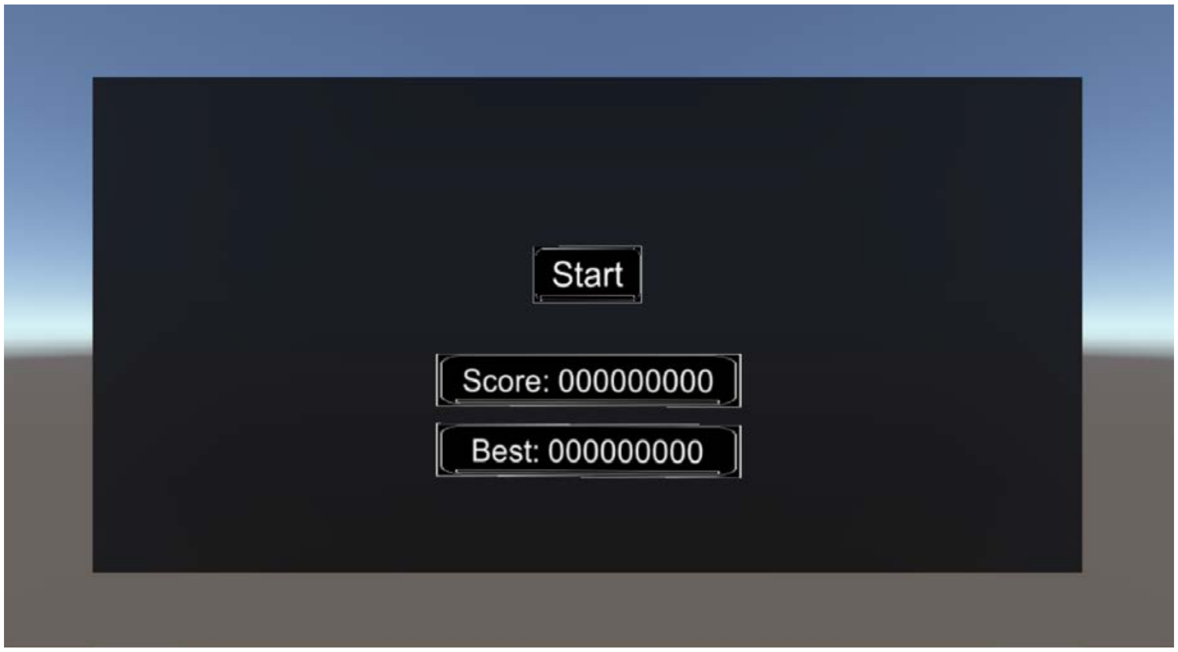
Purpose:

The purpose of this project is to create an entertaining game around the unique experiences and limitations of virtual reality. The game was created for the Samsung Gear VR headset and controller. The object of the game is to maneuver through a series of rings. Points are increased for each ring the player hits. Bonus points and increased speed are given to the player when they go without missing rings. The game continues until the player misses a total of ten rings and then the player can see how their score compares to the high score. The game was created in Unity with the GoogleVR plugin to support the VR camera and Oculus Integration plugin to support the Gear VR controller.

Project Repository:	https://bitbucket.org/ck1011/vr-project/src/master/
Gameplay Video:	https://www.youtube.com/watch?v=phs74q25OWg&feature=youtu.be

Graphics Overview:

Main Menu GUI:



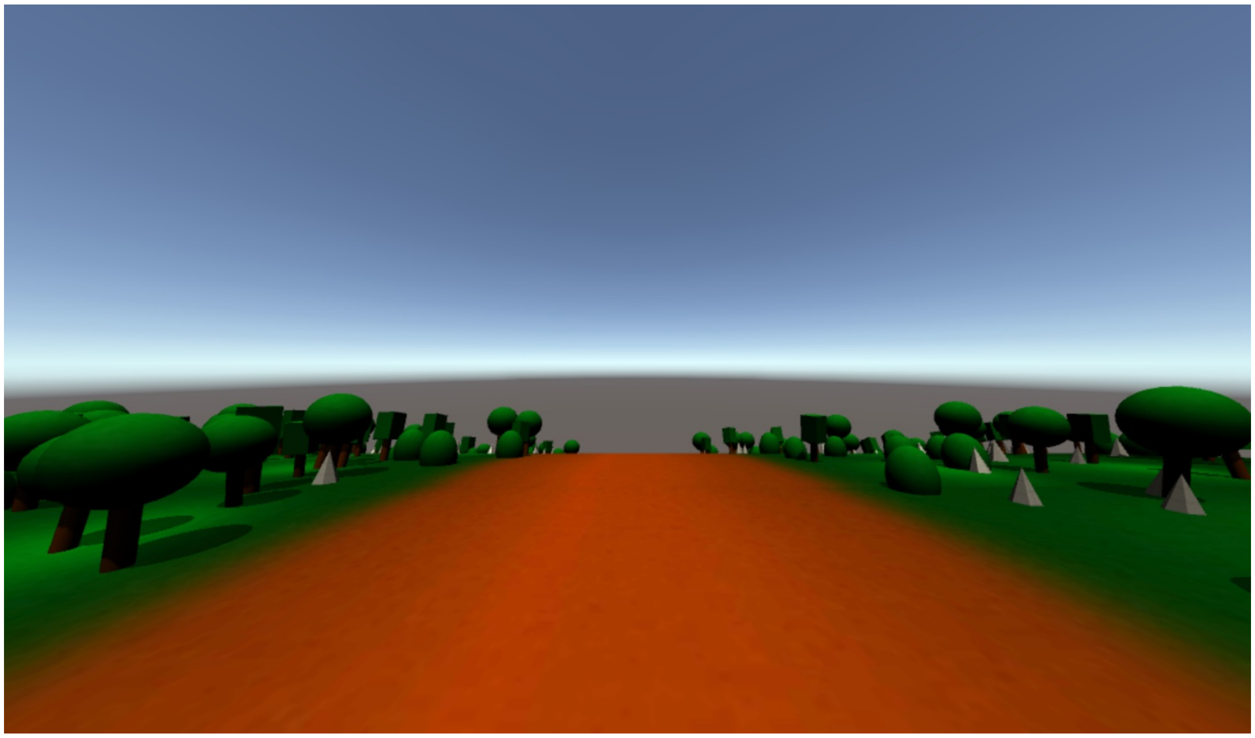
The main menu is very minimal, featuring a single button and two score displays. The style is an attempt to emulate early black and white arcade graphics.

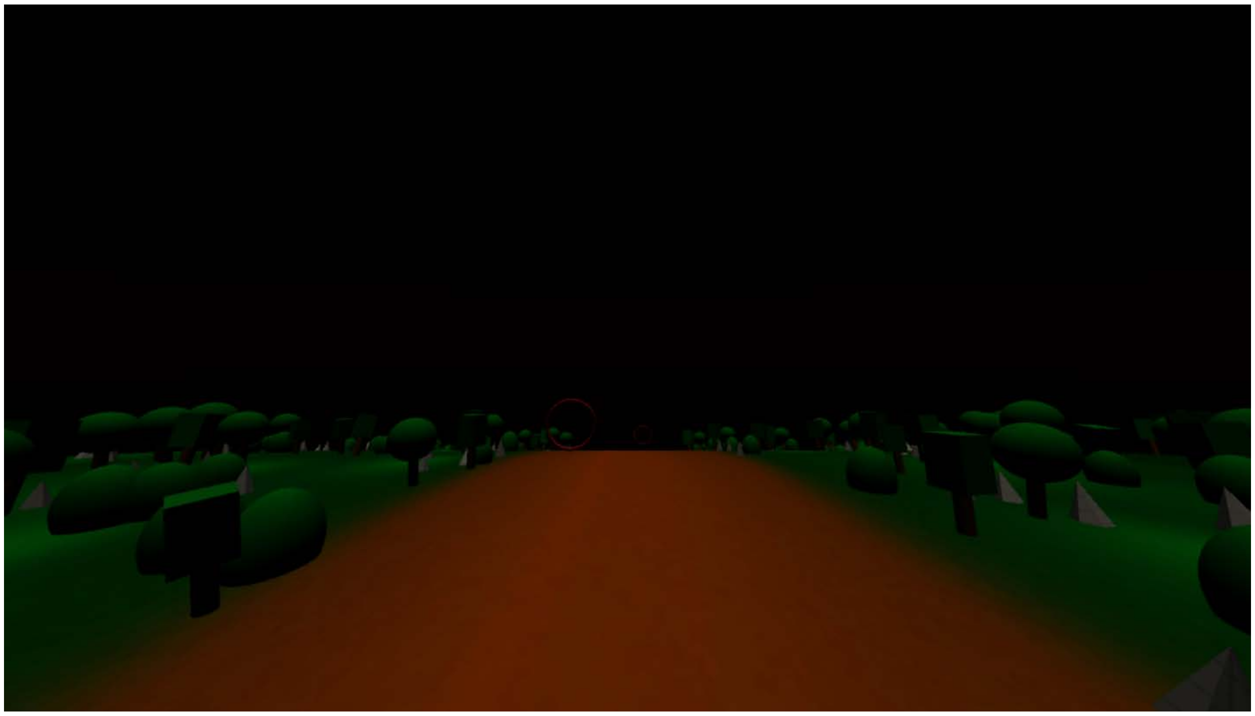
Start: When the player looks at this button for five seconds the game begins. While looking at the button the player receives visual feedback of the timer's progress.

Score: The score of the last playthrough.

Best: The highest score during any playthrough.

Gameplay Graphics:

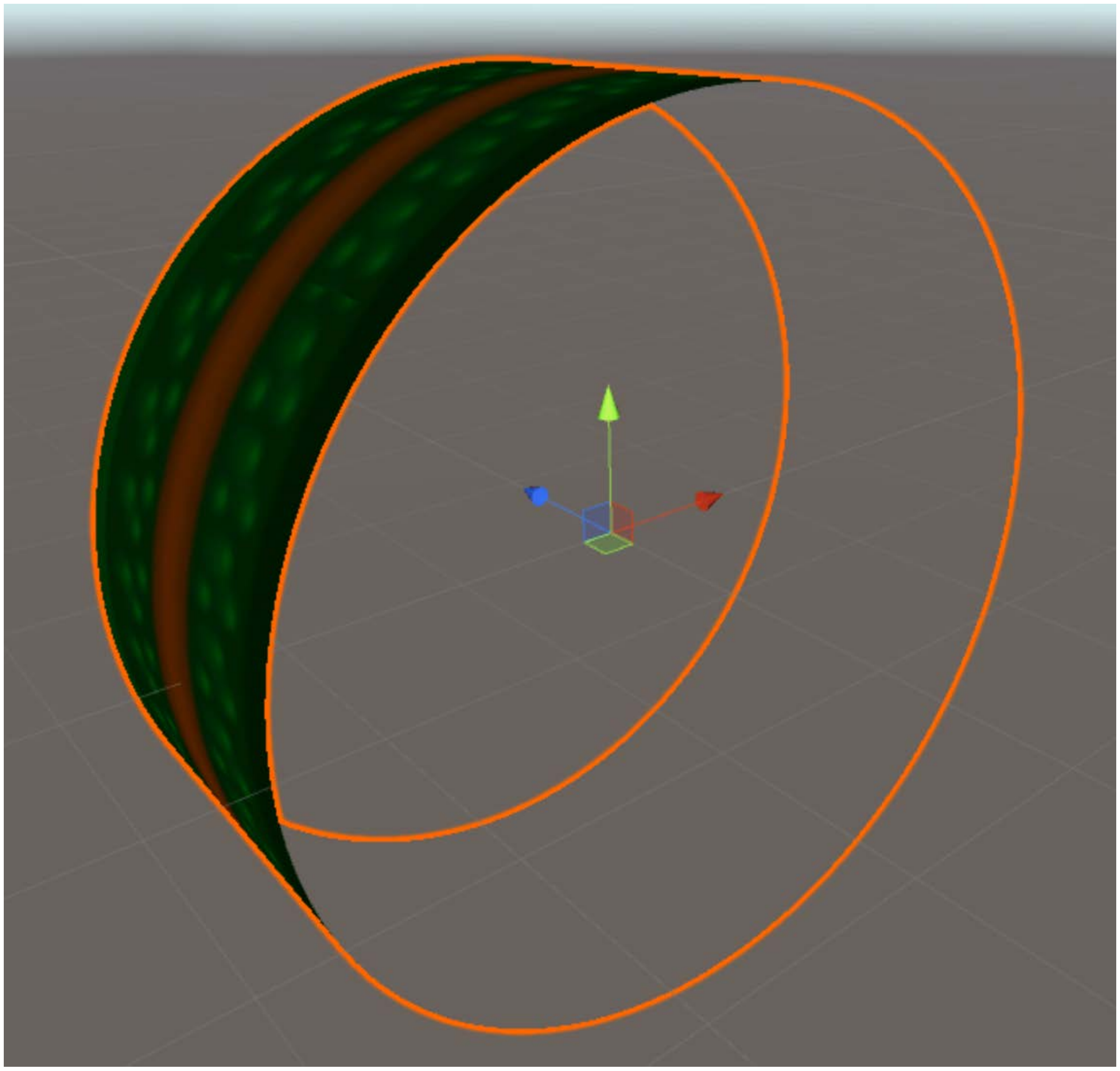




The goal of the graphics was to have the player moving towards a very close horizon, as if on a tiny version of the earth. An important reason for this style is that it limits the distance the player can see and reduces the number of elements in the scene. The limitations of smartphone hardware and the demanding nature of virtual reality requires developers to find creative ways to balance graphics and playability.

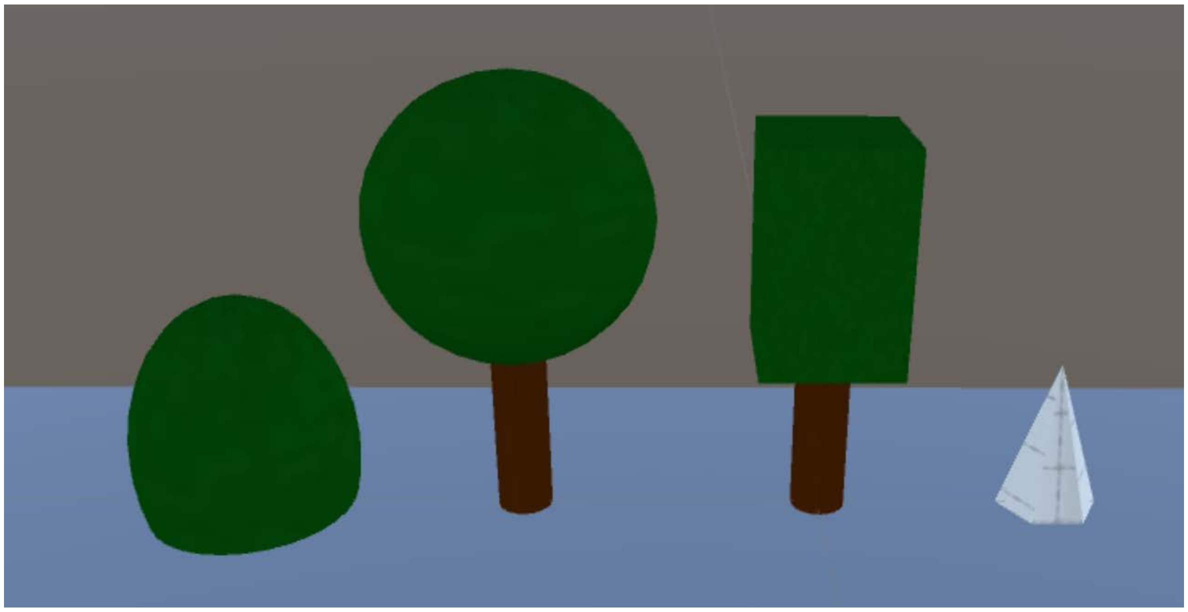
There are three main components of the gameplay graphics.

1. Ground



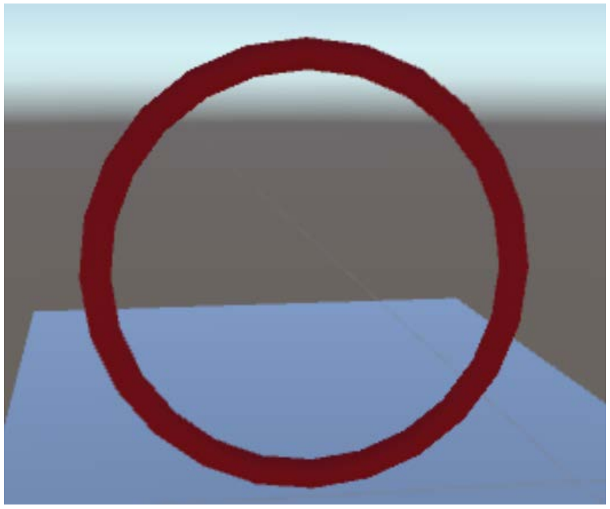
The ground is made up of a single ring with 200 sides. The player sits atop the ring as it rotates beneath them to simulate motion. As the ring rotates the lighting changes to simulate a day/night cycle.

2. Environment



The environment is made up of a combination of four objects. From left to right they are the bush, round tree, square tree and rock. How the environment is created from these four objects is discussed in the functionality section.

3. Rings



Flying through the rings is the goal of the player so they are red to contrast with the green and brown of the environment.

Functionality:

The functionality of the game can be broken up into three broad categories based on their location in the code. These categories are WorldControl, WorldGen and Miscellaneous.

1. WorldControl:

This script translates the players inputs into the game. There are two sets of inputs, one for testing in Unity and another for testing in VR. The WorldControl script was created to handle inputs relating to forward thrust, but this was set to a constant value to make the game more challenging.

```
#if (PCMODE)
        float forward_thrust = 1;//= forward_accel * Input.GetAxis("Vertical");
        float vertical_thrust = vertical_accel * Input.GetAxis("Jump");
        float lateral_thrust = lateral_accel * Input.GetAxis("Horizontal");
#else
        float forward_thrust = 1;//= forward_accel * Input.GetAxis("Vertical");
        float vertical_thrust = 0;
        if (OVRInput.Get(OVRInput.Button.PrimaryIndexTrigger))
            vertical_thrust = vertical_accel * 1;
        float lateral_thrust = lateral_accel * Input.GetAxis("Jump5");
#endif
```

The forward, vertical and lateral controls are handled in very similar ways. Vertical speed is slightly different because it also has to account for gravity. Gravity is an important gameplay factor because the player does not have the ability to thrust downwards.

```
////////////////vertical
        if (vertical_thrust != 0)//under accel
        {
            vertical_speed += Time.deltaTime * vertical_thrust;
        }
        else//no accel
        {
            vertical_speed -= gravity;//always add gravity

            if (vertical_speed > 0)
            {
                vertical_speed -= Time.deltaTime * drag;
                if (vertical_speed < 0)//crossed 0
                    vertical_speed = 0;
            }
            else if (vertical_speed < 0)
            {
                vertical_speed += Time.deltaTime * drag;
                if (vertical_speed > 0)//crossed 0
                    vertical_speed = 0;
            }
        }

        if (vertical_speed > vertical_limit * Time.deltaTime)//enforce limits
            vertical_speed = vertical_limit * Time.deltaTime;
        else if (vertical_speed < -vertical_limit * Time.deltaTime)
```

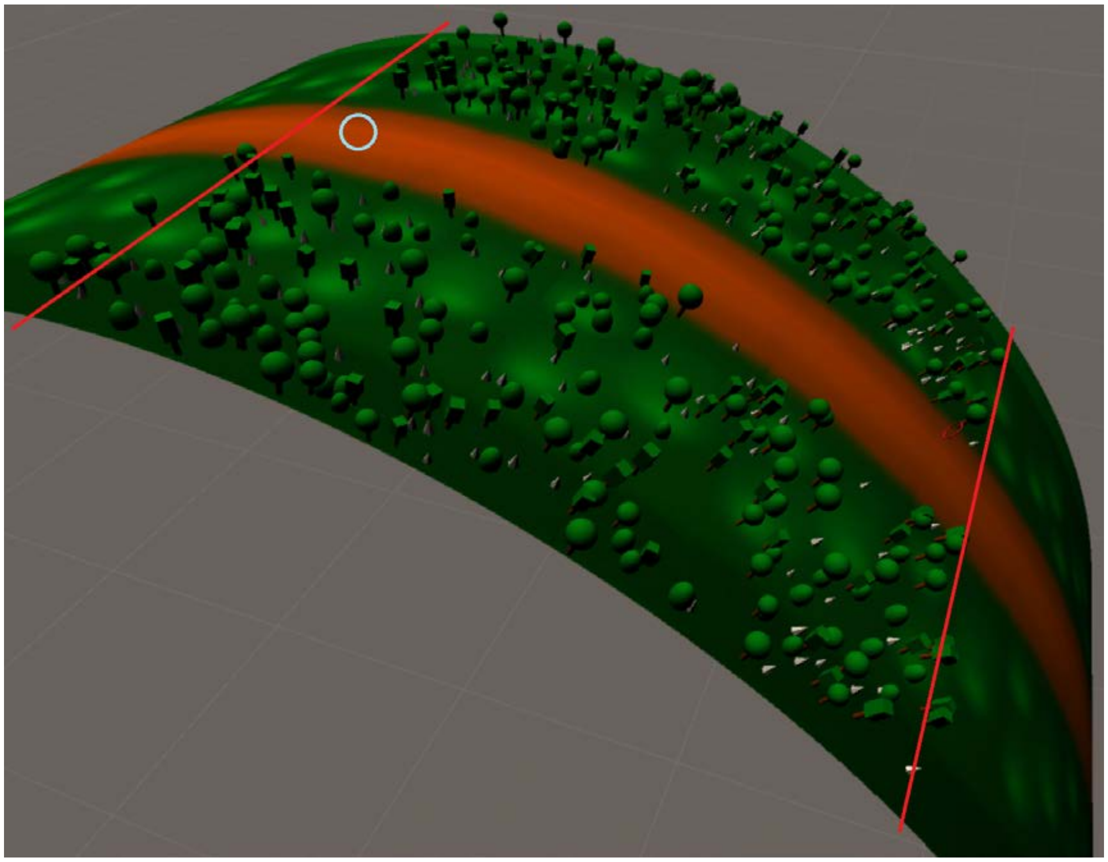
```
            vertical_speed = -vertical_limit * Time.deltaTime;

        if (plyr.transform.position.y < radius + down_bound && vertical_speed <
0)//prevent ground clip
        {
            vertical_speed = 0;//hit ground and speed->0
        }
        else if (plyr.transform.position.y > radius + up_bound && vertical_speed > 0)
        {
            vertical_speed = 0;//hit limit and speed->0
        }
        else
            plyr.transform.Translate(0, vertical_speed, 0);
        //////////////////////
```

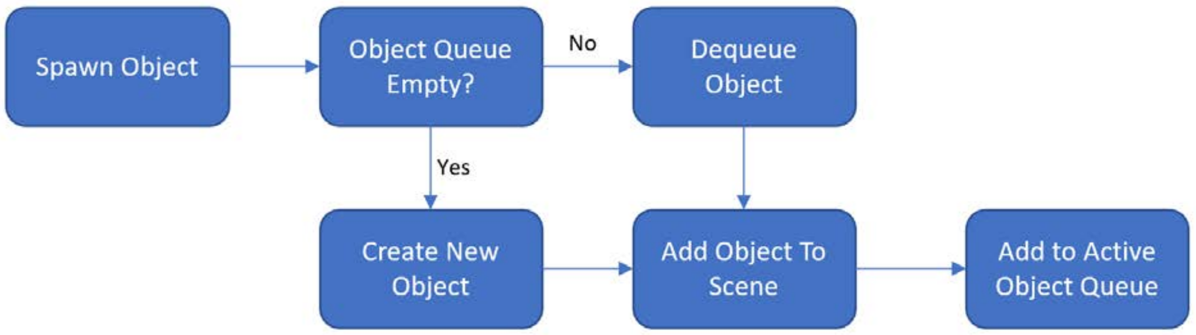
Gravity and drag are only applied if there is no player input, this kept the up and down speeds even when the player originally could apply thrust in both directions of the vertical axis. Drag is only applied in the opposite direction of velocity and not allowed to result in speed in the opposite direction. The vertical speed is then checked to see if it has exceeded the limit and adjusted if necessary. The player is then moved vertically by the speed only if the resulting transform would not push the player out of the play area.

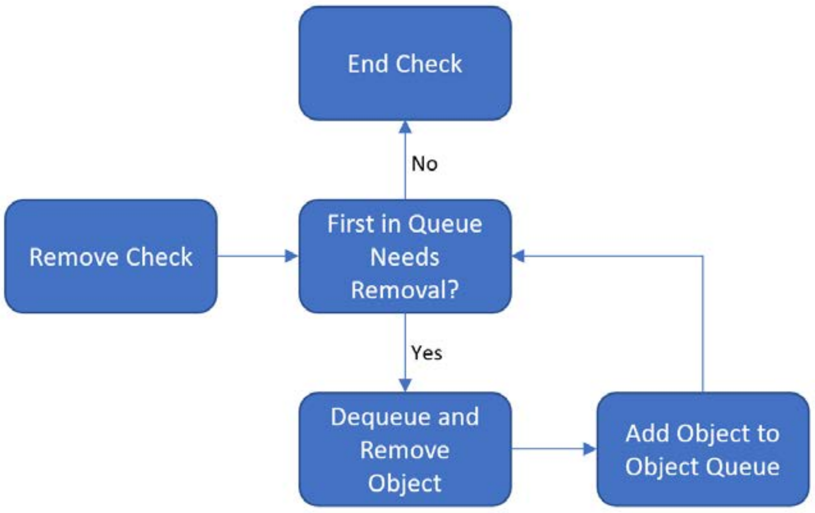
2. WorldGen:

The largest and most complex script is WorldGen. The reason for this complexity is the way in which the number of elements in the scene is limited to increase performance.



Because the curvature of the ground limits the range of the players vision, objects are added to the scene 45 degrees in front of the player and removed when they are more than 10 units behind the player. This area is represented in the image above by red lines, the player's location is indicated by the blue circle. To increase performance when checking if an object needs to be removed from the scene, objects that are added to the scene are also added to an active object queue. This means that only the first object in the queue needs to be checked each frame to see if it needs to be removed, instead of each object in the scene.




End Check
No
Remove Check
First in Queue Needs Removal?
Yes
Dequeue and Remove Object
Add Object to Object Queue

Objects removed from the scene are added to queues of like objects, i.e. a square tree is added to a queue of square trees as shown in the code below.

```
int count = 0;
    void remove_objects()
    {
        while (true)
        {
            if (activeObjects.Count != 0 && activeObjects.Peek().transform.position.z < -
10)
            {
                var temp = activeObjects.Dequeue();
                temp.transform.parent = null;//set to global coord system
                temp.SetActive(false);

                switch (temp.name)
                {
                    case "object_1(Clone)":
                        objectQueue_1.Enqueue(temp);
                        break;
                    case "object_2(Clone)":
                        objectQueue_2.Enqueue(temp);
                        break;
                    case "object_3(Clone)":
                        objectQueue_3.Enqueue(temp);
                        break;
                    case "object_4(Clone)":
                        objectQueue_4.Enqueue(temp);
                        break;
                    case "ring(Clone)":
                        ringQueue.Enqueue(temp);
                        GetComponent<ScoreManager>().ringcheck();//check if missed ring
                        break;
                    default:
                        Debug.Log("Despawn sort issue " + count++ + "   " +
temp.name.ToString());
                        Destroy(temp);
                        break;
                };
            }
            else
                break;
        }
    }
```

The code for the removal of objects was designed to be expandable and not rely on the actual names of environmental objects. This allows for objects to be added, changed or removed easily in the future.

```
void spawn_object(float y, float z, float angle)
    {
        for (int i = -1; i <= 1; i += 2)
        {
            GameObject obj = get_object();
            obj.SetActive(true);

            //////size
            float scale = Random.Range(0.8f, 1.2f);
            obj.transform.localScale = new Vector3(scale, scale, scale);

            //////location
            obj.transform.position = new Vector3(Random.Range(20 * i, 100 * i), y, z);

            ///////rotation
            obj.transform.rotation = new Quaternion(0, 0, 0, 0);
            obj.transform.Rotate(angle + Random.Range(-skew, skew), 0, 0);///////fix
rotation around relative axis
            obj.transform.RotateAround(obj.transform.up, Random.Range(0, 360));

            obj.transform.parent = cyl.transform;
            activeObjects.Enqueue(obj);
        }
    }
```

When an object is added to the scene it needs to be positioned correctly. Because the objects always spawn 45 degrees from the player the x and y coordinates are known beforehand and are precalculated when the game starts. Environmental objects are spawned on each side of the play area and its x coordinate is generated randomly. Currently objects spawn every 0.2 degrees the player travels. The size and rotation of the object are also randomized to create a more natural looking environment. After the object is created it is added to the ground object and begins traveling with the rotation of the ground. Rings are added in a separate process that is very similar. The space between rings is currently 15 degrees.

3. Miscellaneous: ScoreManager, StaticScore and TextManager

When the player flies through a ring the ScoreManager is called. After hitting a ring the players ring multiplier increases, this increases the points the player earns as a function of time and raises the forward speed of the player which also increases the difficulty of the game.

```
public void ringhit()
    {
        hitring = true;
        ringmult++;
        if (ringmult >= 5)
            ringmult = 5;

        updatespeed();
    }
```

Ringmult is a value between zero and five, this means the players score/second ranges from 1x to 6x. Each level of ringmult will increase the forward speed by 20%, up to 100% extra speed. The bool hitring is used to check if a ring has been missed.

```
public void ringcheck()
    {
        if (hitring == false)
            ringmiss();

        hitring = false;
    }
```

If the flag is not set to true and a ring is removed then the player missed that ring. When the player misses 10 rings the game ends. The ScoreManager also handles game overs by updating the StaticScore variables and loading the Main Menu scene. When the Main Menu is loaded the TextManager script updates the current and high scores.

Future Plans:

Most of the variables used for controls and generating the environment can be changed internally. This can be used in the future to add different difficulty levels by changing the forward speed and ring separation. Also, changing the variables can increase or decrease the amount and types of objects in the scene. This means in the future the game can be played in different environments like a desert, tundra or grassland. These variables can also be changed by the player if an options menu is added in the future. Finally, adding sound effects and music can greatly increase the immersion and enjoyment of the player in future builds.

Faculty Advisor:

Prof. Gordon Stein