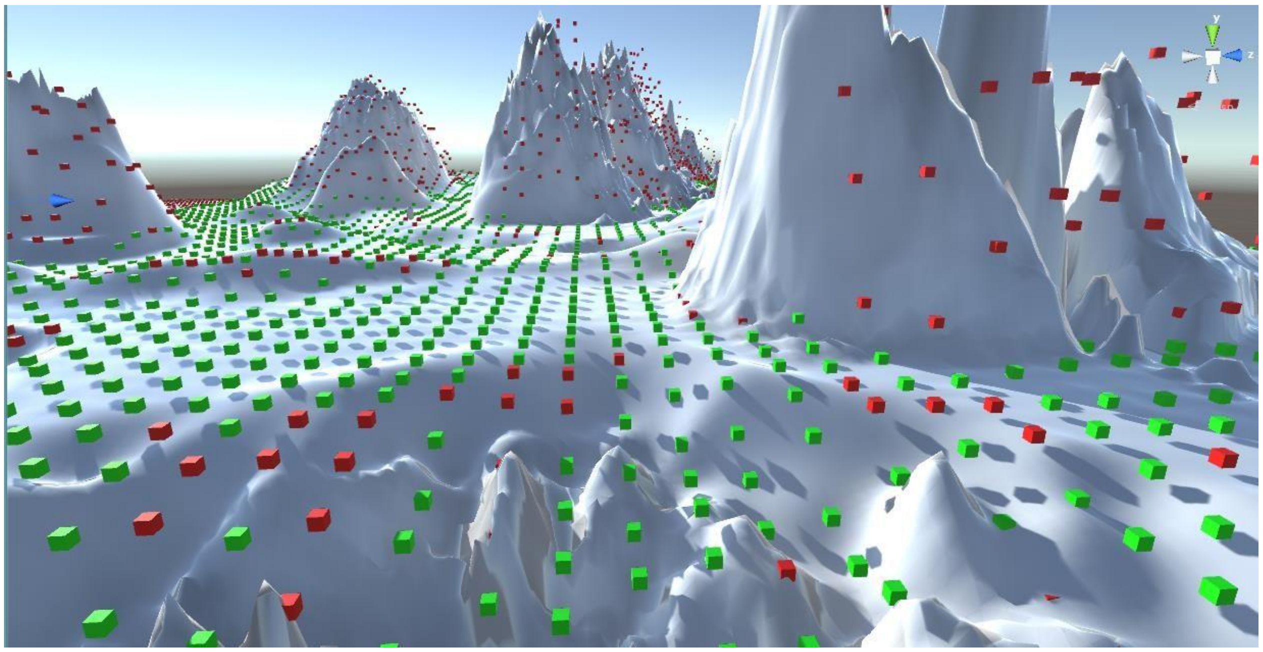
# Games AI Coursework 2

Student: Calvin Fuss

Lecturer: Dr Jeremy Gow



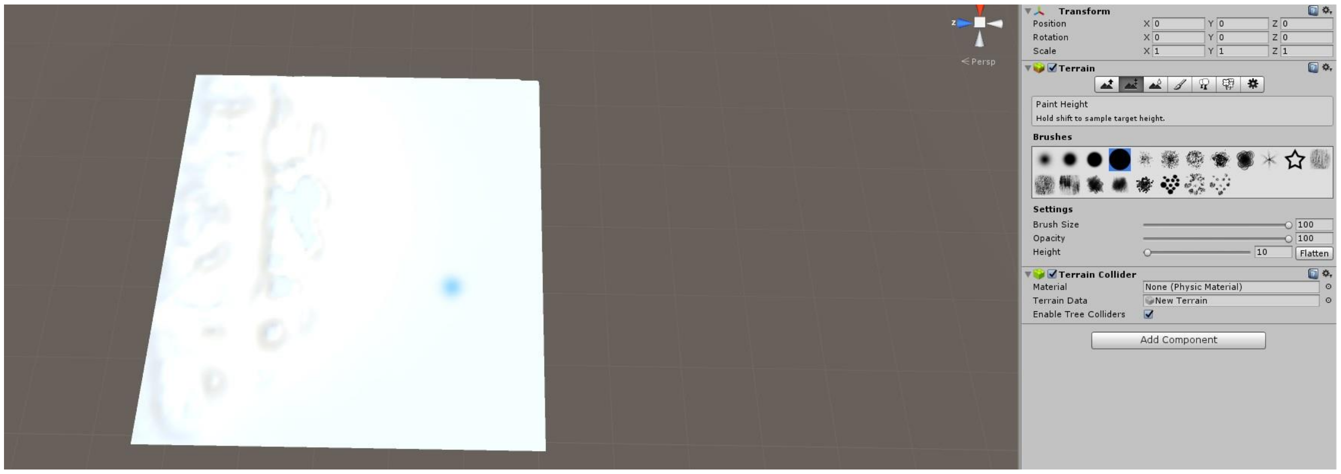
**Contents** **Page**

## Introduction

For Part two of the Games AI coursework I created a dynamic terrain waypoint generation algorithm. These waypoints would act as the walkable and unwalkable areas within the map. Using these traversable waypoints, I created a pathfinding algorithm to demonstrate the functionality of the waypoint generation algorithm. Below are the steps taken to produce the final result.

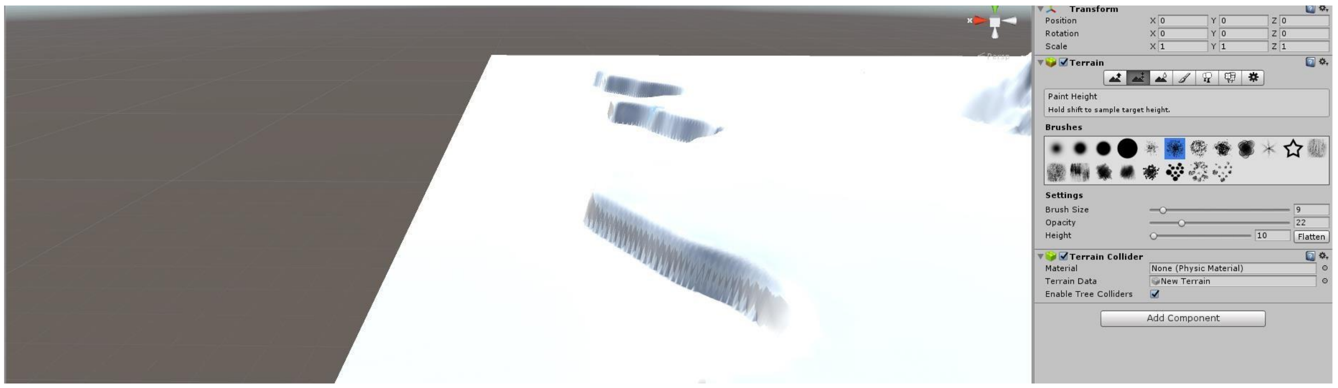
## Terrain Map Setup

I started with a basic 3D terrain plane object.



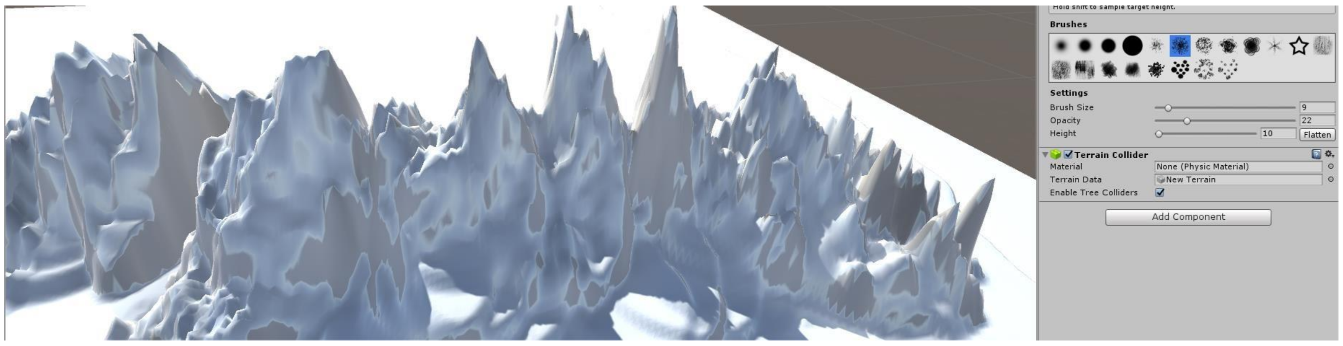
```
Transform
Position          X 0     Y 0     Z 0
Rotation          X 0     Y 0     Z 0
Scale             X 1     Y 1     Z 1
Terrain
Paint Height
Hold shift to sample target height.
Brushes
Settings
Brush Size                 100
Opacity                    100
Height               10    Flatten
Terrain Collider
Material              None (Physic Material)
Terrain Data          New Terrain
Enable Tree Colliders ✓
Add Component
```

I used the various brush tools to shape and structure to create varied terrain.



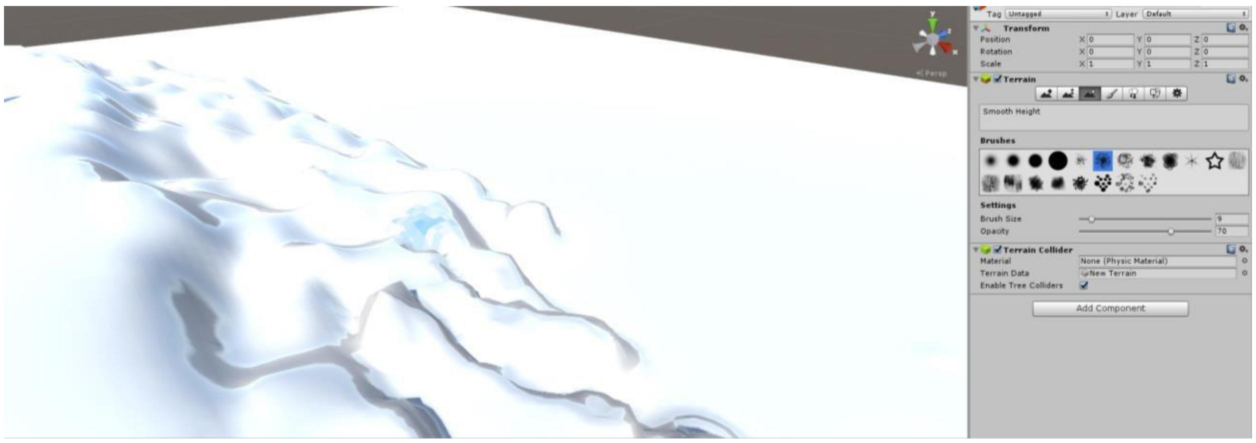
```
Transform
Position          X 0     Y 0     Z 0
Rotation          X 0     Y 0     Z 0
Scale             X 1     Y 1     Z 1
Terrain
Paint Height
Hold shift to sample target height.
Brushes
Settings
Brush Size                 9
Opacity                    22
Height               10    Flatten
Terrain Collider
Material              None (Physic Material)
Terrain Data          New Terrain
Enable Tree Colliders ✓
Add Component
```

I created peaks and troughs as these features would be present in a real-world game map

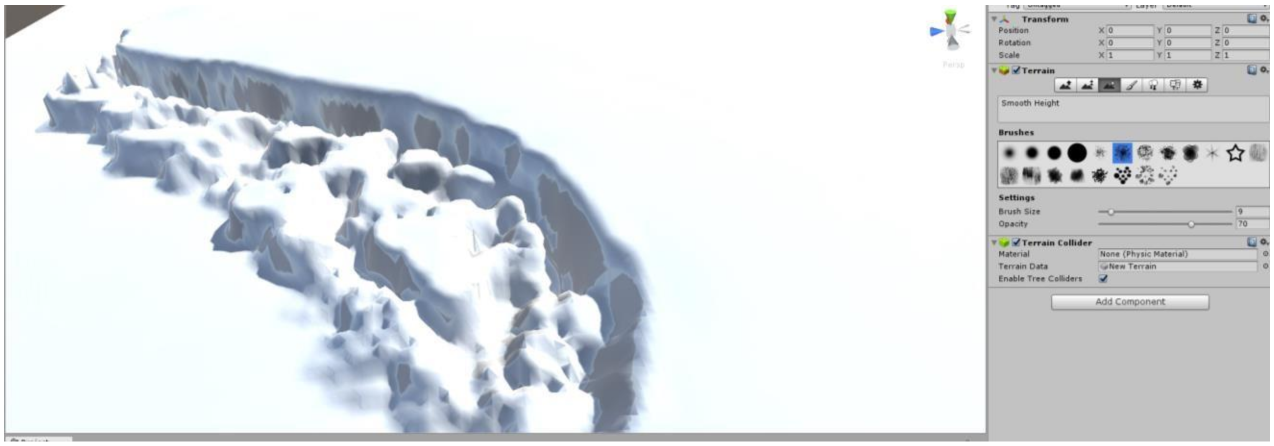


```
Hold shift to sample target height.
Brushes
Settings
Brush Size                 9
Opacity                    22
Height               10    Flatten
Terrain Collider
Material              None (Physic Material)
Terrain Data          New Terrain
Enable Tree Colliders ✓
Add Component
```

The terrain acted as an island. This would allow me to define large unwalkable areas (water)

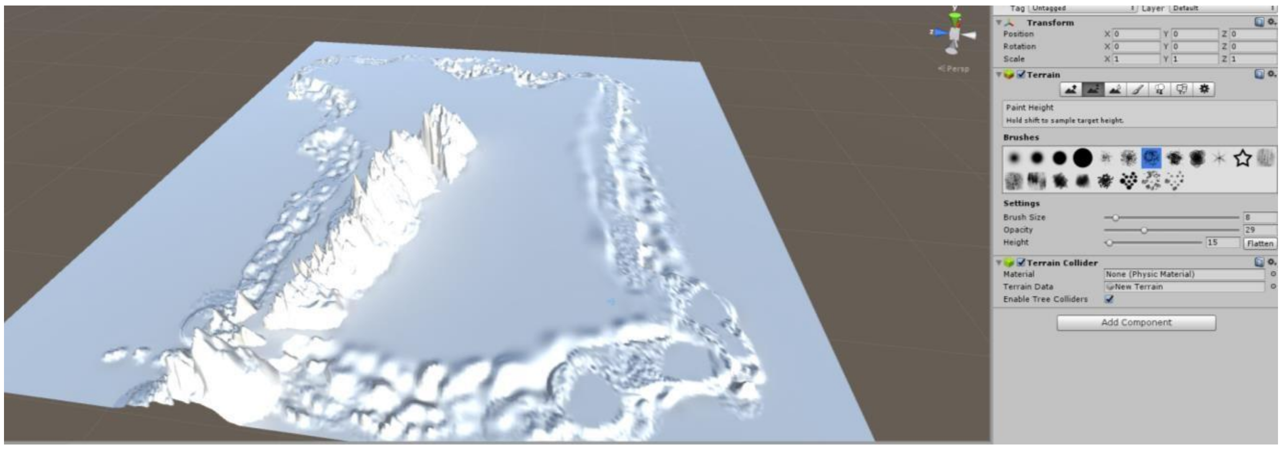


Tag Untagged Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Terrain
Smooth Height
Brushes
Settings
Brush Size 9
Opacity 70
Terrain Collider
Material None (Physic Material)
Terrain Data New Terrain
Enable Tree Colliders ✓
Add Component



Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Terrain
Smooth Height
Brushes
Settings
Brush Size 9
Opacity 70
Terrain Collider
Material None (Physic Material)
Terrain Data New Terrain
Enable Tree Colliders ✓
Add Component

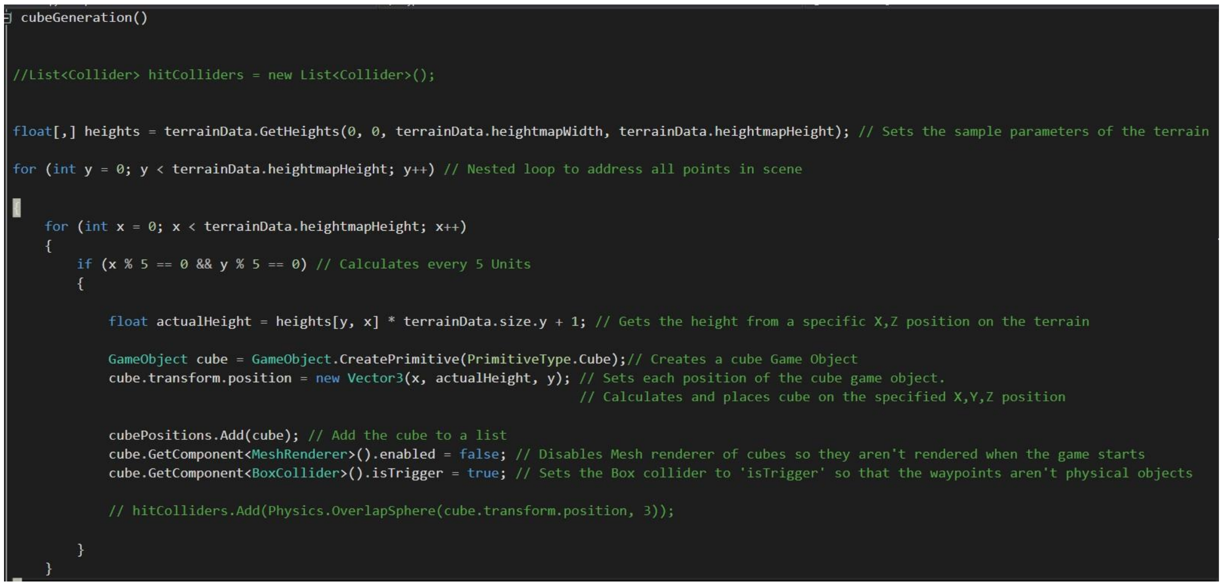
The final map design



Tag Untagged Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Terrain
Paint Height
Hold shift to sample target height.
Brushes
Settings
Brush Size 8
Opacity 29
Height 15 Flatten
Terrain Collider
Material None (Physic Material)
Terrain Data New Terrain
Enable Tree Colliders ✓
Add Component

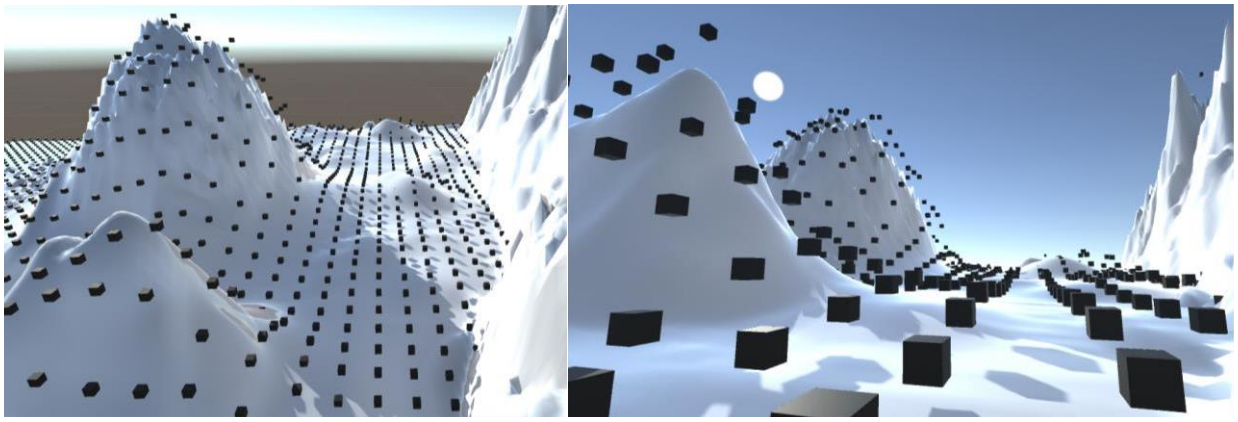
I now had an example of varied terrain which I used to create my waypoint algorithm. I used the terrain's data (selectable with public variable) to calculate the height map of the terrain to systematically cover the surface of the terrain with waypoints. A range of parameters then defined the walkable and unwalkable areas.

I used the ‘terrainData.GetHeights’ function to store the height variables in a double float. A nested for loop addressed each point in the scene. To reduce processing outlay, I set a sample rate to calculate positions every 5 units. The ‘actualHeight’ variable calculated the Y position of the terrain data at a specific position. A cube was then placed at each calculated instance and saved into a list named ‘cubePositions’. I disabled the mesh renderer of the cubes so that cubes aren’t rendered when the scene starts and set the BoxCollider of each cube to act as a trigger instead of a physical object.



```
cubeGeneration()


//List<Collider> hitColliders = new List<Collider>();


float[,] heights = terrainData.GetHeights(0, 0, terrainData.heightmapWidth, terrainData.heightmapHeight); // Sets the sample parameters of the terrain

for (int y = 0; y < terrainData.heightmapHeight; y++) // Nested loop to address all points in scene


    for (int x = 0; x < terrainData.heightmapHeight; x++)
    {
        if (x % 5 == 0 && y % 5 == 0) // Calculates every 5 Units
        {

            float actualHeight = heights[y, x] * terrainData.size.y + 1; // Gets the height from a specific X,Z position on the terrain

            GameObject cube = GameObject.CreatePrimitive(PrimitiveType.Cube);// Creates a cube Game Object
            cube.transform.position = new Vector3(x, actualHeight, y); // Sets each position of the cube game object.
                                                                      // Calculates and places cube on the specified X,Y,Z position

            cubePositions.Add(cube); // Add the cube to a list
            cube.GetComponent<MeshRenderer>().enabled = false; // Disables Mesh renderer of cubes so they aren't rendered when the game starts
            cube.GetComponent<BoxCollider>().isTrigger = true; // Sets the Box collider to 'isTrigger' so that the waypoints aren't physical objects

            // hitColliders.Add(Physics.OverlapSphere(cube.transform.position, 3));

        }
    }
```

The waypoints covering the scene

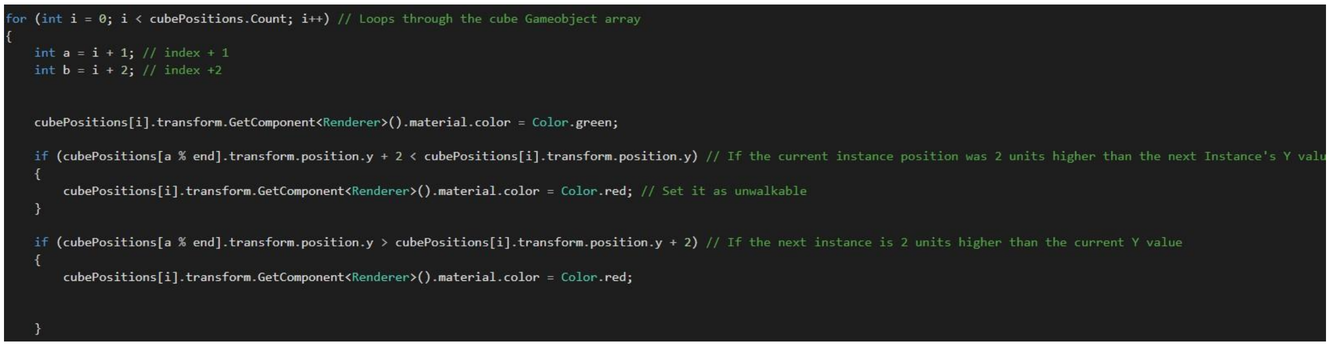


I needed to set the walkable and unwalkable areas of the map by creating conditions based on the positions of waypoints. To begin, I set the colour of each waypoint to green. This would later be the defining colour of walkable waypoints. The subsequent parameterisation would define the unwalkable waypoints.

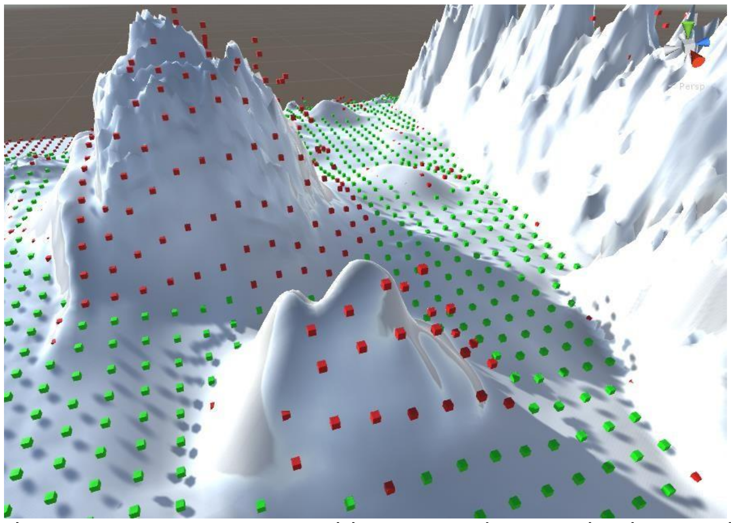
I began with setting the waypoints which were on a sharp terrain incline/decline as unwalkable. To achieve this, I used a for loop to address each waypoint in the ‘cubePositions’ list. The unwalkable waypoints were classified using an ‘If’ statement within the for loop.

The if statement was defined as true if the current waypoint Y position in the for loop was greater or smaller than two units to that of the next instance in the for loop. This meant that waypoints on an incline/decline which had a difference of 2 units or higher between each instance were set to unwalkable.

The code which defined the unwalkable inclined/declined waypoints.



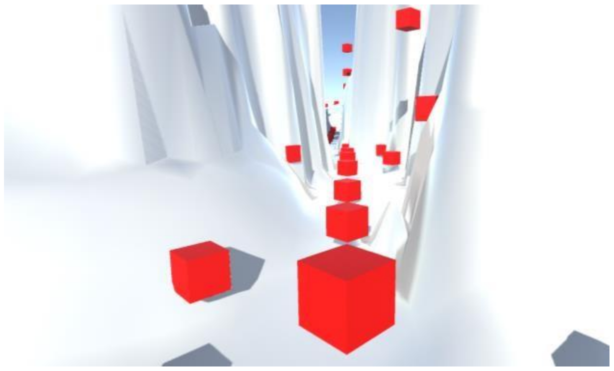
```
for (int i = 0; i < cubePositions.Count; i++) // Loops through the cube Gameobject array
{
    int a = i + 1; // index + 1
    int b = i + 2; // index +2


    cubePositions[i].transform.GetComponent<Renderer>().material.color = Color.green;

    if (cubePositions[a % end].transform.position.y + 2 < cubePositions[i].transform.position.y) // If the current instance position was 2 units higher than the next Instance's Y valu
    {
        cubePositions[i].transform.GetComponent<Renderer>().material.color = Color.red; // Set it as unwalkable
    }

    if (cubePositions[a % end].transform.position.y > cubePositions[i].transform.position.y + 2) // If the next instance is 2 units higher than the current Y value
    {
        cubePositions[i].transform.GetComponent<Renderer>().material.color = Color.red;


    }
```

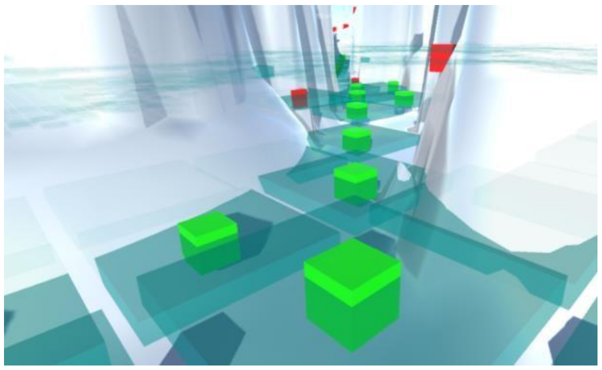


A visual representation of the walkable/unwalkable waypoints.

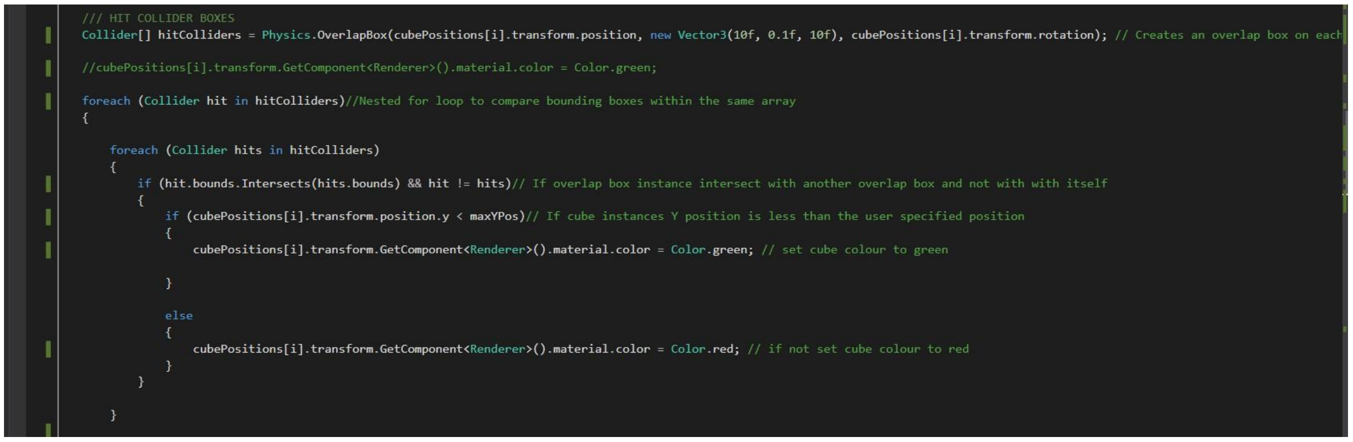
Although the waypoints within areas of incline/decline were parameterised, there was an issue with waypoints that were placed within narrow paths surrounded by high peaks. This was because the Y position difference of the current to the neighbouring waypoints were outside of the defined parameter. The waypoints were therefore rendered as unwalkable when they should in fact be traversable by the agent. To set these waypoints to traversable, I created an overlap box on the position of each waypoint under a certain Y value on the terrain. If two overlap boxes were colliding, the objects within were set to traversable.

An example of this issue can be seen on the left. Bounding boxes solved this issue as seen on the right.



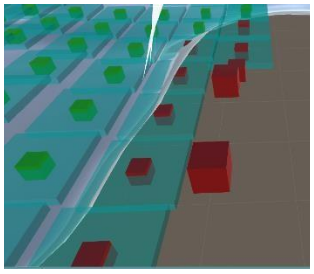


Below is the code for the OverlapBoxes. I created an overlap box on each cube position and stored these in a list. I then used a nested for loop to compare the Overlap boxes with each other to see if boxes were touching. If they were, then the waypoints would be set to walkable (green).

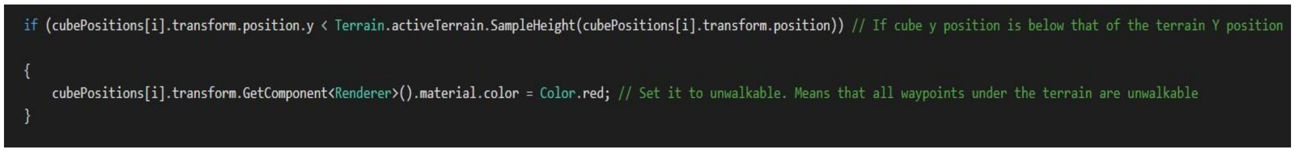


```
/// HIT COLLIDER BOXES
Collider[] hitColliders = Physics.OverlapBox(cubePositions[i].transform.position, new Vector3(10f, 0.1f, 10f), cubePositions[i].transform.rotation); // Creates an overlap box on each

//cubePositions[i].transform.GetComponent<Renderer>().material.color = Color.green;

foreach (Collider hit in hitColliders)//Nested for loop to compare bounding boxes within the same array
{

    foreach (Collider hits in hitColliders)
    {
        if (hit.bounds.Intersects(hits.bounds) && hit != hits)// If overlap box instance intersect with another overlap box and not with with itself
        {
            if (cubePositions[i].transform.position.y < maxYPos)// If cube instances Y position is less than the user specified position
            {
                cubePositions[i].transform.GetComponent<Renderer>().material.color = Color.green; // set cube colour to green

            }

            else
            {
                cubePositions[i].transform.GetComponent<Renderer>().material.color = Color.red; // if not set cube colour to red
            }
        }

    }
```

Some waypoints were placed under the terrain map. This usually occurred on steep inclines/declines. I set each object which was under the terrain map to unwalkable in order to prevent errors.



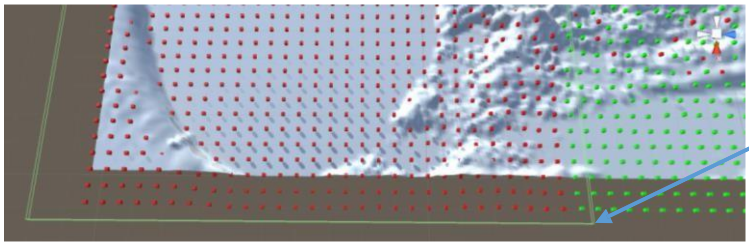
This was achieved with the following code:



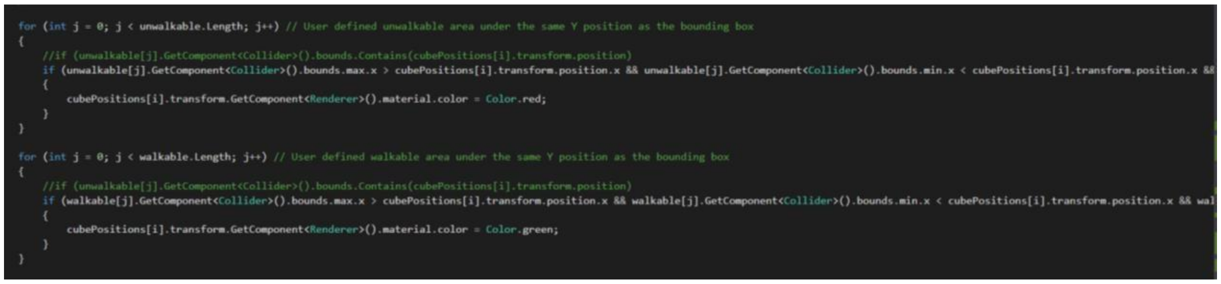
```
if (cubePositions[i].transform.position.y < Terrain.activeTerrain.SampleHeight(cubePositions[i].transform.position)) // If cube y position is below that of the terrain Y position

{
    cubePositions[i].transform.GetComponent<Renderer>().material.color = Color.red; // Set it to unwalkable. Means that all waypoints under the terrain are unwalkable
}
```

I now had the waypoints parameterised by their respective positions. However, I wanted to create user flexibility that would allow for an entire area to be classes as walkable/unwalkable regardless of the parameters, simply by placing a tagged object above an area in the scene. To achieve this, I created two new game object tags – walkable and unwalkable. The user would create a game object, tag it and place it over the required area. Any waypoints below that tagged game object would either be set to walkable or unwalkable. This is useful for creating out of bounds areas such as in water.

Placing an “Unwalkable” tagged game object over an area of waypoints.  These are then classified as unwalkable.

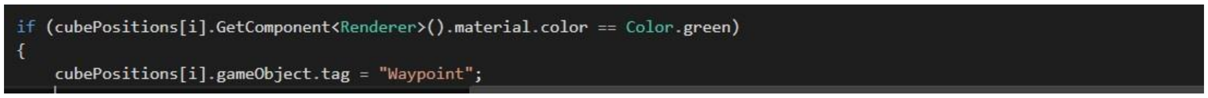


In order to accomplish this in code, I took the x,y,z coordinate parameters of the unwalkable/walkable tagged game objects in the scene and compared these to each instance of the waypoint. Every waypoint which had the same X and Z and a smaller Y coordinate of the tagged game object’s collision box was defined as walkable/unwalkable. This gave the user flexibility when applying the algorithm to their own terrain.



```
for (int j = 0; j < unwalkable.Length; j++) // User defined unwalkable area under the same Y position as the bounding box
{
    //if (unwalkable[j].GetComponent<Collider>().bounds.Contains(cubePositions[i].transform.position)
    if (unwalkable[j].GetComponent<Collider>().bounds.max.x > cubePositions[i].transform.position.x && unwalkable[j].GetComponent<Collider>().bounds.min.x < cubePositions[i].transform.position.x &&
    {
        cubePositions[i].transform.GetComponent<Renderer>().material.color = Color.red;
    }
}

for (int j = 0; j < walkable.Length; j++) // User defined walkable area under the same Y position as the bounding box
{
    //if (unwalkable[j].GetComponent<Collider>().bounds.Contains(cubePositions[i].transform.position)
    if (walkable[j].GetComponent<Collider>().bounds.max.x > cubePositions[i].transform.position.x && walkable[j].GetComponent<Collider>().bounds.min.x < cubePositions[i].transform.position.x && wal
    {
        cubePositions[i].transform.GetComponent<Renderer>().material.color = Color.green;
    }
}
```

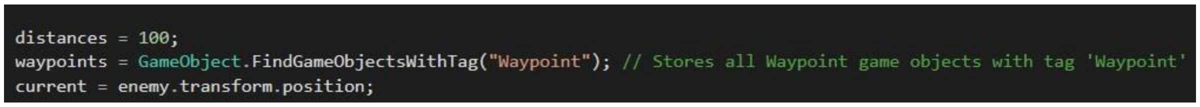
Finally, each game object with the colour green (walkable waypoint) was tagged as “Waypoint” in the scene.



```
if (cubePositions[i].GetComponent<Renderer>().material.color == Color.green)
{
    cubePositions[i].gameObject.tag = "Waypoint";
```

**Testing Waypoints with pathfinding algorithm**

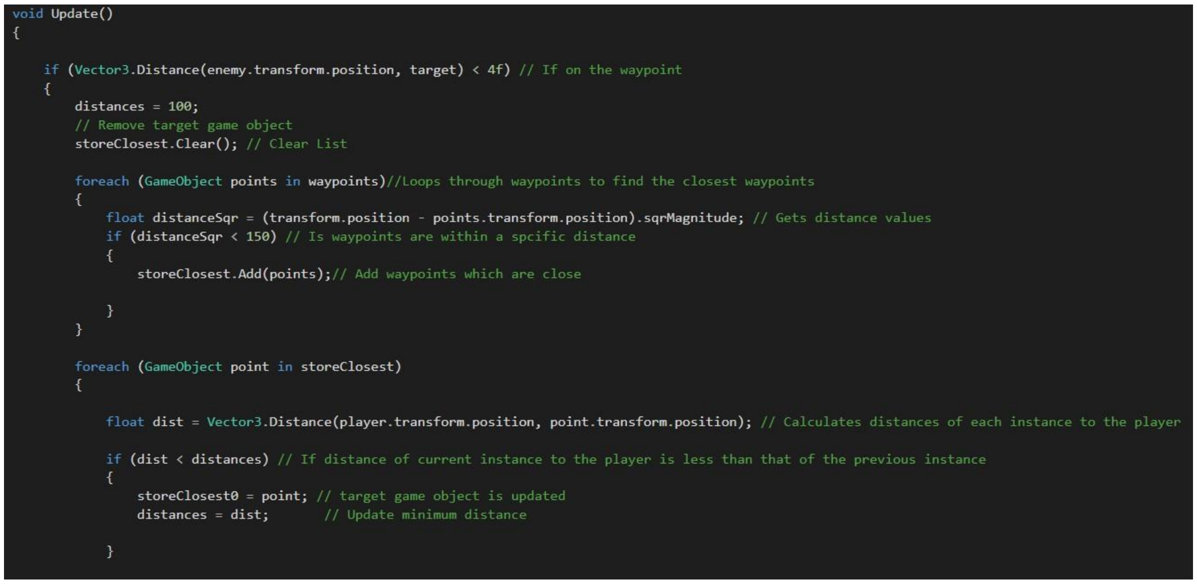
I created a pathfinding algorithm to test the functionality of the traversable waypoints. I created an array called ‘waypoints’ which contained all game objects tagged as “Waypoint” in the scene. I additionally created a list called ‘storeClosest’ which stored waypoints within a certain distance to the enemy transform.



```
distances = 100;
waypoints = GameObject.FindGameObjectsWithTag("Waypoint"); // Stores all Waypoint game objects with tag 'Waypoint'
current = enemy.transform.position;
```

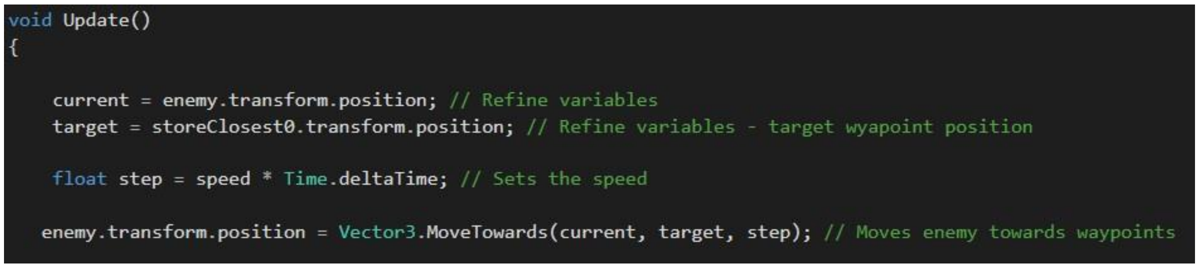
In order to do this, I used a for loop which looped through all the instances in the waypoints array. The position from each waypoint to transform was calculated. Any objects within a distance of 150 units to the transform were added to the ‘storeClosest’ list. I then used Unity’s distance function to calculate the distance from each instance in the ‘storeClosest’ list to find the game object which was closest to the player’s position. The closest game object to the player in the ‘storeClosest’ list was then set to the target waypoint.

This code is run if the distance to the enemy is less than 4 units. This allows the target waypoint to be updated once and only when the waypoint has been reached. When this ‘If’ statement is true, the element’s in the ‘storeClosest’ list is removed and the ‘distance’ variable is reset.



```
void Update()
{

    if (Vector3.Distance(enemy.transform.position, target) < 4f) // If on the waypoint
    {
        distances = 100;
        // Remove target game object
        storeClosest.Clear(); // Clear List

        foreach (GameObject points in waypoints)//Loops through waypoints to find the closest waypoints
        {
            float distanceSqr = (transform.position - points.transform.position).sqrMagnitude; // Gets distance values
            if (distanceSqr < 150) // Is waypoints are within a spcific distance
            {
                storeClosest.Add(points);// Add waypoints which are close

            }
        }

        foreach (GameObject point in storeClosest)
        {

            float dist = Vector3.Distance(player.transform.position, point.transform.position); // Calculates distances of each instance to the player

            if (dist < distances) // If distance of current instance to the player is less than that of the previous instance
            {
                storeClosest0 = point; // target game object is updated
                distances = dist;       // Update minimum distance

            }
```

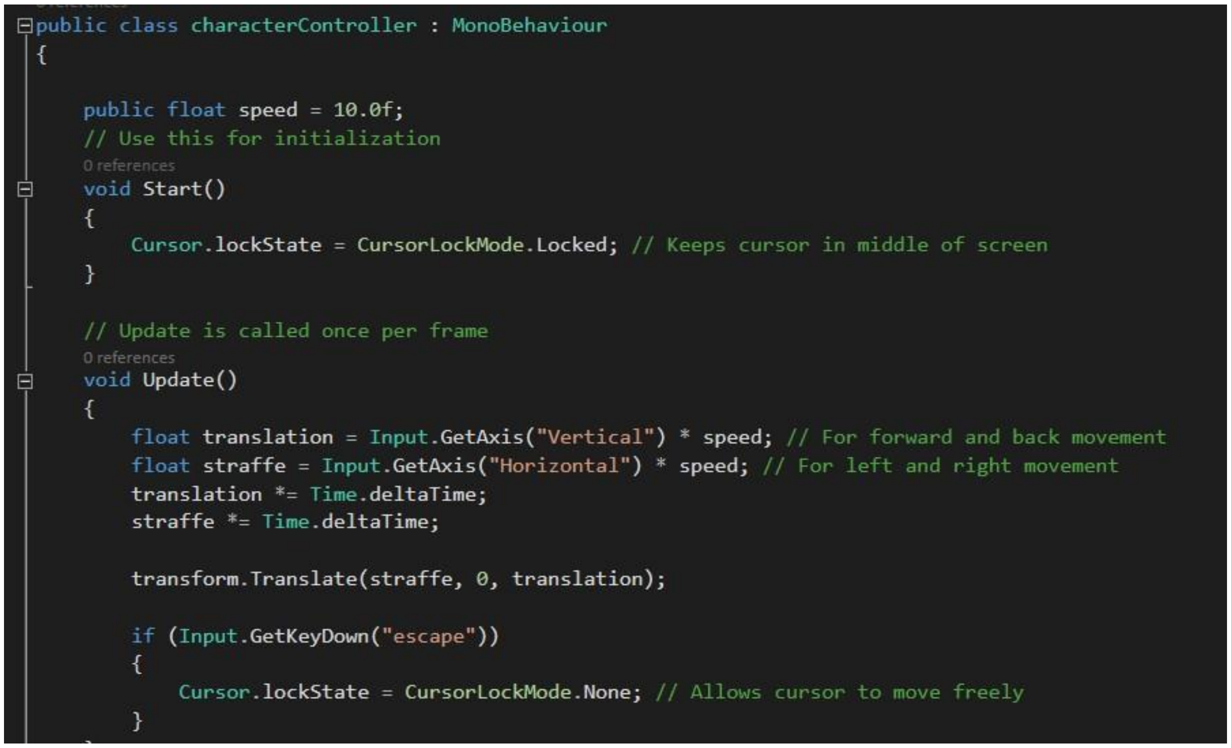
Move enemy toward target waypoint code:



```
void Update()
{

    current = enemy.transform.position; // Refine variables
    target = storeClosest0.transform.position; // Refine variables - target wyapoint position

    float step = speed * Time.deltaTime; // Sets the speed

   enemy.transform.position = Vector3.MoveTowards(current, target, step); // Moves enemy towards waypoints
```

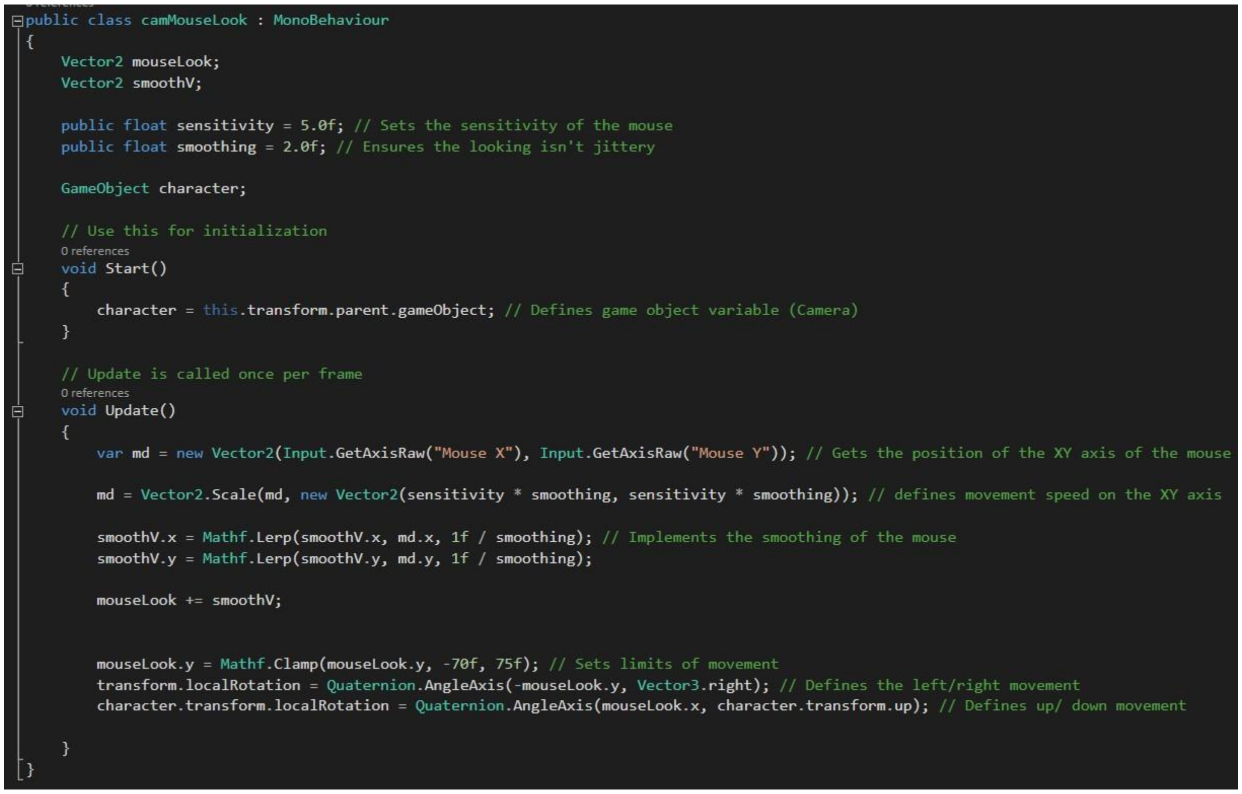
**Other scripts**

Player Movement



```
public class characterController : MonoBehaviour
{

    public float speed = 10.0f;
    // Use this for initialization
    0 references
    void Start()
    {
        Cursor.lockState = CursorLockMode.Locked; // Keeps cursor in middle of screen
    }

    // Update is called once per frame
    0 references
    void Update()
    {
        float translation = Input.GetAxis("Vertical") * speed; // For forward and back movement
        float straffe = Input.GetAxis("Horizontal") * speed; // For left and right movement
        translation *= Time.deltaTime;
        straffe *= Time.deltaTime;

        transform.Translate(straffe, 0, translation);

        if (Input.GetKeyDown("escape"))
        {
            Cursor.lockState = CursorLockMode.None; // Allows cursor to move freely
        }
```

Player Look



```
public class camMouseLook : MonoBehaviour
{
    Vector2 mouseLook;
    Vector2 smoothV;

    public float sensitivity = 5.0f; // Sets the sensitivity of the mouse
    public float smoothing = 2.0f; // Ensures the looking isn't jittery

    GameObject character;

    // Use this for initialization
    0 references
    void Start()
    {
        character = this.transform.parent.gameObject; // Defines game object variable (Camera)
    }

    // Update is called once per frame
    0 references
    void Update()
    {
        var md = new Vector2(Input.GetAxisRaw("Mouse X"), Input.GetAxisRaw("Mouse Y")); // Gets the position of the XY axis of the mouse

        md = Vector2.Scale(md, new Vector2(sensitivity * smoothing, sensitivity * smoothing)); // defines movement speed on the XY axis

        smoothV.x = Mathf.Lerp(smoothV.x, md.x, 1f / smoothing); // Implements the smoothing of the mouse
        smoothV.y = Mathf.Lerp(smoothV.y, md.y, 1f / smoothing);

        mouseLook += smoothV;


        mouseLook.y = Mathf.Clamp(mouseLook.y, -70f, 75f); // Sets limits of movement
        transform.localRotation = Quaternion.AngleAxis(-mouseLook.y, Vector3.right); // Defines the left/right movement
        character.transform.localRotation = Quaternion.AngleAxis(mouseLook.x, character.transform.up); // Defines up/ down movement

    }
}
```

## Conclusion

I found the second Games AI Coursework 2 very challenging. However, taking one step at a time I made progress and pieced together each feature. I learnt a lot about the capabilities of Unity as well as many new features of C# which I believe I would not have discovered if I hadn't taken this module in Games AI. Although it took a lot of research to discover the resolutions to small issues as well as many hours of trial and error to get the desired results, I found this project extremely enjoyable and I am proud of the outcome.

## Bibliography

Unity's scripting tutroials (Unity, n.d.) –

Unity's User Manual (Unity, n.d.)

Unity answers (Various authors, n.d.)

Changing Object Colour (jbloon, 2013)

Move Object Toward Closest Enemy (Musty, 2015)

First Person Controller (Hollistic3D, 2017)

Rotate Object - (Unity, 2017)

Find closest enemy (Rider, 2017)

Terrain Generation (alucardj, 2013) – I used the 'Best Answer' on this Unity forum question to figure out on how to create the basis for the generation of cubes on my created terrain. I translated the code from this forum answer (which was written in Java) into C# and implemented this into my alogrithm which is present within the Waypoint.cs script.

In order to showcase the specific code used from the Unity forum link, I created a script within my project named 'Jeneratio.js'. This script contained the specific code that I used to translate from Java into C#. The 'Jeneratio.js' script has no functionality other than the use as a reference to the specific code used from the link above.