COMP4121 Major Project

Developing a Real Time Strategy Game With Unity

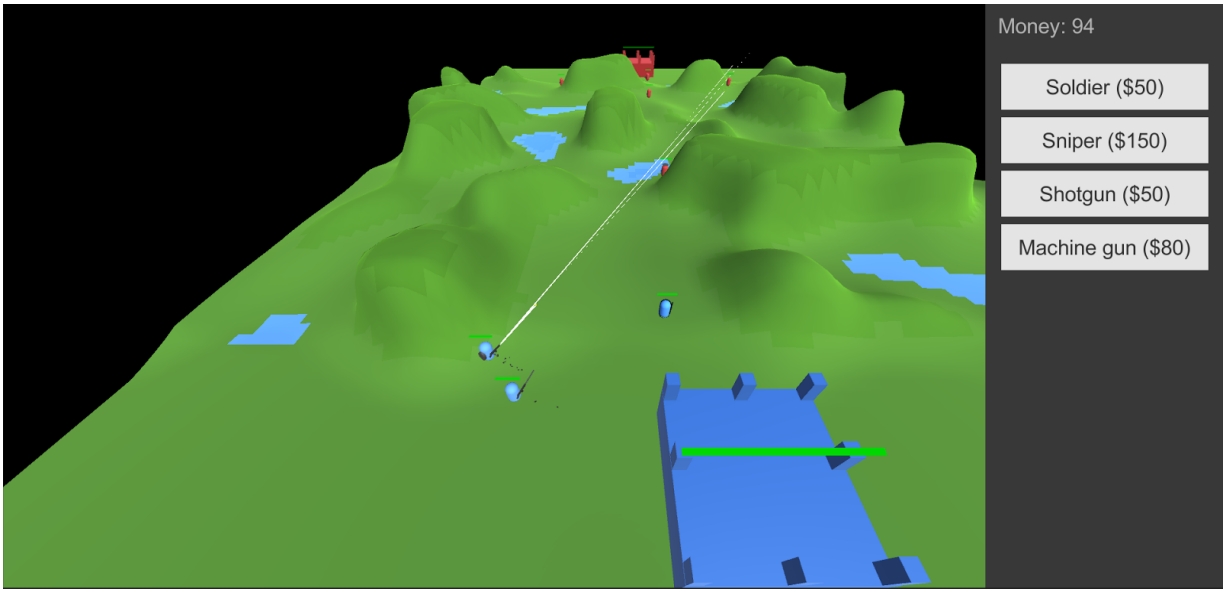
Table of Contents

Game Overview

The game is a 1 vs 1 real time strategy game played against a computer player where each player tries to destroy the opponent's base using different types of units. Units spawn at your base and can be controlled to move around the map. Each unit type has a different range, damage, accuracy, and firing rate which make them better in different scenarios. Players generate money every second which can be spent on spawning units. The player who destroys their opponent's base first wins the game.

The terrain is procedurally generated differently every time a new game is played, so that the player plays on a different map each game. The terrain determines where the units can move and affect where they can shoot. The hills and the water are both unwalkable, however you can shoot across water while hills block field of vision such that the units can not shoot over hills.

The game is written in C# using the Unity game engine.



Game screen which shows the two bases (blue and red buildings) and some units fighting

Unity Overview

Development of the game was done using the Unity game engine. Unity is a cross-platform game engine which supports writing games in C#. Unity comes with a built-in physics engine and is capable of creating both 2D and 3D games.

Entity-Component-System

Unity uses a type of entity-component-system (ECS) where components (scripts) are attached to entities (objects) to affect how they behave. These components can interact with other components within their code, or act independently. For example, a rigidbody component (which handles physics interaction for an object) will assume that there is also a collider component for the object (which defines the shape of the object) in order for the rigidbody to work properly. Alternatively, a component such as “PlayerController” which could be used to control an object using user input does not have to rely on any other component.

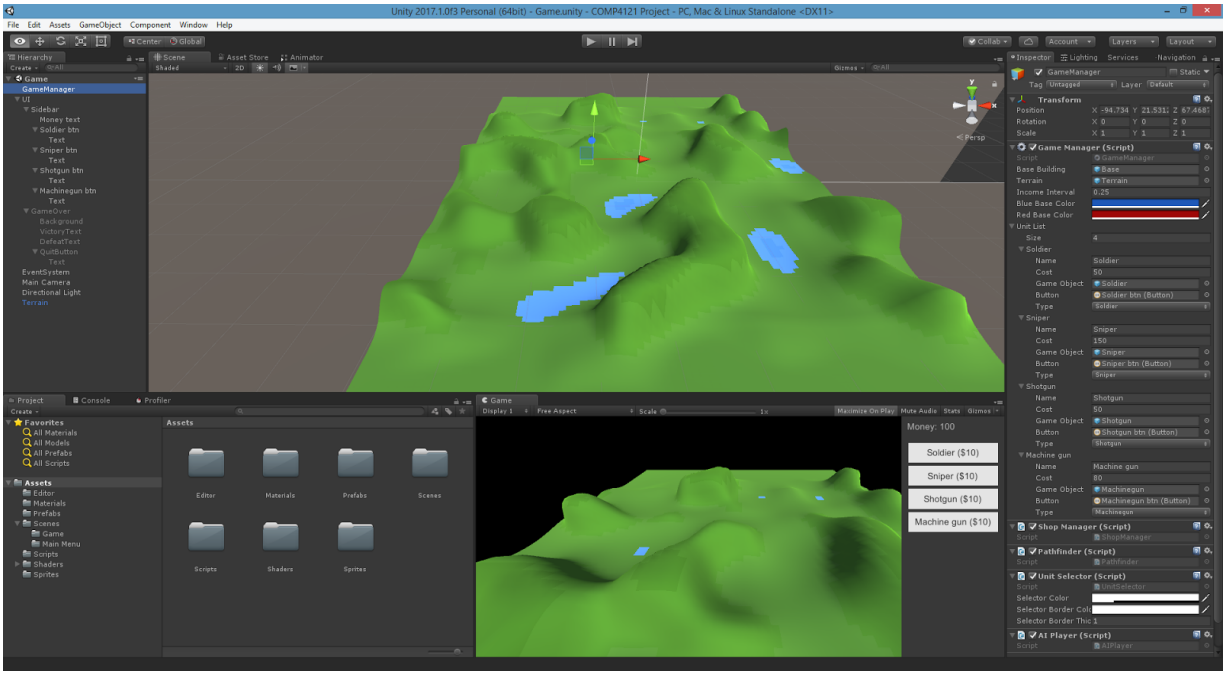
These components can be reused on different objects to achieve the same type of behaviour on different objects. There is no limit to the number of components which can be placed on an object.

Components in Unity inherit from the MonoBehaviour class. These components have a lifecycle loop which is executed independently for each component of an object. The most notable lifecycle functions are:

Awake: Called once when the object is instantiated and before any Start function is called in any component.
Start: Called once after Awake has been called on all the components.
Update: Called once per frame.

Thus, every component may have its own Update function which is executed every frame to process things related to the component.

Integrated Development Environment



Unity's Integrated Development Environment (IDE)

In the image above, the inspector can be seen on the right, which shows all the components that are attached to the currently selected object. Each component may have variables which can be modified in the inspector (for example, the radius of a sphere collider) to change how the component behaves, although sane defaults are provided. This allows variables to be changed without having to go into the code and can allow for different variable values for objects that use the same component.

At the top centre of the image, the current scene is shown. This can be thought of as the editor the developer uses to add objects into the game environment, or look around the current game world when developing the game.

Below this, the actual game view is shown. This is the view which is seen by the user when playing the game. This view is determined by attaching a camera component to an object, which then renders a view of the scene depending on the position and rotation of the object.

Lastly, on the left, the game objects which are currently placed inside the current scene are displayed.

Game Structure and Design

Game Objects

The initial game scene contains some game objects which are required in order for the game to function properly at runtime. These are:

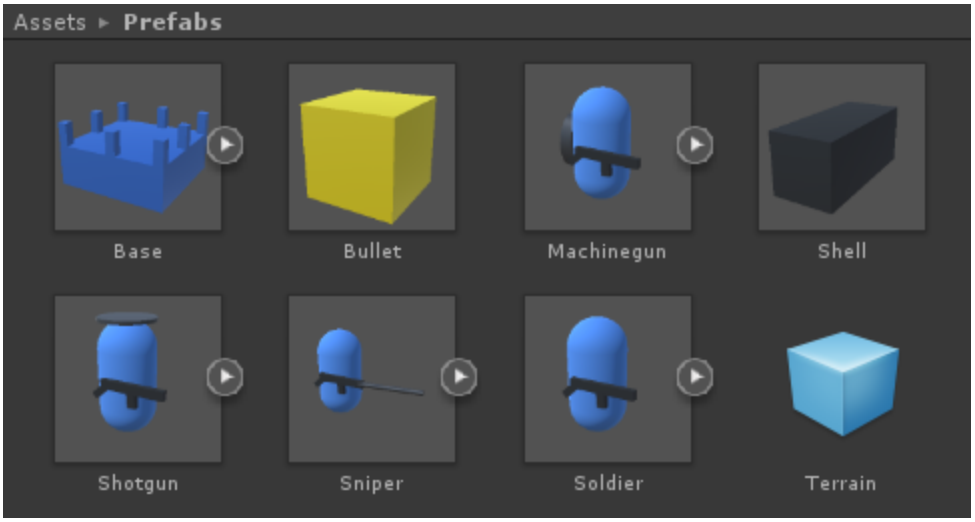
Name	**Description**
GameManager	An empty game object which has components attached to it which manage the current game state. These components are: AIPlayer, UnitSelector, GameManager, ShopManager, and Pathfinder.
UI	A Unity canvas object which contains parts of the game which are not a part of the game world itself, but the interface shown to the user. This object contains a GameUI component which handles button presses on the UI and updates the UI text.
EventSystem	An object which is required to be in the scene in order for Unity's canvas object to work properly.
Main Camera	The game object which has a camera component attached to it to render the game view to the user. This object also contains a CameraController component which allows the object (and therefore the camera) to move around using user input.
Directional Light	An object with a light component used to brighten up the scene.
Terrain	The object which generates the terrain and renders it onto a mesh renderer (making it visible on screen). It also contains a grid component which is used for pathfinding on the generated terrain.

Listed under the appendix is a set of images which show in depth which components each object consists of.

Game objects which are placed in the initial game scene

Other objects such as the player base buildings and units are created at runtime on demand. These objects include:

Name	**Description**
Base	The base object is created at the start of runtime for each player. This object contains a BaseController component.
Bullet	The bullet object is created when a unit is firing. This object contains a ProjectileController component.
Shell	A projectile shell object is also created when a unit is firing. This is ejected from the side of the gun as a visual effect. This object contains a Shell component.
Soldier	The soldier object is a type of unit in the game. This object contains a UnitController component.
Sniper	These behave the same as the soldier object, except for having different graphics and different values set in the UnitController component for range, damage, accuracy, attack speed, and movement speed.
Shotgun	
Machinegun	



Prefab objects which can be instantiated at runtime

Components

As mentioned before, many of the objects use components which have been specifically written for the game. Each of these components have a specific task which they responsible for. These components are:

Name	**Description**
AIPlayer	Runs the AI logic for the enemy player. This includes determining which/when units spawn and where they move.
CameraController	Takes in user input to move the camera object around the map, including zooming in and out on the map.
GameUI	Handles button presses which occur on the UI and updates the UI text.
TerrainRenderer	Given a renderer and collider component, it will draw the given mesh onto the components. This is called from the TerrainGenerator component after the terrain is generated in order to render the terrain onto the scene.
TerrainGenerator	Responsible for procedurally generating the terrain. This component assumes that there is also a TerrainRenderer and Grid component, so that it can render the generated terrain, and a grid can be constructed for the terrain for pathfinding.

Grid	This component assumes that there is also a TerrainGenerator component on the same object. It takes the generated terrain from the TerrainGenerator and creates a grid of walkable/unwalkable nodes from it depending on the height of the terrain.
Pathfinder	This component is used as a singleton (mentioned below in further detail) and is responsible for handling all the pathfinding, given a grid.
ProjectileController	Handles the movement of the projectile (bullet) and its collision detection with other objects.
SelectableController	This component is used for objects which have health (i.e. the base building and units) and is responsible for handling their behaviour when taking damage (i.e. destroying them when they reach 0 health). BaseController and UnitController both inherit from this component class.
BaseController	Inherits from the SelectableController component. This behaves similar to the base class except there is an additional method to change the colour of the base (so each player has a different coloured base).
UnitController	Inherits from the SelectableController component. This component has additional functionality to handle movement and shooting of the unit.
Shell	Handles setting initial force of the instantiated shell and destroying the object after a certain amount of time has passed.
UnitSelector	Responsible for drawing the selection box on the screen and calculating which units have been selected. It also handles movement of selected units when the right mouse button is pressed.
ShopManager	This component is used as a singleton and is responsible for handling purchasing of units by the players.
GameManager	This component is used as a singleton and is responsible for handling the current state of the game, as well as other functionality related to the game which does not fit in any other component.

General Design Patterns and Techniques Used

Singletons

The singleton design pattern is used as a part of the game for certain components. Singletons restrict the instantiation of a class to a single object of that class. This is useful because it makes it easy to access the singleton object by calling `ClassName.instance`.

For example, the GameManager singleton mentioned above can be accessed using `GameManager.instance`. This makes it easy to access the object from any other component which may rely on it. This is useful for GameManager since it manages the game state and has many other components calling it. Similarly, for the ShopManager and Pathfinder, it is expected that there will only be one of each of these components in the game, and many other components call upon these two components, so the singleton pattern can be used.

Singletons are implemented by giving the class a static variable for the singleton instance, typically called `instance`, which is then assigned to the current object when it is initialised. For the game, this was done in Unity's lifecycle function, Awake.

```
public static GameManager instance = null;

void Awake() {
    if (instance == null) {
        instance = this;
    }
}
```

Object Pooling

Object pooling is a design pattern which keeps objects initialised in a "pool" so that when they are destroyed, instead of garbage collecting them from memory, they are returned to the pool, ready to be reused. This is done primarily for performance reasons as it can be expensive to continually instantiate new objects (i.e. allocating memory, loading textures and materials), especially when a large number of objects are required and they are used for a short amount of time. Thus, it is faster to keep these objects instantiated and reuse them rather than continually allocating and destroying new objects when they are required.

For this reason, the bullet and shell objects are pooled due to the large amount of these objects required and their short lifespan. The pool class can be found in the ObjectPool.cs file, where a pool of reusable objects is contained in a list. There is a pool for each type of object which is object pooled. These pools are managed by the GameManager component, where they are

initialised. Now, whenever a component requires a pooled object to be instantiated (for example, a shell) rather than calling:

```
Instantiate(shellPrefab);
```

You call:

```
ObjectPool pool = GameManager.instance.GetObjectPool(PoolType.Shell);
pool.Instantiate(shellPrefab);
```

This will either give an object from the pool, or if no objects are available in the pool, a new object will be instantiated normally and given to the user, which will be later returned to the pool when the user no longer needs the object. Using this method allows us to only instantiate as many objects as we realistically need, as they are initially instantiated on demand, but additional objects are only instantiated afterwards if the pool is empty.

Similarly, when you want to destroy a pooled object, rather than calling:

```
Destroy(gameObject);
```

You call:

```
ObjectPool pool = GameManager.instance.GetObjectPool(PoolType.Shell);
pool.Destroy(gameObject);
```

Which will return the object back to the pool.

Coroutines

Unity supports the use of coroutines. Coroutines are similar to regular functions, except that they can have multiple entry points. This means that coroutines can be halted in the middle of the function and continued later on from where it left off. This allows coroutines to execute across multiple frames.

Typically when a normal function is executed, the whole function run in the frame it is executed in until the end of its execution block. With coroutines, the function can be partially executed and continued later on a different frame. This can be useful in certain situations where you want to wait until the next frame before continuing, or to wait a certain amount of time.

An example of a coroutine used in the game is for generating income for the players. Each player periodically receives income to spend on units until the game is over. This can be easily implemented using a coroutine:

```
private IEnumerator GenerateIncome() {
    while (!gameOver) {
        foreach (Player p in players) {
            p.money += p.income;
        }
        yield return new WaitForSeconds(incomeInterval);
    }
}
```

This increases each player's income every *incomeInterval* seconds while the game is not over.

Raycasting

Raycasting is a technique to determine whether an object is in the way of another object. This is done by projecting a line from the object in a desired direction to see if it hits another object. This can be useful to see if a projectile has hit an object, or to get the height of the terrain a unit is standing on.

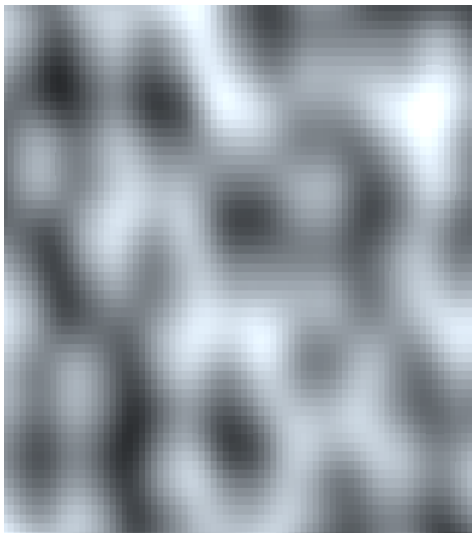
Implementation Details

Procedural Terrain Generation

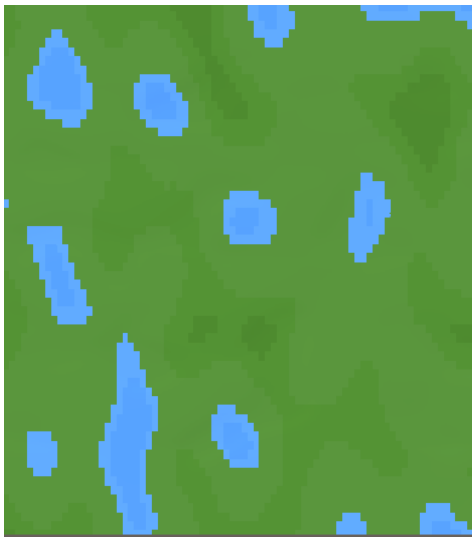
The terrain used in the game is procedurally generated so that the player plays on a new map every game. This terrain is generated using perlin noise.

Unlike regular noise which picks random values for each pixel regardless of the values of surrounding pixels, perlin noise is a type of gradient noise which changes in value gradually such that the value of the pixel depends on the values of its surrounding pixels as well. This smooth transition between values is what allows the terrain to gradually change from puddles of water all the way into hill formations.

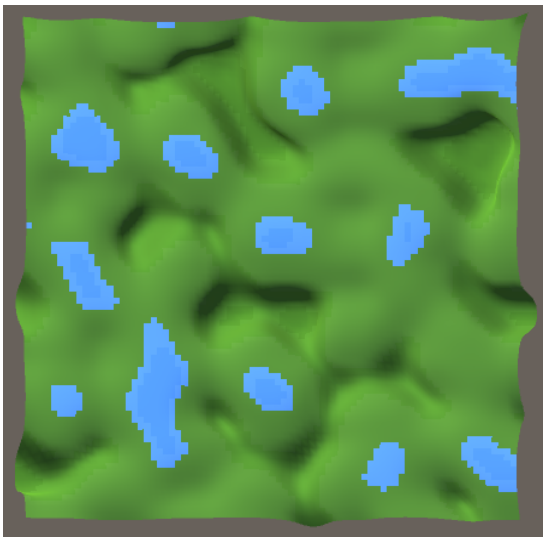
Unity's built-in perlin noise function was used to deterministically gather perlin noise from an x and y point. The perlin noise function returns a value from 0 to 1 which can be interpreted as the height we use for the terrain. Each time the game is played, the initial x and y values used for the perlin noise function is changed by adding a random offset (using a random seed) to the values.



Sample of perlin noise - noise map



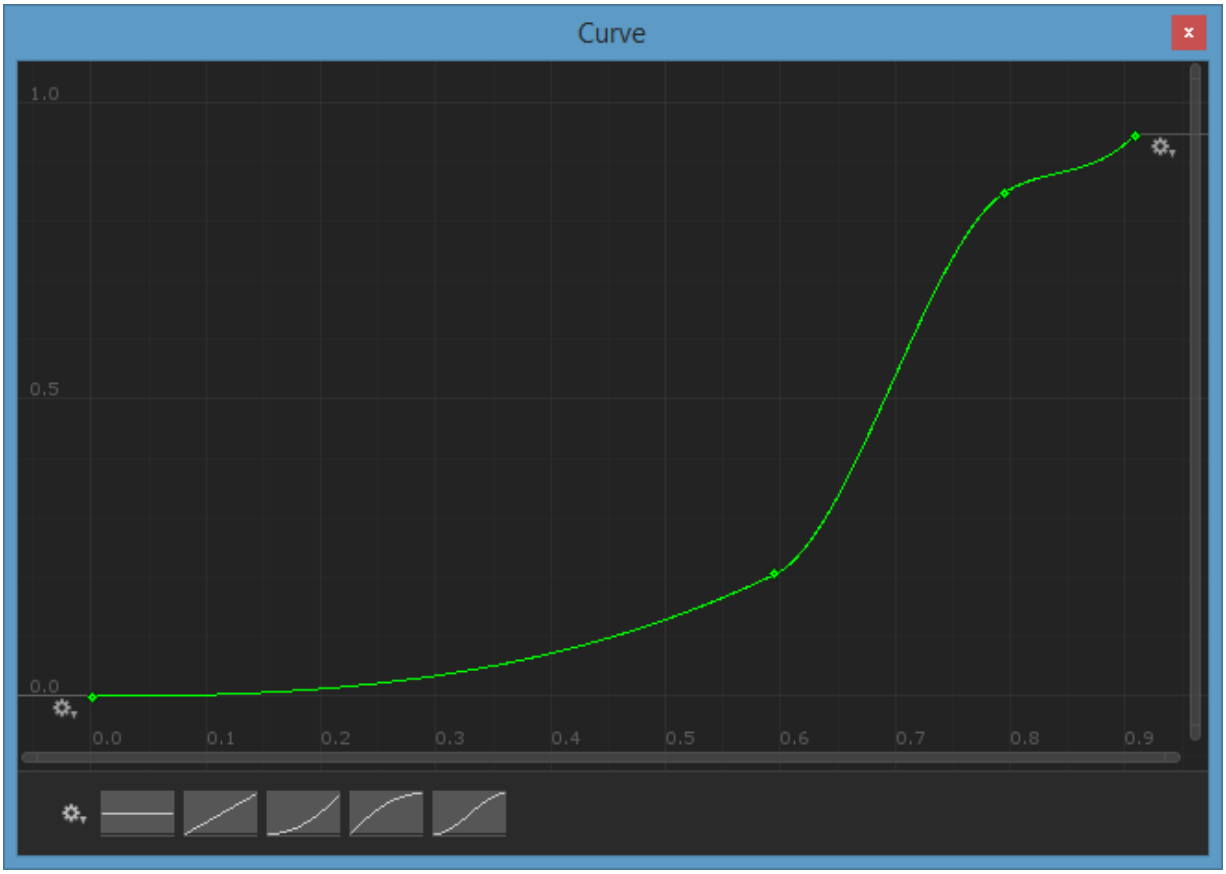
Same sample of perlin noise using our colour map for the terrain



Same sample of perlin noise applied to a mesh which has height

The colour map is generated by converting different ranges of values from the generated perlin noise into different colours. The darker pixels in the noise map image represent low values which get converted into water, and lighter pixels are converted into different types of land.

To generate the mesh with heights, an animation curve (Unity feature) is used.



Animation curve for the terrain heights

This can be seen as a typical function which evaluates the x values (values from the perlin noise) and returns the y values (height) as output. This allows us to control the outputted height for all ranges of the generated perlin noise. In addition to this, the evaluated value is then multiplied by a height multipler (`meshHeightMultiplier`) to further increase the height of the terrain.

Enhancements

To make further enhancements to our perlin noise, we can sum up multiple passes (octaves) of the perlin noise to produce terrain with more detail. However, to do so we must introduce two new variables:

- Lacunarity: affects the frequency of each subsequent octave
    - Controls the amount of small details in the terrain
- Persistence: affects the amplitude of each subsequent octave (value 0 to 1)
    - Controls how much each subsequent octave affects the perlin noise

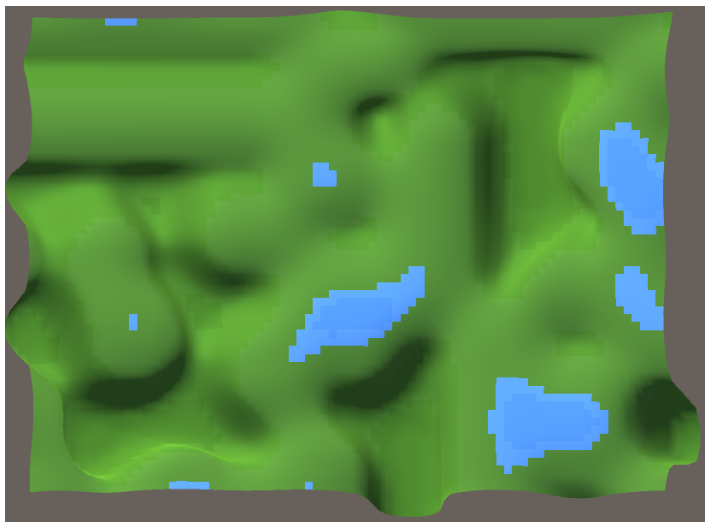
After each iteration of the octaves, the frequency and amplitude values are multiplied by the lacunarity and persistence respectively. This changes how much the remaining octaves affect the perlin noise.

We can then specify the number of octaves we want to sum up, and for each pass we scale the octave's values according to the frequency (affected by lacunarity) and amplitude (affected by persistence).

```
System.Random prng = new System.Random(seed);
Vector2[] octaveOffsets = new Vector2[octaves];
for (int i = 0; i < octaves; i++) {
    float offsetX = prng.Next(-100000, 100000) + offset.x;
    float offsetY = prng.Next(-100000, 100000) + offset.y;
    octaveOffsets[i] = new Vector2(offsetX, offsetY);
}

float amplitude = 1;
float frequency = 1;
float noiseHeight = 0;

for (int i = 0; i < octaves; i++) {
    float posX = (x - halfWidth) / noiseScale * frequency + octaveOffsets[i].x;
    float posY = (y - halfHeight) / noiseScale * frequency + octaveOffsets[i].y;

    // * 2 - 1 to be able to get negative values
    float perlinValue = Mathf.PerlinNoise(posX, posY) * 2 - 1;
    noiseHeight += perlinValue * amplitude;

    amplitude *= persistence;
    frequency *= lacunarity;
}
```

As shown above, each octave samples a random x and y position multiplied by the frequency and is then scaled down by the amplitude when summing it up with the noiseHeight.

To get a better understanding of how this affects the perlin noise, we can observe the images below, which uses four octaves:

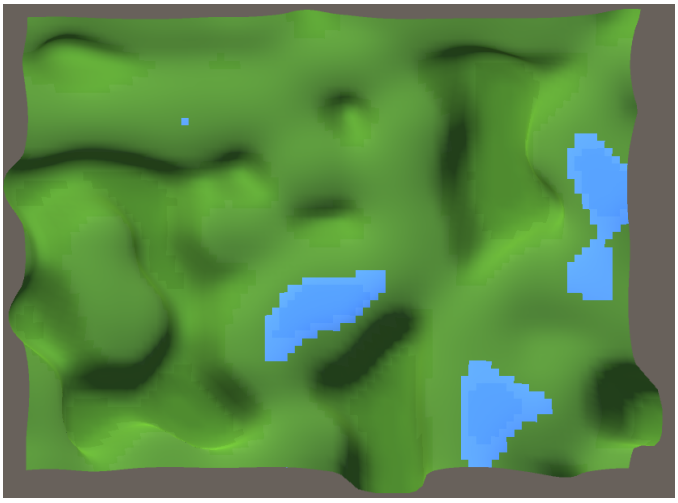


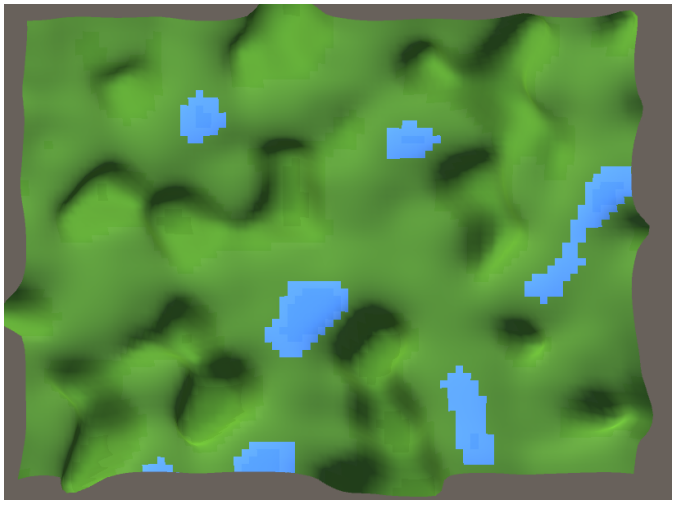
Original terrain

Low Persistence

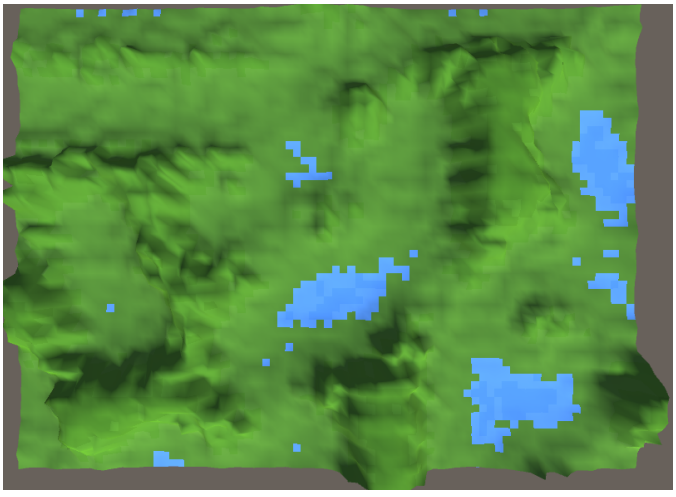
High Persistence

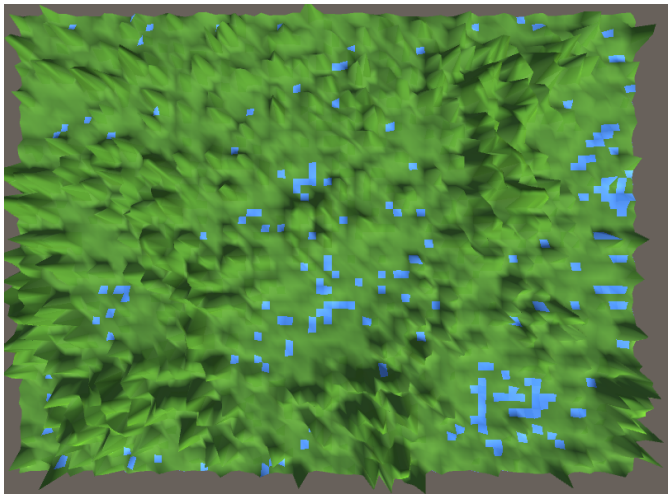
Low Lacunarity





High Lacunarity





As seen above, the terrain with low lacunarity and low persistence looks very similar to the original terrain.
High lacunarity and low persistence keeps the shape of the original terrain while containing much more smaller detail.
Low lacunarity and high persistence gives us a terrain which looks completely different from the original terrain with low amounts of detail.
High lacunarity and high persistence gives us a bizarre terrain which is completely unrecognisable with lots of small details.

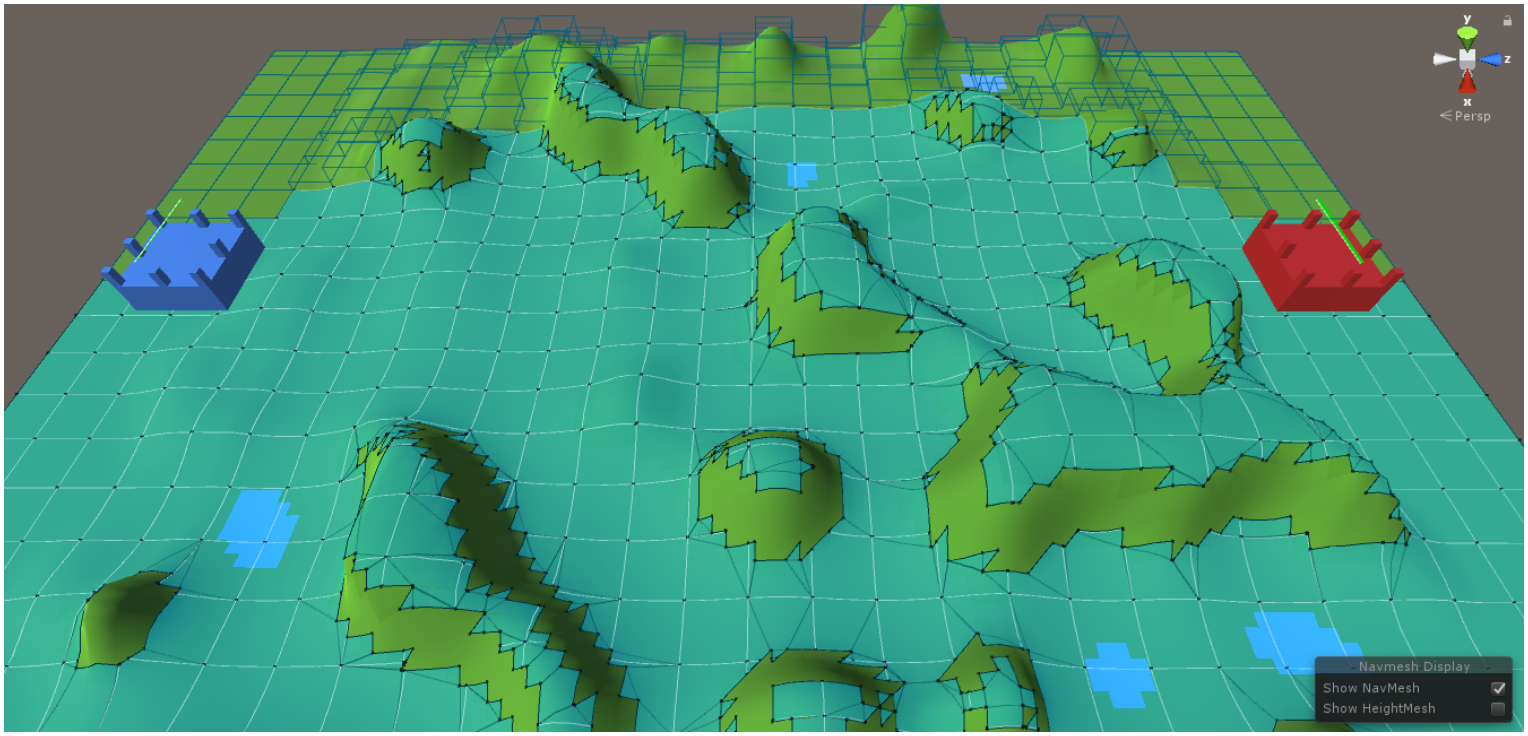
Pathfinding

The pathfinding for the terrain can be done in two different approaches:

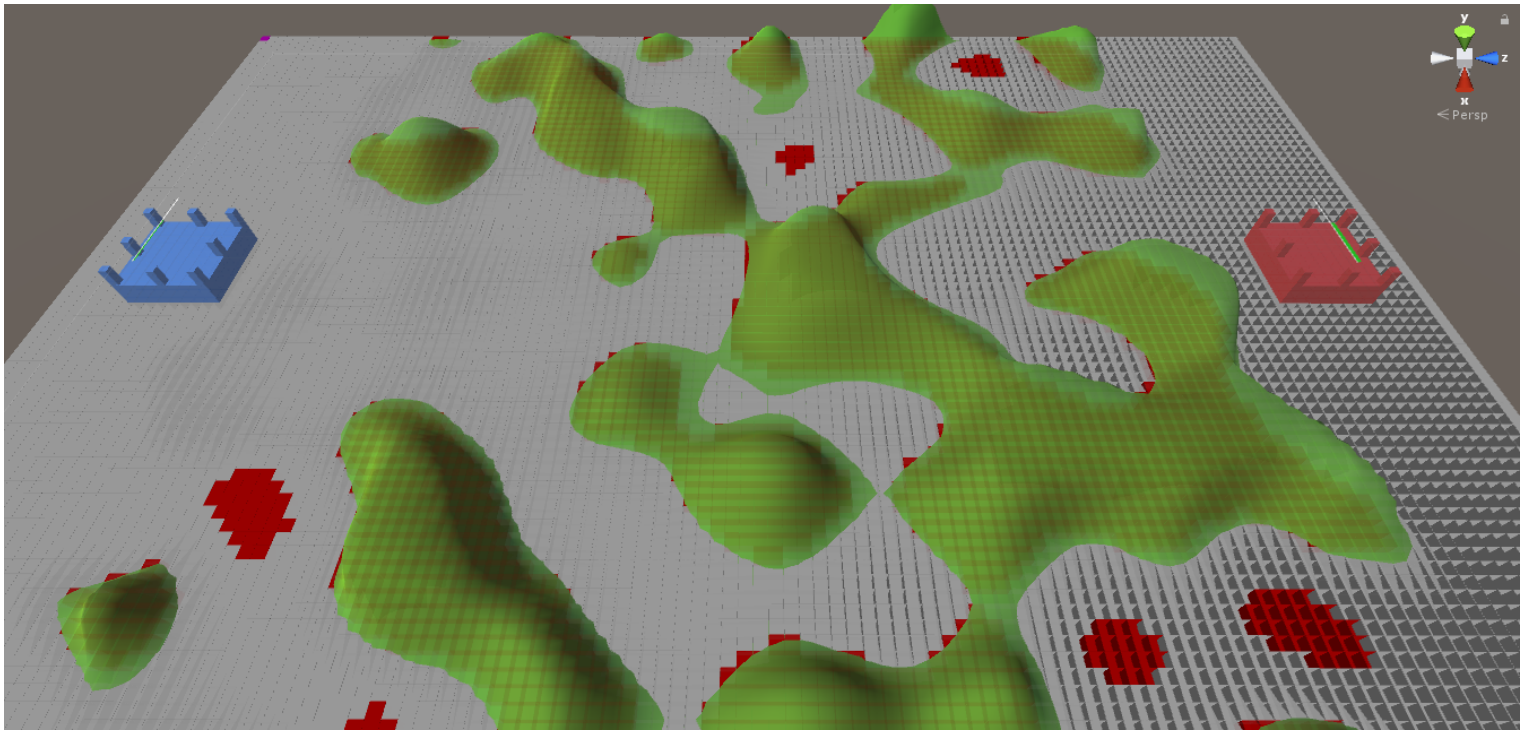
- Grid-based pathfinding
    - Uses a 2D grid system to define which areas are traversable
- Navigation mesh pathfinding
    - Uses a collection of polygons to define areas which are traversable

The most realistic choice between these two was the grid-based pathfinding. Implementing a navigation mesh would involve a much more complicated process which goes beyond the scope of this project (albeit Unity does have support for dynamically generated navigation meshes, it is quite CPU intensive, and also I wanted to write most of the pathfinding myself).

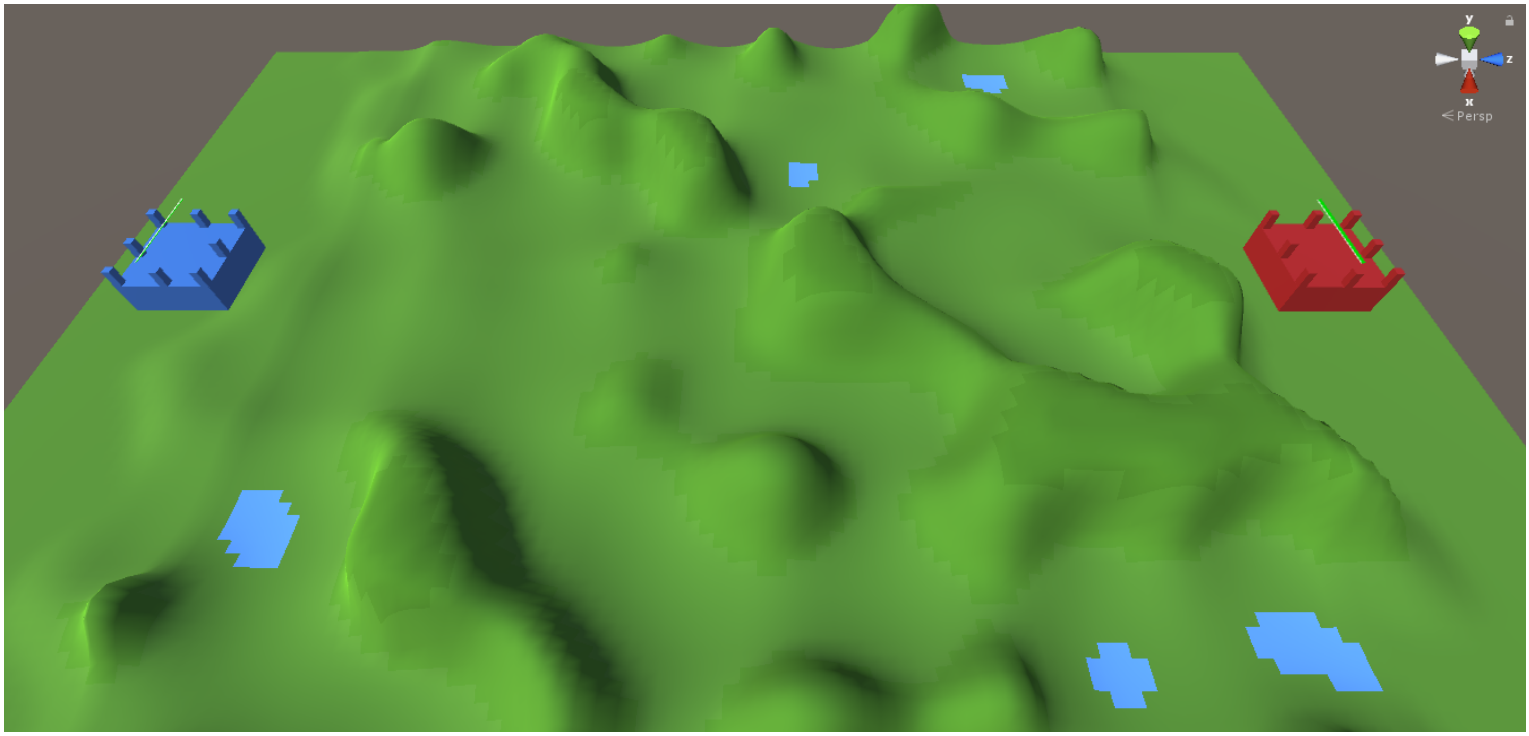
To illustrate the differences between to two approaches, three images are shown below to show Unity's navigation meshes, my 2D grid-based implementation, and the original terrain which these pathfinding methods are basing their pathfinding on. It should be noted that the visualisation of the pathfinding shown in the images below is only seen during development for debugging purposes and can be created in Unity via code (in the OnDrawGizmos lifecycle function).



Navigation mesh generated by Unity (teal polygons = walkable area)



My 2D grid-based implementation (white = walkable, red = non-walkable) - looking carefully you can see the areas under the hills are all unwalkable (red), as expected from hills



The original terrain which the two above pathfinding methods are basing their pathfinding on

As mentioned before, the grid system used for pathfinding is created in the Grid component. The Grid component requires knowledge of the TerrainGenerator component so that it can get the heights of the terrain at each position and create the grid using them.

The grid is created by making a 2D array of nodes. As shown in the image above with the 2D grid system, each node is marked as either walkable or unwalkable depending on the height of the terrain at that position. This makes it simple to prevent walking on water or over hills, since the height would be either too low or too high.

Once the grid is created, pathfinding can be performed using the A* algorithm to find the path between two points on the grid. The actual pathfinding is done in the Pathfinder component. The path returned by the pathfinder is a list of node positions on the map which should be traversed by the object to reach its final destination.

To prevent frame lag when performing multiple pathfinding operations at the same time, an asynchronous pathfinding method has been implemented in the Pathfinder component which is performed on a separate thread and then calls the provided callback function when it has completed.

Since it is possible that the generated terrain may have no path from one base to the other (e.g. covered by water), a simple path find is performed after the grid is initially created to see if there is a path from one base to the other. If there is not, a new map is generated until a map is found which is not blocked between bases.

AI Player

The AI logic is attached to the GameManager object as a component.

The AI uses a simple algorithm to play the game. It chooses a random unit to buy and waits until it has enough money for it. Once it purchases the unit, it moves the unit to a random position near its base and then makes its way towards the opponent's base, where it either dies to enemy units or takes down the opponent's base.

A coroutine is used to run the AI's logic as it allows the AI to pause for a random amount of time in order to emulate a real person's behaviour. This is done at certain parts of the coroutine by calling:

```
yield return new WaitForSeconds(Random.Range(1, 3));
```

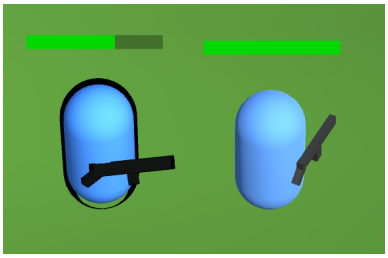
To make it more challenging for the user, the AI receives additional income to increase the number of units it can spawn, making the user having to decide on more tactical ways to approach the game, such as hiding behind hills or using range advantages.

Unit Selection

As previously mentioned, unit selection behaviour occurs in the UnitSelector component which is a component of the GameManager object.

Selection Shader

When a unit is selected, the unit's material shader is changed from the standard shader into an outline shader (defined in the CustomOutline.shader file) as shown in the image below. This is used as an indicator that the unit is selected. Deselecting the unit changes the shader back to the standard shader.



Outline shader shown on the left, standard shader shown on the right

Selection Box

When creating the unit selection box (shown in the image below) to select units, to know whether a unit is within the bounds of our selection box, we have to calculate whether the world coordinates of the unit are within the viewport coordinates of our screen.

The viewport coordinates are the screen coordinates normalized to an area of (0, 0) to (1, 1), where (0, 0) represents the bottom-left and (1, 1) represents the top-right of the screen.

The world coordinates are the coordinates of the object inside the 3D space of the game world.

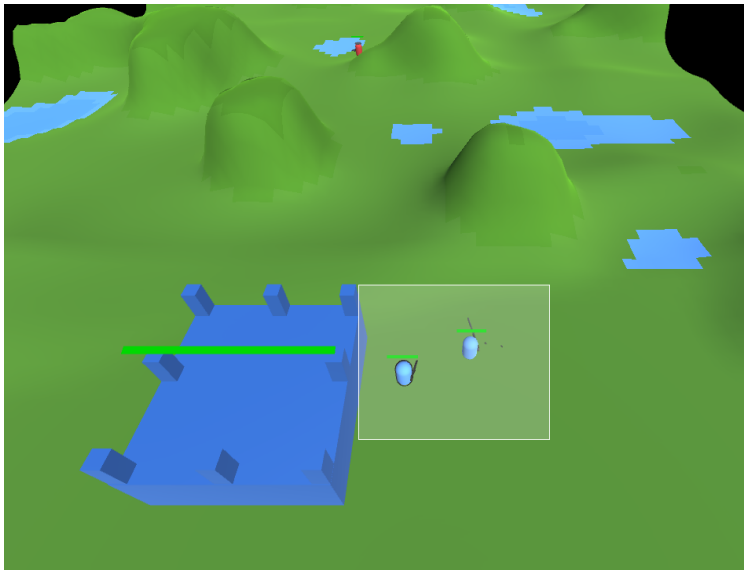
The screen coordinates are similar to the viewport coordinates except rather than going from (0, 0) to (1, 1), it goes from (0, 0) to (screen width, screen height).

Thus, to determine whether a unit is within our selection box, we must go from screen coordinates (since the selection box coordinates are in screen coordinates) to viewport coordinates, and similarly convert the unit's world coordinates into viewport coordinates.

Thankfully, to convert between these coordinate systems, we can utilise Unity's built-in functions:

- `Camera.ScreenToViewportPoint`
- `Camera.WorldToViewportPoint`
- `Bounds.Contains`

We then just have to do this calculation for each unit to see if the unit is within the selection box.



Unit selection box shown selecting two units

Unit Movement

Units which are selected can walk around the map towards a destination. When a destination position is set for a selected unit, the Pathfinder component is invoked to calculate the best path for the unit. This is done asynchronously using a different thread to reduce the amount of frame lag.

When moving towards a destination, the unit follows the path provided by the Pathfinder class. However, the path provided by the pathfinder only tells us the x and z coordinates to head towards. In order to determine the height the unit should be placed at, a raycast is performed down onto the terrain from above to determine where the ground is. This allows us to always keep the unit on the same level as the terrain.

Unit Shooting

Finding Nearby Enemy Units

To find other units within range of the unit's attack range, `Physics.OverlapSphere` is called whenever the unit tries to find a target to shoot. This function finds all the units within the overlapping sphere around the unit. The closet unit within the sphere from the opposing team is then selected as the target.

Projectile Randomness

When a unit fires a projectile, it is sent in the direction of the target. However, it does not go directly towards the target. Each unit type has an accuracy value which affects the accuracy of its shots. This accuracy randomness is applied to each projectile to simulate the inaccuracy of firing a gun.

Inside the ProjectileController component, when a new target is set, the direction is set as:

```
dir = (value - transform.position).normalized;
dir = (dir * 100 + Random.insideUnitSphere * shotRandomness).normalized;
```

Scaling the original direction by 100 allows us to keep the initial direction relatively the same while adding a bit of randomness to the shot by adding a slight change in the direction using `Random.insideUnitSphere * shotRandomness`.

Collision Detection

Due to the fact that projectiles are so small and travel at such high speeds, it is possible that a projectile may travel past an object it was supposed to hit without any collision detection triggering. This is because it may have gone through the object in a single frame such that it never actually *collided* with the object. For example, going from in front of an object to behind an object in a single frame means that in no frame was the projectile actually inside the object, which would be the event which triggers the collision detection.

To prevent this, we use raycasting to cast a ray in the direction that the projectile is travelling in. The ray casted would be equal to the distance the projectile is meant to travel in that frame. This allows us to find any objects in the projectile's path, rather than just updating the projectile's position each frame and checking if it is colliding with anything.

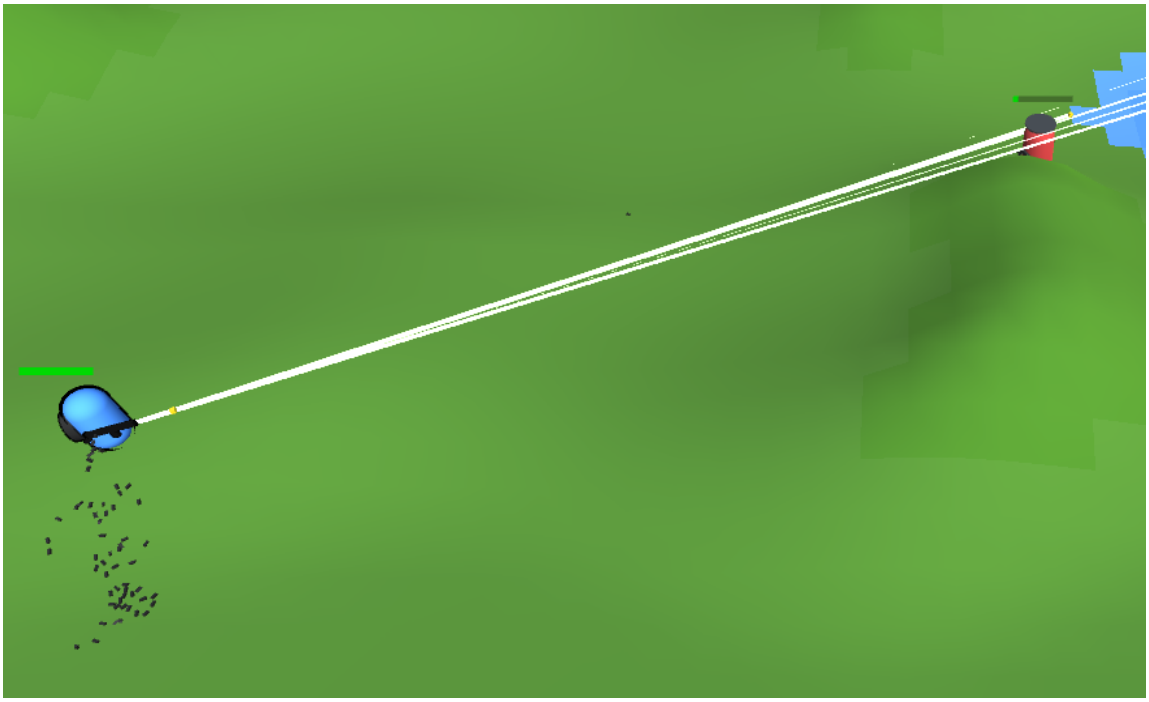
Visual Effects

To improve the visuals of the game, whenever a projectile is shot:

- A white trail of the projectile is shown along its travel path
- A projectile shell object is created next to the firing unit

The projectile trail is implemented by adding a trail renderer component (which is built into Unity) to the projectile. This component creates a trail which follows the game object as it moves around the world.

The projectile shell objects contain a rigidbody component which allow them to interact with the terrain's mesh collider to simulate physics as they hit the ground, giving them a more realistic effect. To prevent an overload of shells on the screen, they are removed after a few seconds have passed.



White trails from the projectiles firing can be seen, as well as shells next to the blue unit each time a projectile is fired

Camera Zoom

To handle zooming in and out of the camera, two different methods were considered:

1. Changing the position of camera such that it is physically closer/further away
2. Changing the field of view of the camera while keeping it stationary, which makes it appear closer/further away

In the end, changing the field of view was the chosen method. This was partly due to a possible issue with moving the camera, where it may accidentally clip through the terrain if we zoom in too much. However, this is not an issue when using the field of view method since the camera is stationary.

Game Over

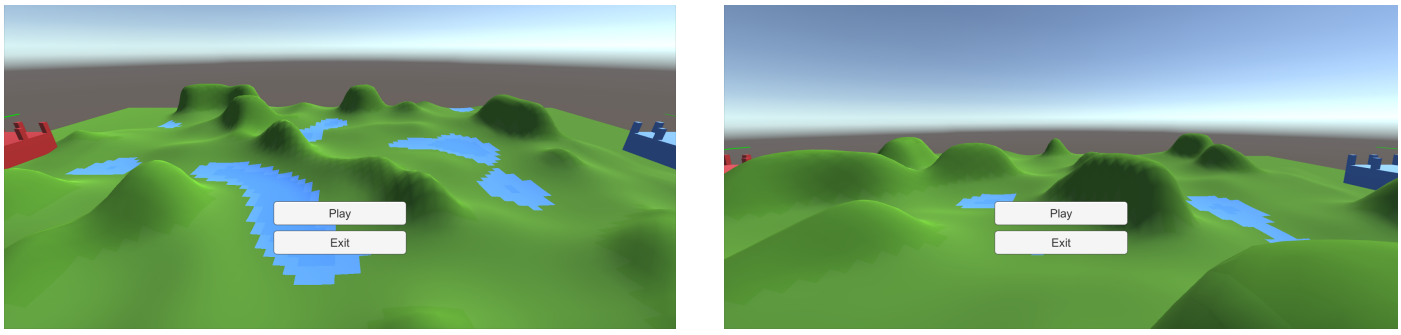
As mentioned before, the state of the game is controlled by the GameManager component. To determine whether the game is over, every frame, the GameManager checks whether one of the bases has been destroyed. When this happens, it calls the `onGameOver` event, which other components can subscribe to in order to know when the game has ended. In this instance, the event is subscribed to by the GameUI component which uses it trigger a function to display the game over screen.

Game over screen

Main Menu

To make for an interesting main menu screen, the background of the menu screen is the procedurally generated terrain which is used within the game. The camera on the menu continually moves up and down smoothly using a coroutine. A coroutine is used for this so that the movement can be easily halted for a number of seconds (using `yield return new WaitForSeconds(n)`) before continuing to move the camera.

The menu is controlled using the MenuManager component which handles the movement of the camera and listening to button clicks.



Main menu screen showing different terrains at different angles

Conclusion

Overall, the game turned out well and was in a playable state by the end of development. A functional game has been produced as a result, comprising of procedurally generated terrain, pathfinding, AI logic, as well as featuring additional visual effects such as the projectile shells and projectile trails.

The most interesting aspects while developing the game were implementing the procedurally generated terrain, as well as looking at the different approaches to pathfinding for a 3D game.

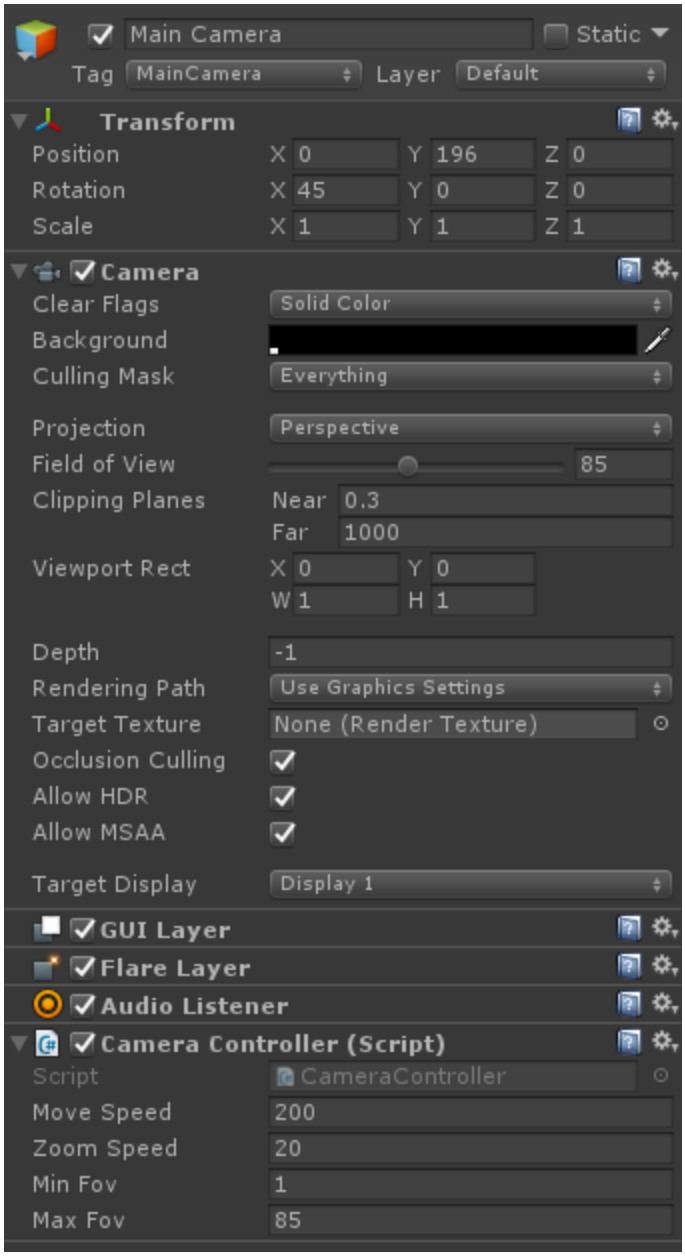
Future Works

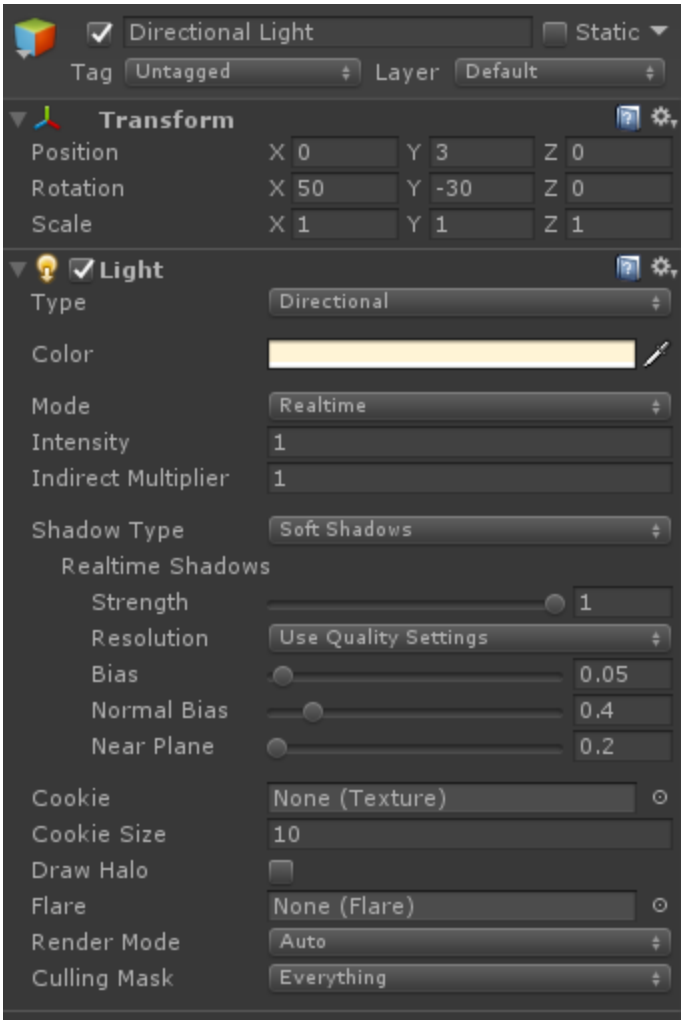
Some ideas for future development of the game which could be implemented include:

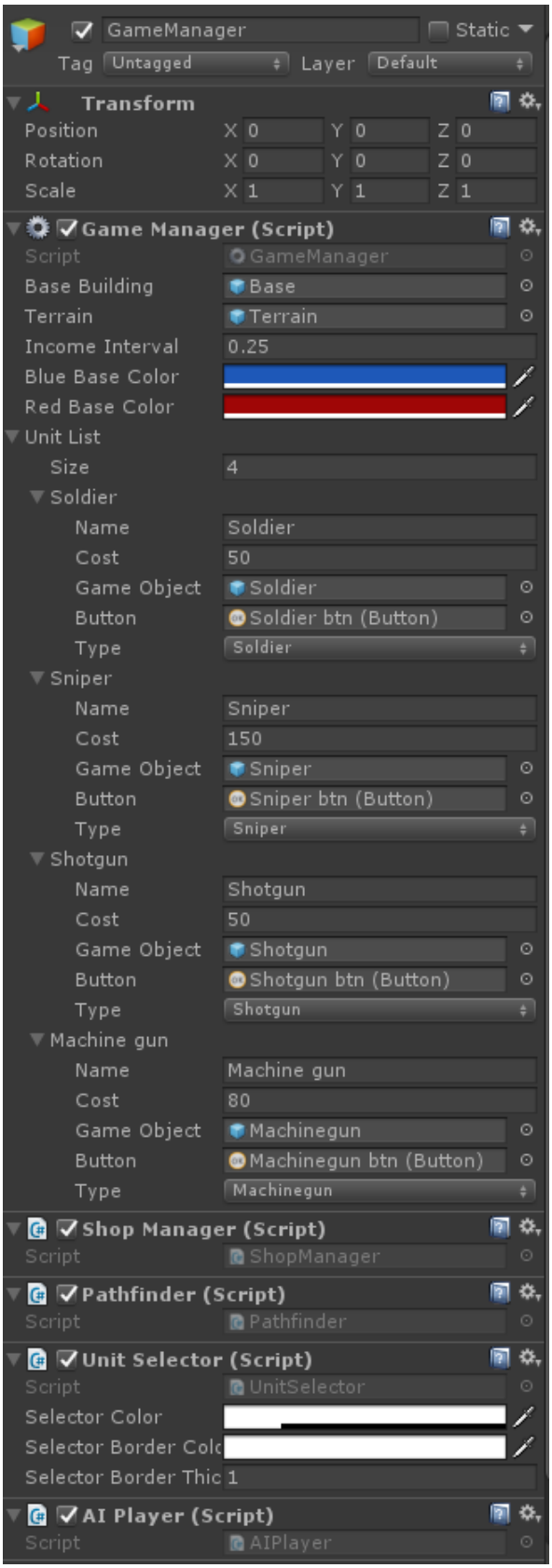
- Implementing boids / flocking behaviour for unit movement (so that units do not clump all together into the same spot)
- Implementing a custom navigation mesh for pathfinding from scratch

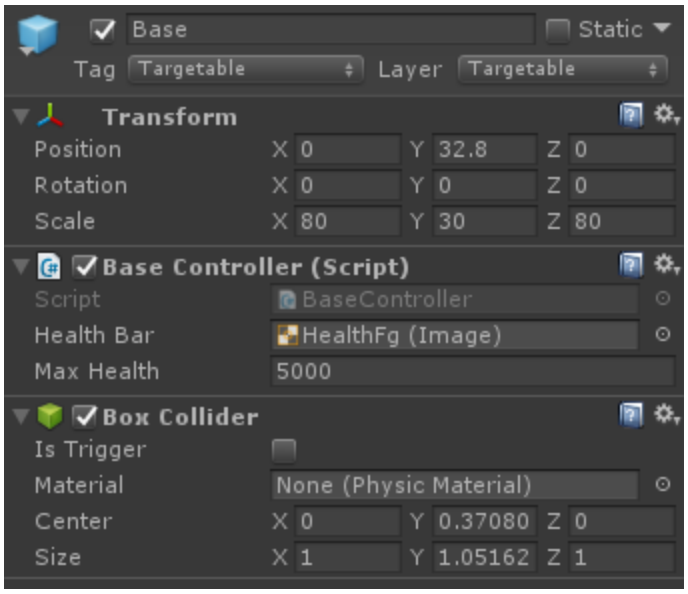
Appendix

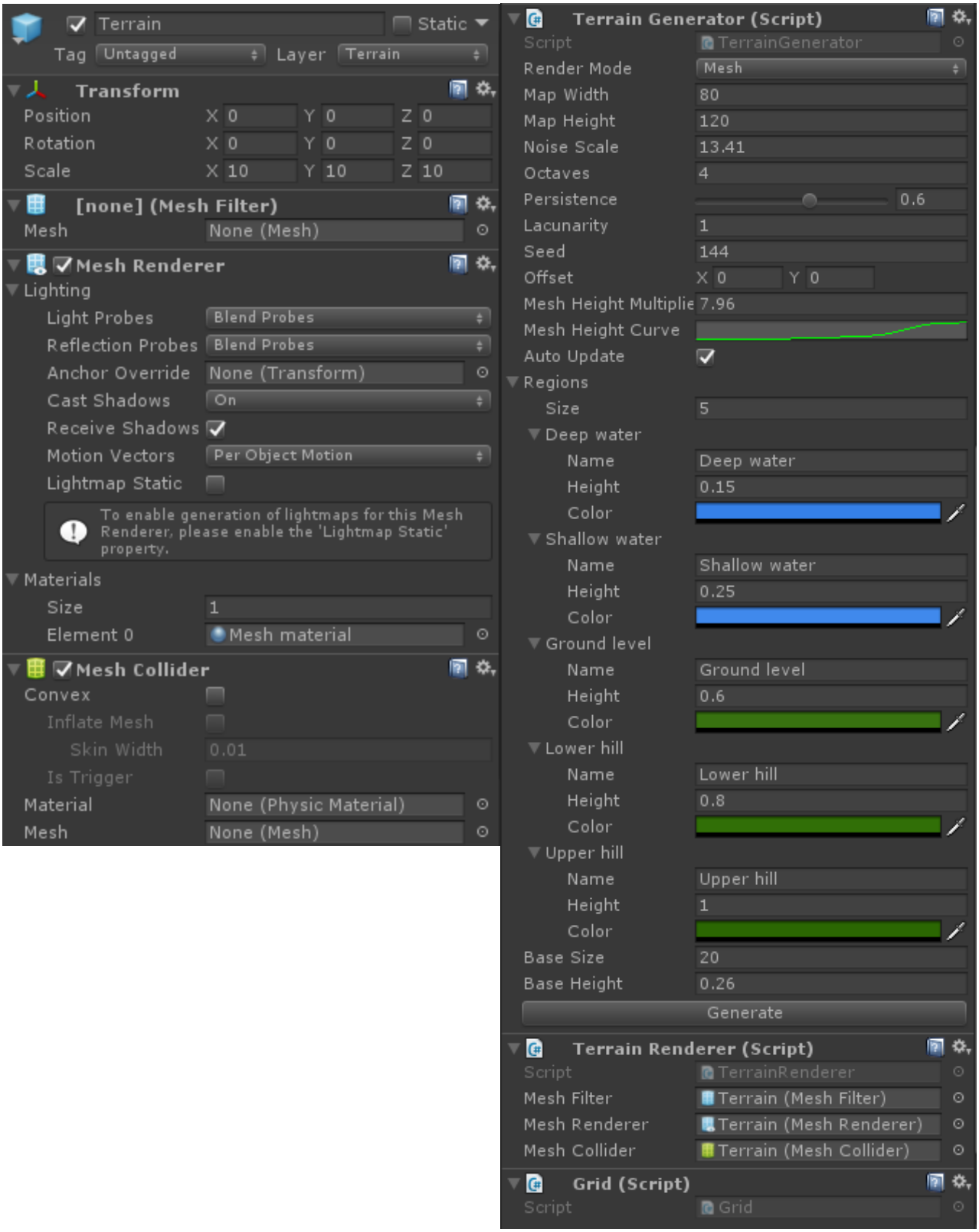
Shown below are some more detailed images of each game object's components. The name of each game object can be seen at the top of the image.

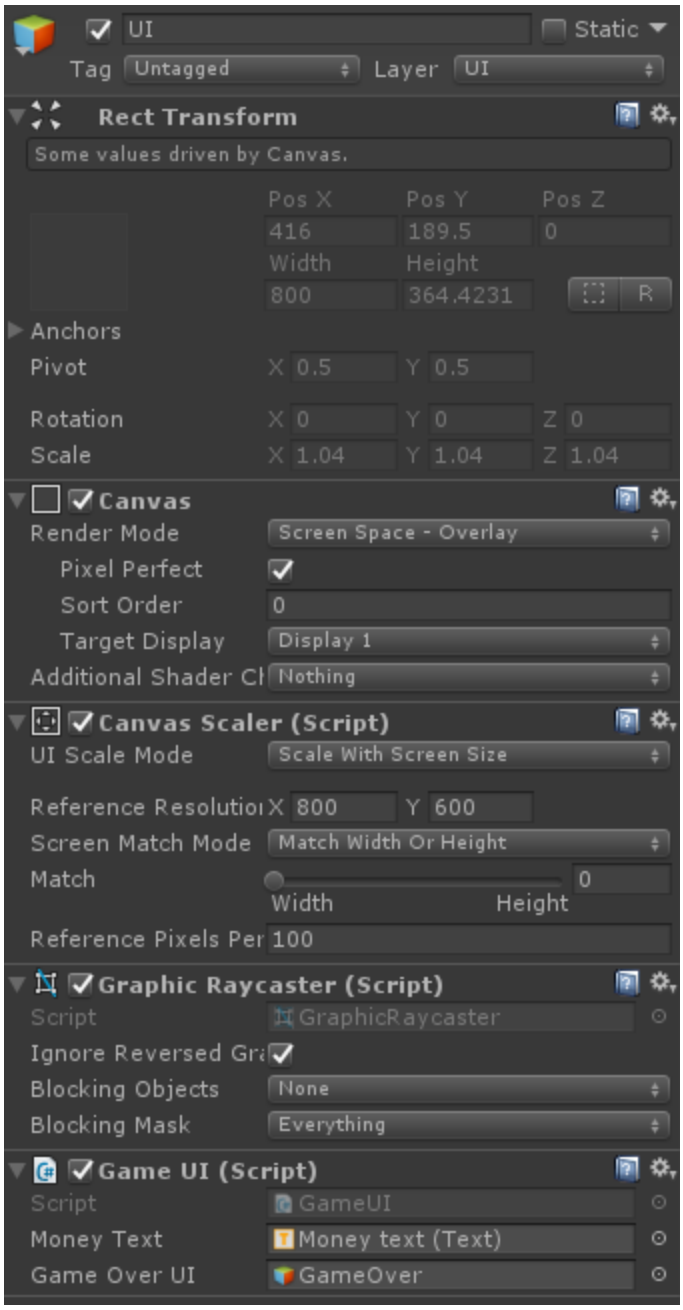


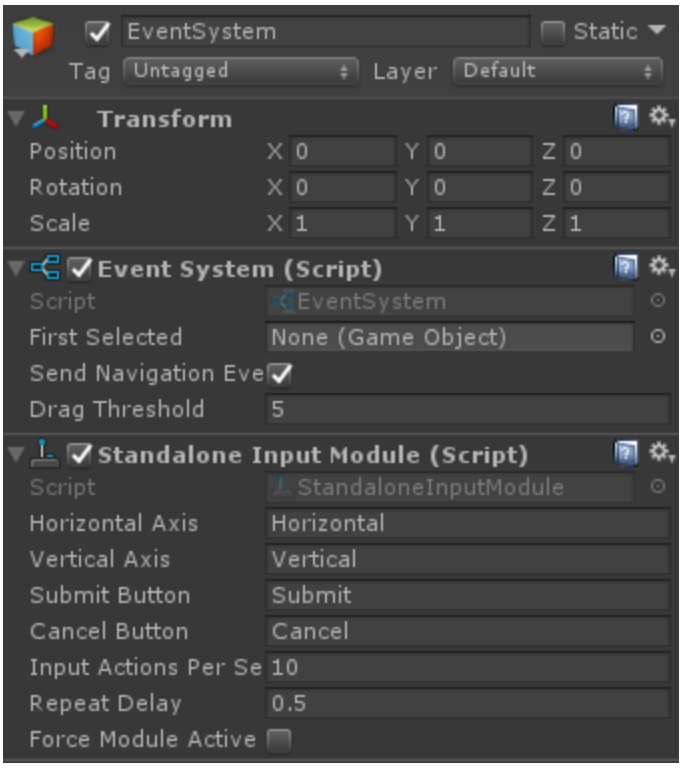


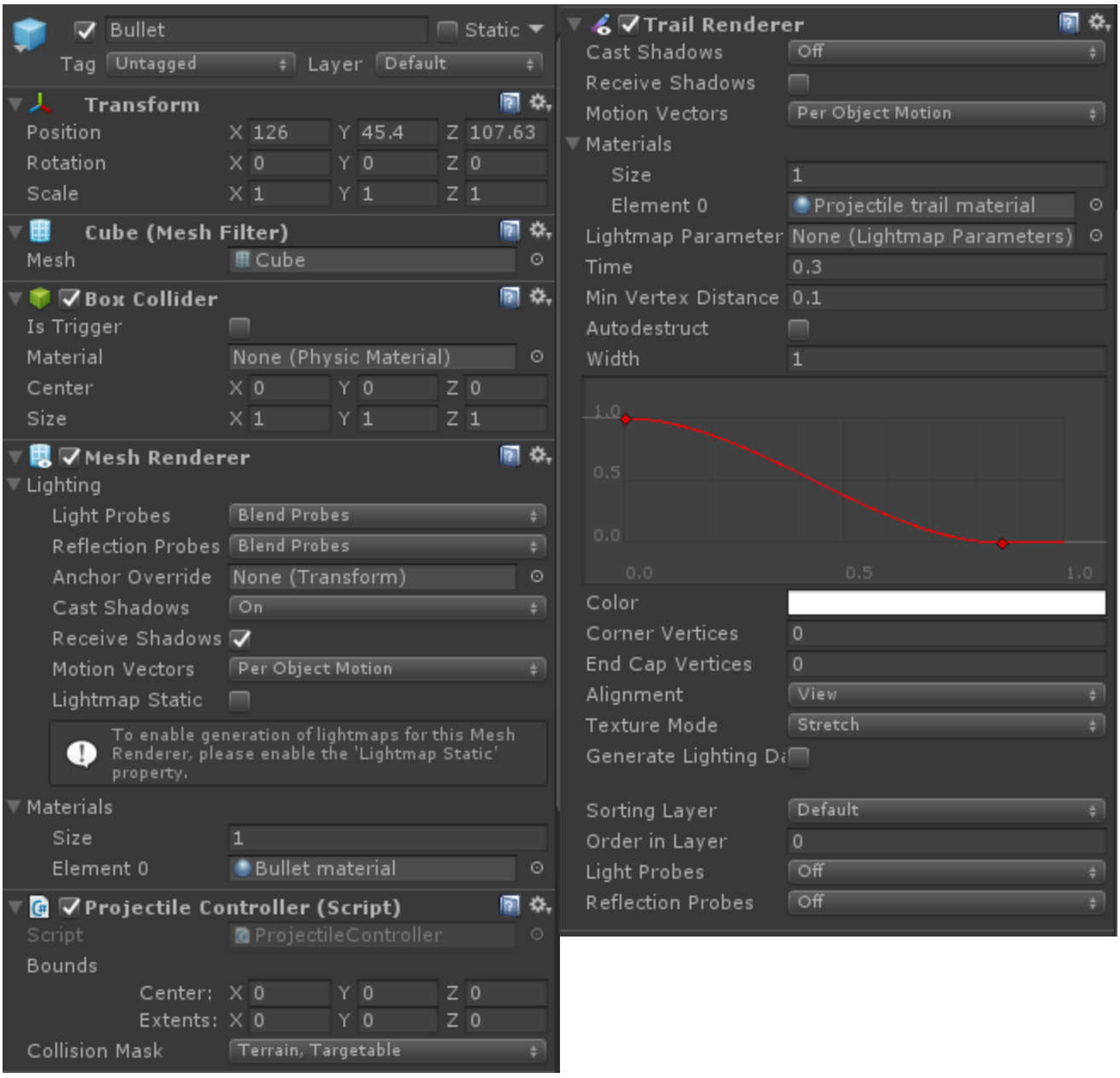

GameManager
Static
Tag Untagged
Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Game Manager (Script)
Script GameManager
Base Building Base
Terrain Terrain
Income Interval 0.25
Blue Base Color
Red Base Color
Unit List
Size 4
Soldier
Name Soldier
Cost 50
Game Object Soldier
Button Soldier btn (Button)
Type Soldier
Sniper
Name Sniper
Cost 150
Game Object Sniper
Button Sniper btn (Button)
Type Sniper
Shotgun
Name Shotgun
Cost 50
Game Object Shotgun
Button Shotgun btn (Button)
Type Shotgun
Machine gun
Name Machine gun
Cost 80
Game Object Machinegun
Button Machinegun btn (Button)
Type Machinegun
Shop Manager (Script)
Script ShopManager
Pathfinder (Script)
Script Pathfinder
Unit Selector (Script)
Script UnitSelector
Selector Color
Selector Border Colc
Selector Border Thic 1
AI Player (Script)
Script AIPlayer

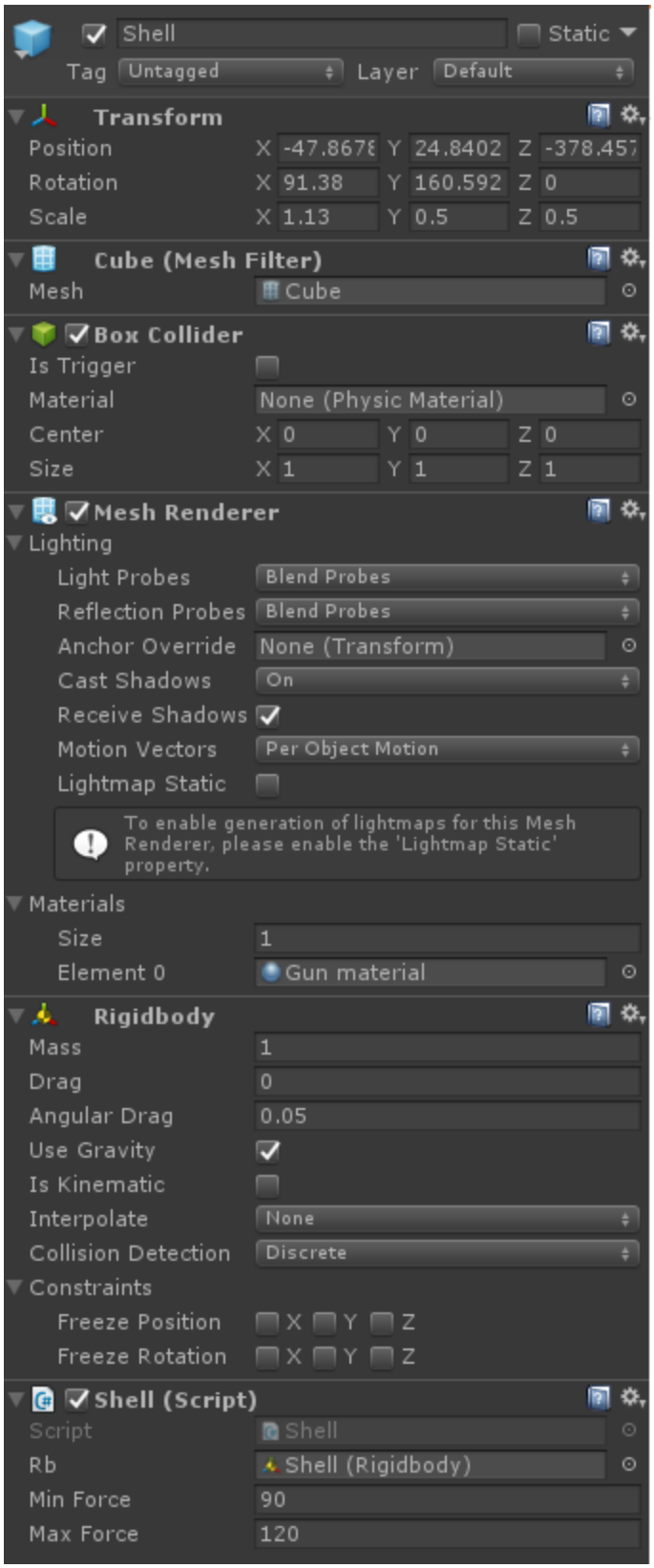

Base
Static
Tag Targetable
Layer Targetable
Transform
Position X 0 Y 32.8 Z 0
Rotation X 0 Y 0 Z 0
Scale X 80 Y 30 Z 80
Base Controller (Script)
Script BaseController
Health Bar HealthFg (Image)
Max Health 5000
Box Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0.37080 Z 0
Size X 1 Y 1.05162 Z 1

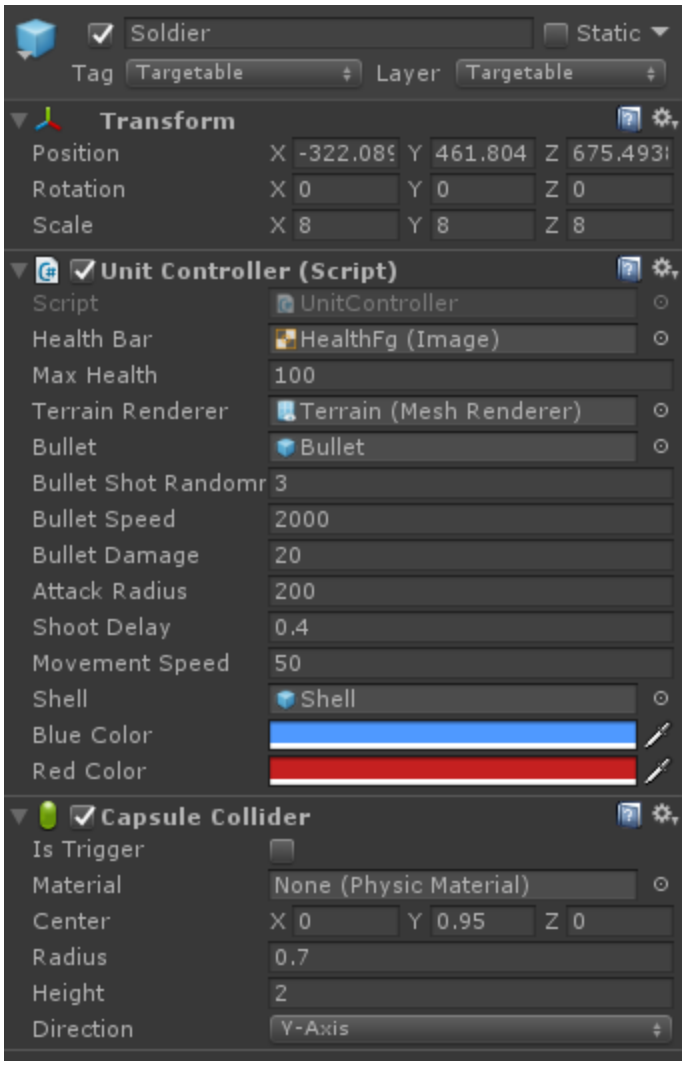

Terrain
Static
Tag Untagged
Layer Terrain
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 10 Y 10 Z 10
[none] (Mesh Filter)
Mesh None (Mesh)
Mesh Renderer
Lighting
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)
Cast Shadows On
Receive Shadows
Motion Vectors Per Object Motion
Lightmap Static
To enable generation of lightmaps for this Mesh Renderer, please enable the 'Lightmap Static' property.
Materials
Size 1
Element 0 Mesh material
Mesh Collider
Convex
Inflate Mesh
Skin Width 0.01
Is Trigger
Material None (Physic Material)
Mesh None (Mesh)
Terrain Generator (Script)
Script TerrainGenerator
Render Mode Mesh
Map Width 80
Map Height 120
Noise Scale 13.41
Octaves 4
Persistence 0.6
Lacunarity 1
Seed 144
Offset X 0 Y 0
Mesh Height Multiplie 7.96
Mesh Height Curve
Auto Update
Regions
Size 5
Deep water
Name Deep water
Height 0.15
Color
Shallow water
Name Shallow water
Height 0.25
Color
Ground level
Name Ground level
Height 0.6
Color
Lower hill
Name Lower hill
Height 0.8
Color
Upper hill
Name Upper hill
Height 1
Color
Base Size 20
Base Height 0.26
Generate
Terrain Renderer (Script)
Script TerrainRenderer
Mesh Filter Terrain (Mesh Filter)
Mesh Renderer Terrain (Mesh Renderer)
Mesh Collider Terrain (Mesh Collider)
Grid (Script)
Script Grid

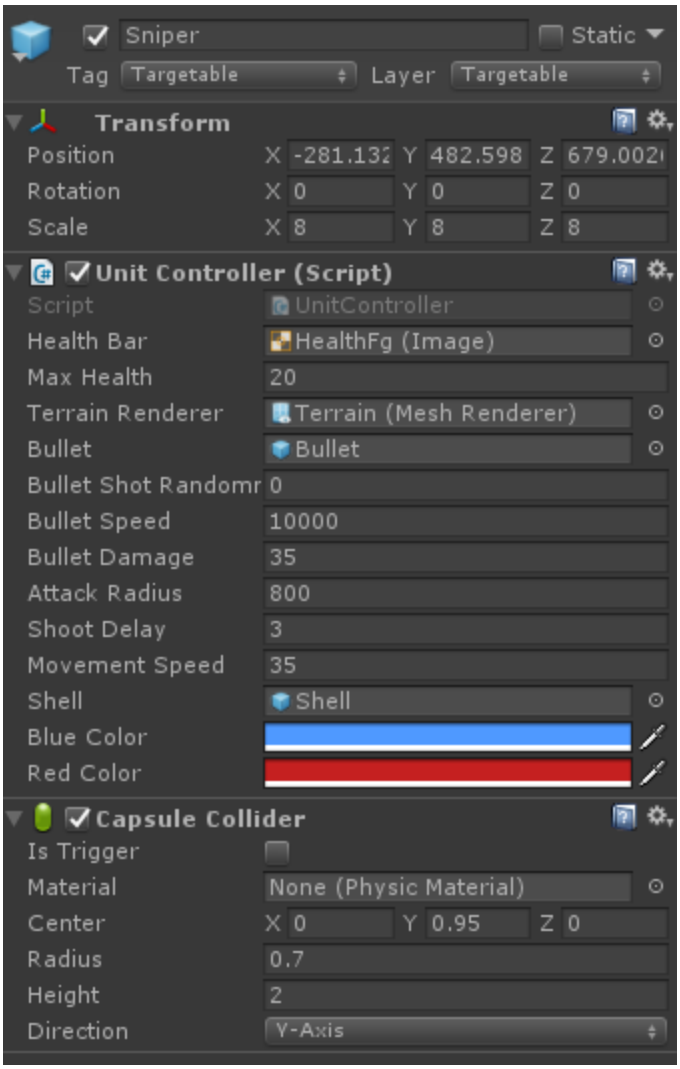

UI
Static
Tag Untagged
Layer UI
Rect Transform
Some values driven by Canvas.
Pos X Pos Y Pos Z
416 189.5 0
Width Height
800 364.4231
R
Anchors
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X 1.04 Y 1.04 Z 1.04
Canvas
Render Mode Screen Space - Overlay
Pixel Perfect
Sort Order 0
Target Display Display 1
Additional Shader Ch Nothing
Canvas Scaler (Script)
UI Scale Mode Scale With Screen Size
Reference Resolutio X 800 Y 600
Screen Match Mode Match Width Or Height
Match 0
Width Height
Reference Pixels Per 100
Graphic Raycaster (Script)
Script GraphicRaycaster
Ignore Reversed Gra
Blocking Objects None
Blocking Mask Everything
Game UI (Script)
Script GameUI
Money Text Money text (Text)
Game Over UI GameOver

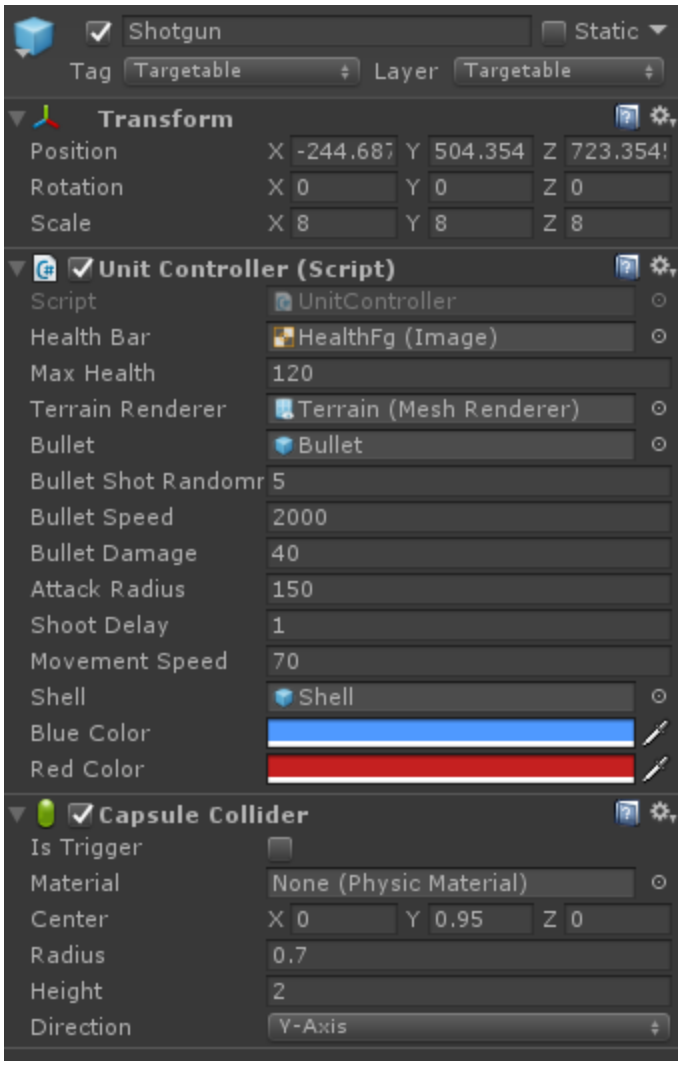

EventSystem
Static
Tag Untagged
Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Event System (Script)
Script EventSystem
First Selected None (Game Object)
Send Navigation Eve
Drag Threshold 5
Standalone Input Module (Script)
Script StandaloneInputModule
Horizontal Axis Horizontal
Vertical Axis Vertical
Submit Button Submit
Cancel Button Cancel
Input Actions Per Se 10
Repeat Delay 0.5
Force Module Active

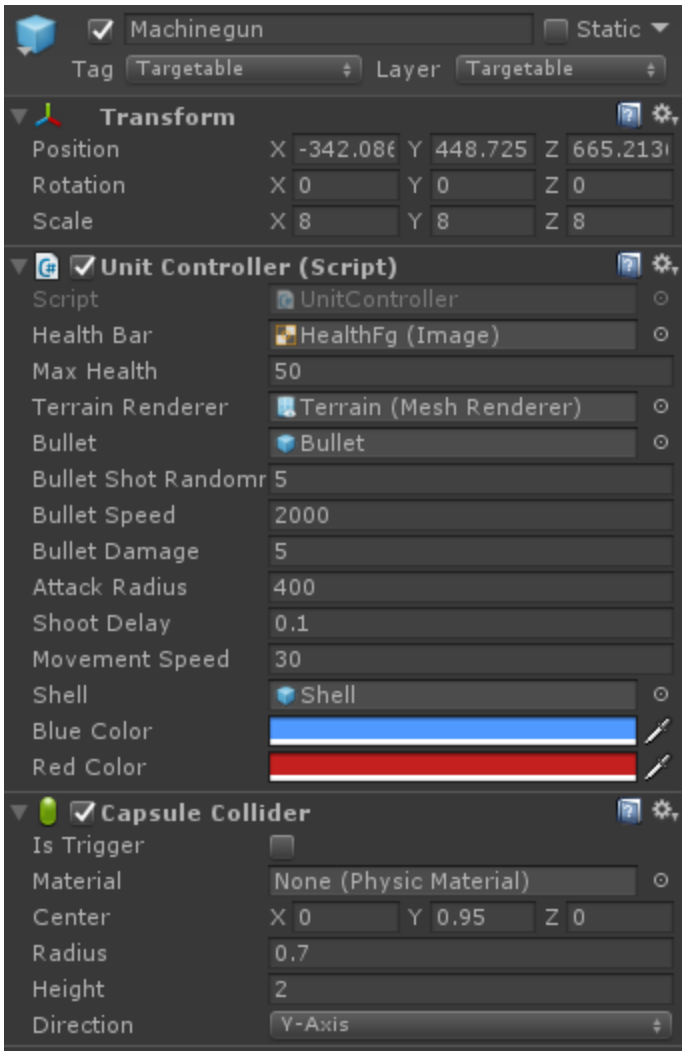

Bullet
Static
Tag Untagged
Layer Default
Transform
Position X 126 Y 45.4 Z 107.63
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Cube (Mesh Filter)
Mesh Cube
Box Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0 Z 0
Size X 1 Y 1 Z 1
Mesh Renderer
Lighting
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)
Cast Shadows On
Receive Shadows
Motion Vectors Per Object Motion
Lightmap Static
To enable generation of lightmaps for this Mesh Renderer, please enable the 'Lightmap Static' property.
Materials
Size 1
Element 0 Bullet material
Projectile Controller (Script)
Script ProjectileController
Bounds
Center: X 0 Y 0 Z 0
Extents: X 0 Y 0 Z 0
Collision Mask Terrain, Targetable
Trail Renderer
Cast Shadows Off
Receive Shadows
Motion Vectors Per Object Motion
Materials
Size 1
Element 0 Projectile trail material
Lightmap Parameter None (Lightmap Parameters)
Time 0.3
Min Vertex Distance 0.1
Autodestruct
Width 1
1.0
0.5
0.0
0.0
0.5
1.0
Color
Corner Vertices 0
End Cap Vertices 0
Alignment View
Texture Mode Stretch
Generate Lighting D
Sorting Layer Default
Order in Layer 0
Light Probes Off
Reflection Probes Off


Shell
Static
Tag Untagged
Layer Default
Transform
Position X -47.8678 Y 24.8402 Z -378.457
Rotation X 91.38 Y 160.592 Z 0
Scale X 1.13 Y 0.5 Z 0.5
Cube (Mesh Filter)
Mesh Cube
Box Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0 Z 0
Size X 1 Y 1 Z 1
Mesh Renderer
Lighting
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)
Cast Shadows On
Receive Shadows
Motion Vectors Per Object Motion
Lightmap Static
To enable generation of lightmaps for this Mesh Renderer, please enable the 'Lightmap Static' property.
Materials
Size 1
Element 0 Gun material
Rigidbody
Mass 1
Drag 0
Angular Drag 0.05
Use Gravity
Is Kinematic
Interpolate None
Collision Detection Discrete
Constraints
Freeze Position X Y Z
Freeze Rotation X Y Z
Shell (Script)
Script Shell
Rb Shell (Rigidbody)
Min Force 90
Max Force 120


Soldier
Static
Tag Targetable
Layer Targetable
Transform
Position X -322.089 Y 461.804 Z 675.493
Rotation X 0 Y 0 Z 0
Scale X 8 Y 8 Z 8
Unit Controller (Script)
Script UnitController
Health Bar HealthFg (Image)
Max Health 100
Terrain Renderer Terrain (Mesh Renderer)
Bullet Bullet
Bullet Shot Randomr 3
Bullet Speed 2000
Bullet Damage 20
Attack Radius 200
Shoot Delay 0.4
Movement Speed 50
Shell Shell
Blue Color
Red Color
Capsule Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0.95 Z 0
Radius 0.7
Height 2
Direction Y-Axis


Sniper
Static
Tag Targetable
Layer Targetable
Transform
Position X -281.132 Y 482.598 Z 679.002
Rotation X 0 Y 0 Z 0
Scale X 8 Y 8 Z 8
Unit Controller (Script)
Script UnitController
Health Bar HealthFg (Image)
Max Health 20
Terrain Renderer Terrain (Mesh Renderer)
Bullet Bullet
Bullet Shot Randomr 0
Bullet Speed 10000
Bullet Damage 35
Attack Radius 800
Shoot Delay 3
Movement Speed 35
Shell Shell
Blue Color
Red Color
Capsule Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0.95 Z 0
Radius 0.7
Height 2
Direction Y-Axis


Shotgun
Static
Tag Targetable
Layer Targetable
Transform
Position X -244.687 Y 504.354 Z 723.354
Rotation X 0 Y 0 Z 0
Scale X 8 Y 8 Z 8
Unit Controller (Script)
Script UnitController
Health Bar HealthFg (Image)
Max Health 120
Terrain Renderer Terrain (Mesh Renderer)
Bullet Bullet
Bullet Shot Randomr 5
Bullet Speed 2000
Bullet Damage 40
Attack Radius 150
Shoot Delay 1
Movement Speed 70
Shell Shell
Blue Color
Red Color
Capsule Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0.95 Z 0
Radius 0.7
Height 2
Direction Y-Axis


Machinegun
Static
Tag Targetable
Layer Targetable
Transform
Position X -342.086 Y 448.725 Z 665.213
Rotation X 0 Y 0 Z 0
Scale X 8 Y 8 Z 8
Unit Controller (Script)
Script UnitController
Health Bar HealthFg (Image)
Max Health 50
Terrain Renderer Terrain (Mesh Renderer)
Bullet Bullet
Bullet Shot Randomr 5
Bullet Speed 2000
Bullet Damage 5
Attack Radius 400
Shoot Delay 0.1
Movement Speed 30
Shell Shell
Blue Color
Red Color
Capsule Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0.95 Z 0
Radius 0.7
Height 2
Direction Y-Axis