AR3

FYP Final Report for

A Turn Based Strategy Game Focusing on Management of Army Logistics

by

HUI Nathan, LEUNG Ho Man Max, LO Yuk Fai, and KONSTANTINO Hubert Aditya

AR3

Advised by

Dr. Sunil ARYA

Submitted in partial fulfillment

of the requirements for COMP 4981/CPEG 4901

in the

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

2021-2022

Date of submission: April 20, 2022

Table of Contents

1. Introduction

1.1. Overview

The gaming industry has experienced colossal growth over the course of the pandemic. COVID-19 has prompted social distancing measures globally, causing many to find alternative ways to interact with their friends, such as through online gaming. The tremendous growth of the gaming industry can be attributed to the surge in need for online socialization, as well as extra time being spent at home with computer access during lockdown.

Currently, the net worth of the gaming industry is valued at around 300 billion US dollars, surpassing the combined worth of two other entertainment titans, movies and music [1]. The most popular gaming titles pay out millions of dollars to their best players while massive audiences spectate. Currently, first person shooter (FPS) and multiplayer online battle arena (MOBA) games dominate the esports scene. The top eight esports by prize pool in 2020 were all FPS and MOBA games [2].

A notable genre missing from the top is strategy games. Real-time strategy games such as Starcraft were extremely popular at the dawn of esports. However, the popularity and development of strategy games have decreased dramatically. In recent years, most strategy game releases have only been sequels based on previous titles.

While strategy games are not the most popular game genre anymore, there remains a large, dedicated following for them. Most strategy games revolve around creating an army to defeat opponents, whether by strategic positioning or brute force. However, these army units in such games never run out of ammunition or food. Napoleon said, “An army marches on its stomach.” Although it is understandable that strategy games are designed without as much focus on the logistics of managing an army to reduce complexity, it can make a game feel unrealistic.

In this project, our team endeavors to create a strategy game with an equal emphasis on combat and positioning, as well as management of army logistics. We feel that this adds a layer of complexity that is not as prominent in other strategy games.

1.2. Objectives

The objective of this project is to create a strategy game with an equal emphasis on combat and positioning, as well as management of army logistics. The project hopes to achieve these goals by:

- **Creating a logistics management system in gameplay**

  The main focus of our project is to incorporate a system that manages a player's army logistics. Thus, the project will create and implement a logistics management system for a player's army that is not overwhelmingly complicated, but adds strategic depth.

- **Creating an attractive strategy game**

  The game should be interesting and fun to play. Ultimately, the purpose of a game is to derive enjoyment and pleasure. An enjoyable new player experience is paramount, so the learning curve should be gradual. Through the design of different systems, the gameplay should reward players with higher strategic skill.

- **Developing multiplayer features**

  Our project hopes to include multiplayer functionality. Pitting player against player generates more fun gameplay.

- **Implementing the above objectives into a smooth game**

  The game should run smoothly, be appealing visually, and have fun gameplay, in order to create an enjoyable user experience.

1.3. Literature Survey

As there are already plenty of strategy games on the market, this Literature Survey will explore and examine the games that most influenced our design and provided inspiration for this project.

Advanced Daisenryaku

Developed by Sega in 1991, Advanced Daisenryaku is a turn-based strategy game that features a series of World War II battles on the European Front. Players play as main combatants in Europe such as Nazi Germany, Great Britain and the Soviet Union.

In the game, players have to control all units they have. It is tedious to micro-manage every unit a player controls, especially during the late game. In light of this, an “auto-pilot” system was adopted for this project to overcome the problem.

Figure 1: Advanced Daisenryaku Gameplay
Source: [3]

Hearts of Iron IV

Released in 2016, Hearts of Iron IV is a real-time grand strategy game developed by Paradox Interactive. Players can play as any nation in World War II. The game focuses on various aspects of the War, including military, diplomacy, economy and espionage.

Figure 2: Hearts of Iron IV Gameplay
Source: [4]

Diplomacy

Diplomacy is a strategy board game developed in 1954 by Allan B. Calhamer. Each player represents one of the seven powers in the times prior to World War I. The game focuses on diplomatic relationships instead of military aspects. Players can win the game by negotiating with other players, or even betraying them.

All players take their turns at the same time instead of following a sequence. This is also known as “simultaneous turns”, which can greatly reduce waiting time of all players in each turn.

Figure 3: Diplomacy
Source: [5]

Starcraft 2

Starcraft 2 is a fast-paced real-time strategy game developed by Blizzard Entertainment. Unlike other games mentioned in this Literature Survey, Starcraft 2 is played in real time. Real-time strategy games are known more for high mechanical skill and intensity, and less on the actual strategies and tactics.

Since the high mechanical intensity of real-time strategy games can take away from the strategic aspect of the game, a turn-based gameplay format was chosen for the project.

Figure 4: Starcraft 2
Source: [6]

2. Methodology

2.1. Design

Our project design consists of 4 main parts: Gameplay, User Interface, Network and Bot Algorithm

Gameplay

Each game will have the same objective: to destroy or capture the enemy's cities. The game will take place on a hexagonal game board.

A sample flowchart for the gameflow of a typical game is shown below.

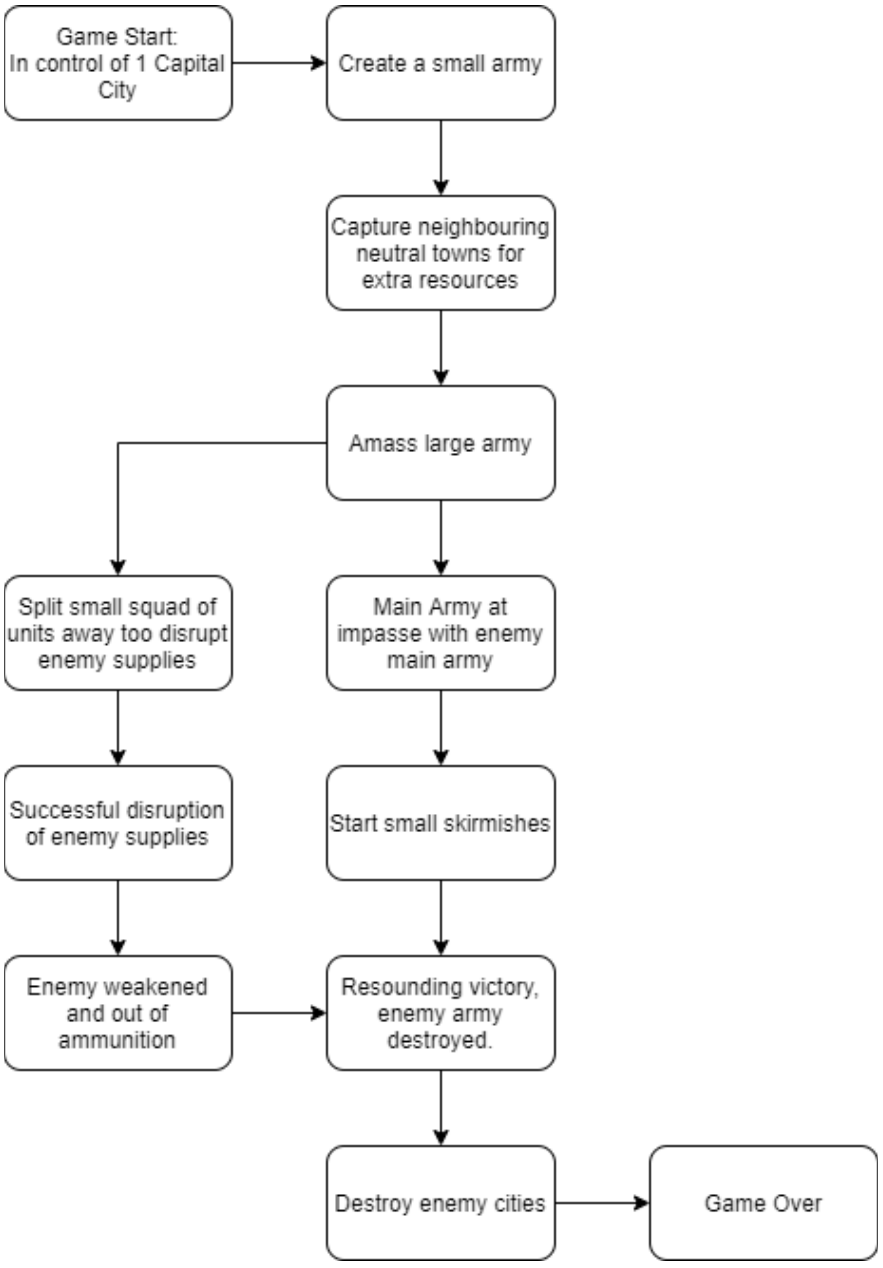


Figure 5: Sample flowchart of a typical game

Our game will also include systems to manage resources, logistics, combat, etc.

Resource Systems

Resources will be automatically generated every turn, based on the amount of cities and villages a player owns.

Logistics Systems

The Logistic System is embedded in the game where each unit of the game carries resources and consumes a certain amount of them each turn. Supplies are consumed by units each turn even if they are holding their position. The further they move, the more supplies they consume. When a unit runs out of supplies, it cannot move until it is re-supplied. The terrains of tiles that the unit traverses also affect its supplies consumption (e.g. crossing a desert terrain will increase supply consumption by 25%). The resupply of the units relies on transportation of resources to military outposts.

Unit Creation Systems

There are unit training facilities in cities that will allow players to train different units, depending on which training facilities were built. Training a unit will consume resources and take time, both increasing based on the strength of the unit.

Combat Systems

Each turn, players can designate units to either move, shoot, suppress, or ambush. A unit cannot execute more than one of these actions per turn.

A unit ordered to move will move to a designated location, limited by its speed.

A unit ordered to shoot will shoot a target it can see. Shooting at a target moving will halve the damage dealt. Shooting at a target moving out of a unit's range will further halve the damage.

A unit ordered to suppress will cause significantly lower damage to the target, but this can build up suppression within it. When the suppression value of the target is over the suppression threshold, the target cannot move or fire until the suppression value drops back below the threshold.

Ambush mode takes a turn to set up, during which a unit will hide. Damage dealt from hiding occurs first. This can allow hidden units to deal lethal damage to an enemy before they can even return fire.

Terrain Systems

The terrain on the map will be randomly generated with a Perlin noise algorithm. Terrain elements include: plains, forests, swampland, hills, mountains, rivers, and other bodies of water. Some terrain will be difficult for units to traverse, reducing the distance a unit can travel through it. Other terrain will be impossible for units to pass through, unless otherwise specified.

Spotting Systems

The game has fog-of-war by default (can be toggled off in battle rules before starting a new game), meaning that locations of hostile units and buildings owned by enemies are not immediately known to players. Thus, in order to annihilate hostile units or sabotage enemy buildings, a player must spot them. The system can be further broken down into 2 sub-systems: visual and sound, meaning that a unit (the observee) can be spotted either "visually" or by sound by another unit (the observer).

- **Visual:**
  Whether or not the observee is spotted by the observer depends on several attributes of both of the observee and the observer. More details will be provided in the implementation section.
- **Sound:**
  A unit can only be spotted by sound after it has fired in that turn. This is to emulate artillery sound ranging in real-life wars. More details will be provided in the implementation section.

Signaling Systems

In order to assign command to friendly units, they must be receiving signals from friendly cities in that turn. The signal is invisible but can be relayed by other friendly cities or any friendly units. If a friendly unit does not receive any signals, it will become disconnected and the player will not know any details of the unit (e.g. supplies left, strength etc.) until it is reconnected again.

Vehicle Module Systems

All vehicles consist of different modules like cannon breech, turret, fuel tank, engine, track, ammo rack, etc. Each time they receive damage from the enemies, there will be a chance of damaging the modules as well. When the integrity of the module drops below a certain threshold, the module will have a chance to malfunction every time it is used. When its integrity drops to 0, it must be repaired by an Engineer unit before it can be functional again. For example, the vehicle may misfire if the cannon breech is damaged, and cannot fire at all if its integrity is 0.

Modules on a vehicle can be replaced for lower cost or higher performance. For example, a larger caliber gun can be mounted on a vehicle in order to increase damage output.

Phases

Each turn is divided into several phases, during which players can only interact with the game during the Planning phase (e.g. giving commands to units, training units etc.) under a time limit (default is 120 seconds, configurable in settings), the other phases are carried out automatically and they serve as execution of commands assigned in Planning Phase, calculations and scene updates. For clarity, the flowchart of a turn is provided below:

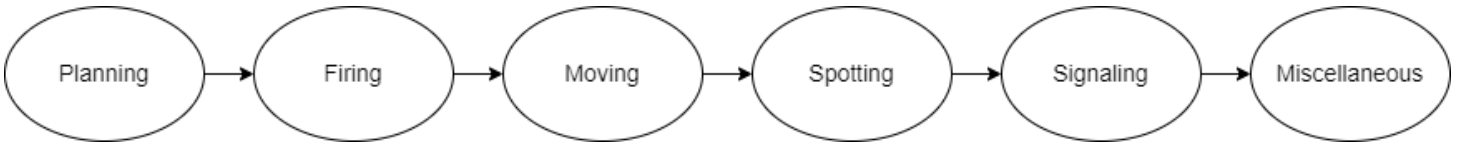


Figure 6: Phases of each turn

Balancing and Modability

All attributes of units (e.g. costs, damage etc.), number of resources added each turn are configurable by editing the corresponding JSON file. We adopt a serialization approach, instead of hardcoding all the attributes, for the ease of balancing and testing. In case we would like to change the value of certain attributes, we can simply modify the JSON file, instead of recompiling the whole game. If some players would like to have their own version of balanced attributes, they can change the files as well, and in multiplayer mode, the values are synchronized with the server host. All updated values will be loaded dynamically when the game runs.

User Interface (UI)

The user interface is an essential part of the game, as it allows players to interact with the game. Creating an understandable and effective UI is critical to the user experience. Various iterations of the UI were designed as the project proceeded.

The User Interface is built with Unity UI library [7]. The game is separated into three parts, Main menu, Waiting Lobby and Battle. These three phases of the game are visualized in three scenes for ensuring neatness of the developing

environment and contributing to different phases of the game. UI objects are grouped and placed under multiple canvases to separate the logic of UI elements.

The design of the UI implements an event-driven system by properly handling events generated from user inputs to create an interactive experience for players and to minimize the chance of having destructive bugs.

For creating easy-to-understand UI, icons, tooltip messages, message box, etc. everything was created from scratch with Unity UI library to ensure a universal style and to provide custom functionality. These UI elements provide the appropriate level of informativity for players.

The above mentioned design decisions contribute to a main part of our goal, to provide a smooth gameplay and visually appealing interface for delivering the immersive world of the game.

Network

In order to allow for multiplayer games, a multiplayer framework must be established. In this project, Unity's Netcode for GameObjects will be used. Netcode for Game Objects is a mid-level networking library created for Unity that allows programmers to abstract networking [8].

Each player should only be able to see information that is relevant to them. As such, the game needs to be designed in such a way that certain information can be hidden from players, as not not all information should be available to all players. This is handled by the networking portion of this project.

The game will be run on a server that different players can access, which handles the entire state of the game. The server will decide what information is relevant to which players and send information to those players accordingly.

There are several options for the server, either being hosted on a private server, or on a cloud server. Both options have a variety of different benefits and drawbacks. However, to reduce the cost of this project, the servers will be run locally.

When playing the game, one machine starts as host, and the others start as clients. Clients connect to the host machine, which serves as a server. The host

machine is a client that is connected to itself. The server keeps track of the game state and sends the relevant information to the clients that need it.

2.2. Implementation

Our project implementation consists of 5 major parts, which are map generation algorithm, gameplay, multiplayer and the models. Gameplay can be further broken down into User Interface (UI) and User Experience (UX), the implementation of the bot algorithm, and data handling.

2.2.1. Map Generation Algorithm

A map consists of tiles. Each tile has a terrain type. Instead of predefining all terrain types, providing the feature of random generations could make the game be more interesting because it creates lots of variety and possibilities for players to explore. They may even have to adopt different strategies based on where the metropolis they are assigned to. This is of utmost importance to fulfilling our project goal. However, a map cannot be generated by using one algorithm solely. Instead, it is generated by using a combination of algorithms, tweaked to suit our use. The map generation consists of 3 stages. Firstly, we generate the terrain without streams and cities. Then, we generate the streams and overwrite the existing terrain. Finally, we generate the cities and overwrite them again.

Terrain generation

The terrain is generated using two 2D Perlin noise maps, one named “Height” and the other “Humidity”. However, the Perlin noise maps generated would be plain and monotonous if we use the *Mathf.PerlinNoise* function directly. Extra parameters are needed to control the details of the Perlin maps. Here is the list of extra parameters used (same parameters can be found in class diagram of PerlinMap class in section 10.5):

- Frequency
- Octaves
- Persistence
- Exponent
- Warp Strength
- Seeds (x and y)

Changing the frequency is basically equivalent to zooming in and out of the Perlin map, so it in fact controls how closely the crests and troughs are packed, the higher the more closely packed.

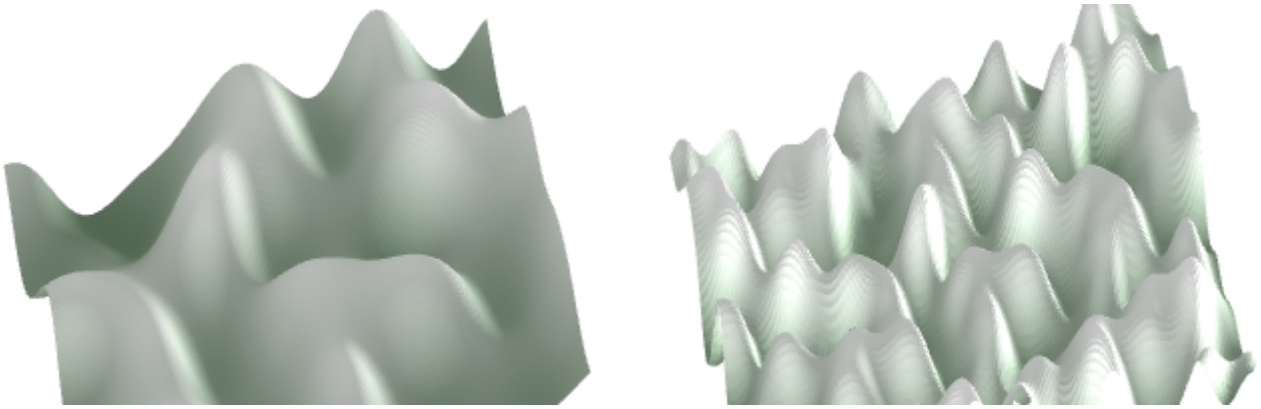


Figure 7: A Perlin map with frequency = 2.0 (left) and 4.0 (right)
Source: [9]

The octave parameter indicates how many maps of exact multiples of power of 2 of the base frequency are added to the base map, and the perlin random result of each octave is attenuated by ½^(octave). For example, an octave of three with base frequency of 1.0 means the following:

$$height = 1 * perlin(1 * x, 1 * y) + 0.5 * perlin(2 * x, 2 * y) + 0.25 * (4 * x, 4 * y)$$

where x and y are the coordinates clamped from 0 to 1 by dividing them by width and height of the map respectively. 1, 0.5, and 0.25 are called the amplitudes.

So in fact the number of octaves controls how detailed the map is (ie. the higher the more detailed). But this creates a problem, that sometimes the sum of the results is larger than 1. A simple fix would be dividing the result by the sum of the amplitudes (1 + 0.5 + 0.25 in this case).

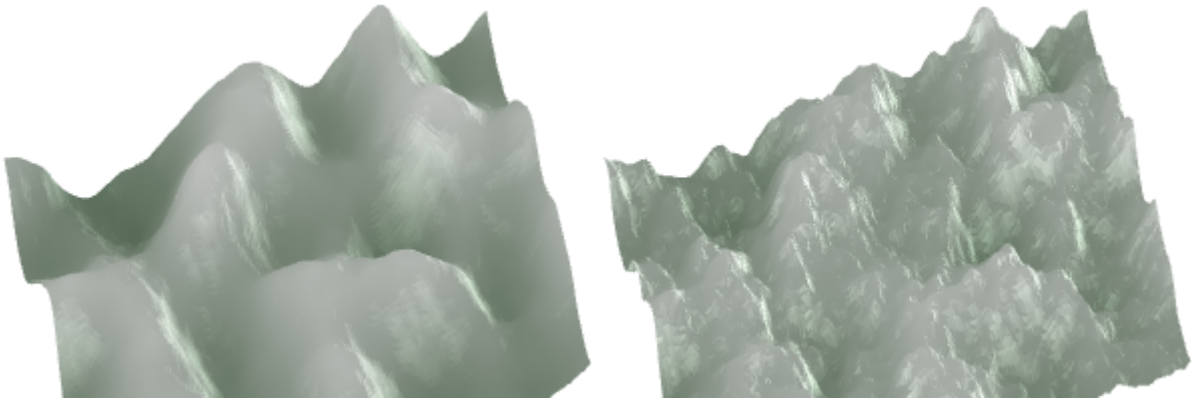


Figure 8: A Perlin map with lower octaves (left) and higher octaves (right)
Source: [10]

The persistent parameter indicates how quickly the amplitude attenuates across octaves. So, if a persistent value of 0.7 is used in the previous example (instead of 0.5), the height will become:

$$height = 1 * perlin(1 * x, 1 * y) + 0.7 * perlin(2 * x, 2 * y) + 0.49 * (4 * x, 4 * y)$$

So in fact, the persistence parameter controls how "clear" the map is, the higher the clearer. And again, the result should be clamped using the aforementioned method.

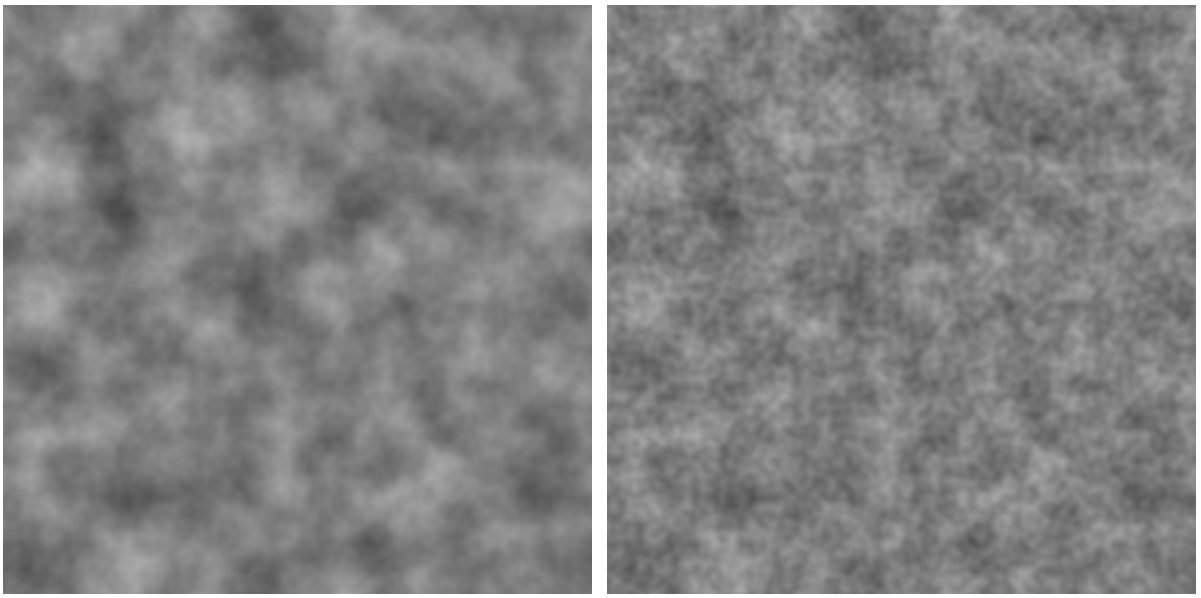


Figure 9: A Perlin map with persistence = 0.1 (left) and 0.8 (right)

The exponent parameter is applied to each point on the perlin map. Raising the power will result in "pointier" crests, vice versa.

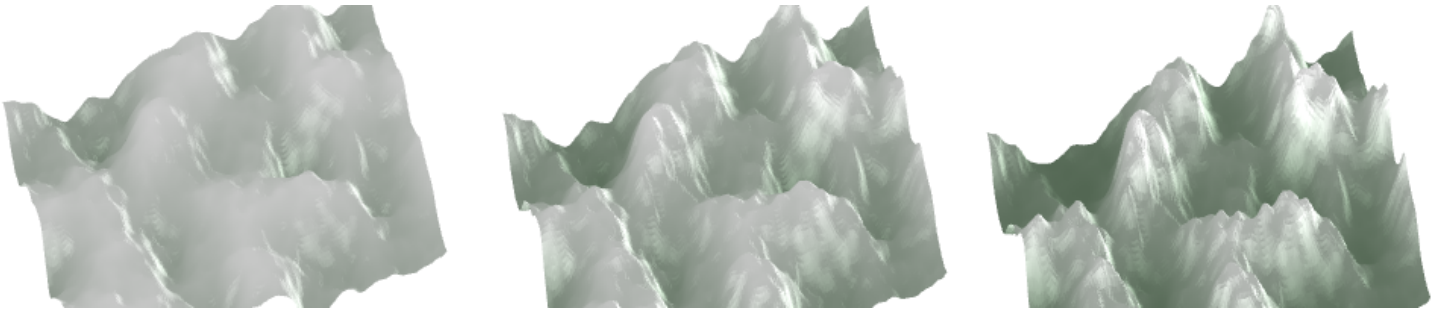


Figure 10: A Perlin map modified with exponent = 0.5 (left), 1.0 (mid, no change), and 2.0 (right)

In order to create an even more irregular map, a technique called warping is used. Warping can create distortion in textures or geometries.

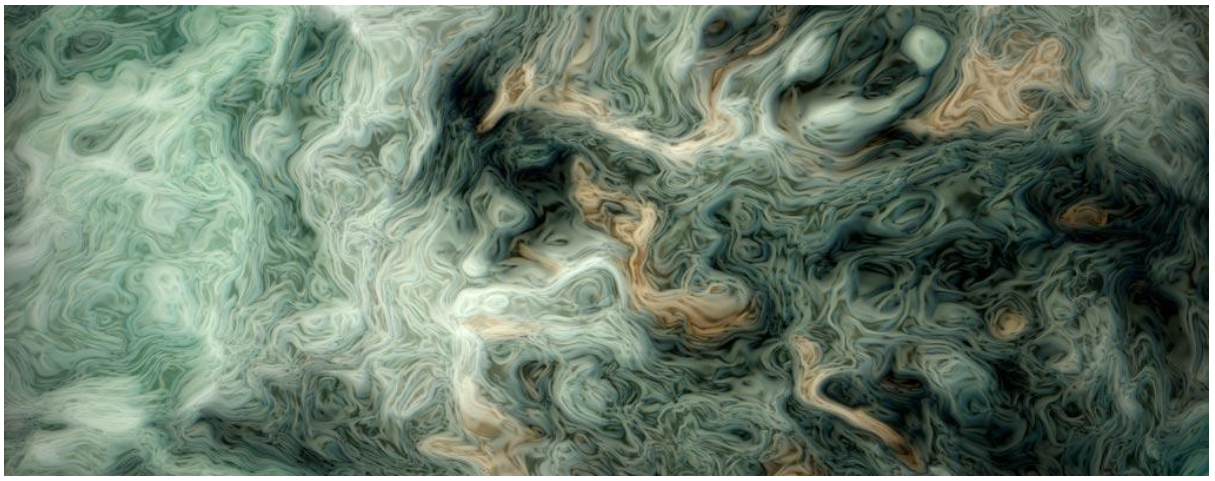


Figure 11: A warped texture generated by fractional Brownian motion
Source: [10]

The mathematical function implemented for distortion in our project is simple, which is: $height = perlin(p + perlin(p) * warp)$
where p is a normalized point (x, y) and warp is the Warp Strength parameter.

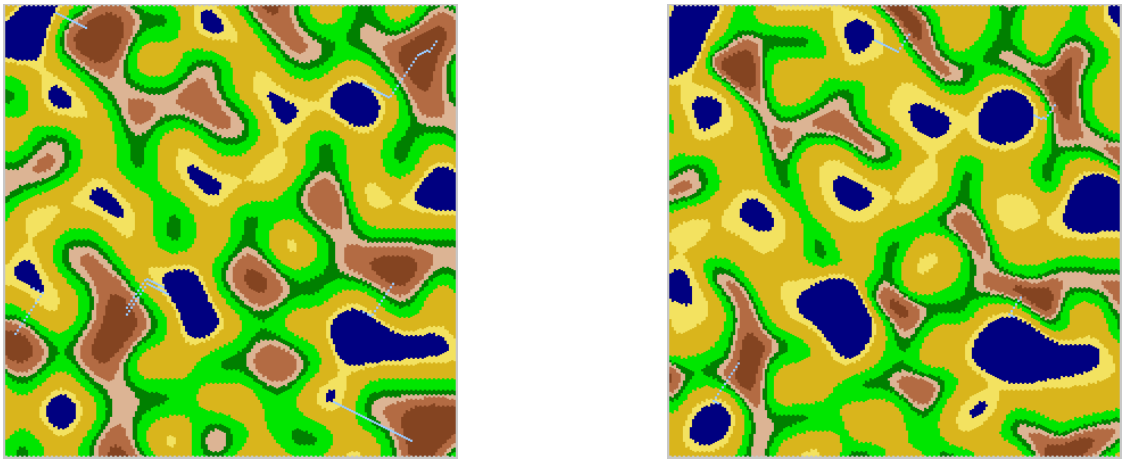


Figure 12: A generated game map with warp strength = 0.0 (left, no warp) and 0.1 (right). Notice the slight distortion of the whole map

As we are using the *Mathf.PerlinNoise* for picking perlin random numbers, we do not have direct control of the permutation table used in that function. So the "Seed" parameter mentioned above is in fact an offset of the starting coordinates of the Perlin map to mimic the randomness of using a seed to generate the permutation table. Further problems and improvement will be discussed in Section 3.1.

After generating the two necessary Perlin maps ("Height" and "Humidity"), the terrain types are assigned according to the two values of the maps. The constants and the corresponding terrain types are listed in the below table:

Height	Humidity	Terrain
[0, 0.25]	Not considered	Ocean
(0.25, 0.4)	[0.7, 1]	Swamp
(0.25, 0.6]	[0, 0.3)	Desert
	[0.3, 0.45)	Plains
	[0.45, 0.55)	Grassland
	[0.55, 0.65)	Forest
	[0.65, 1]	Jungle
(0.6, 0.65]	Not considered	Hillock
(0.65, 0.75]		Hills
[0.75, 1]		Mountains

Table 1: Constants used for determining terrain types from “Height” and “Humidity” map

Stream generation

Stream generation is the most challenging task compared with the other two stages. The final implementation follows the steps below:

1) From all high land tiles (hillock, hills and mountains), randomly pick a number of tiles equals to *highlands count* / 1000 as number of water sources
2) From all ocean tiles, randomly pick a number of tiles equal to the number of water sources as the destinations of the rivers. If there’s no ocean in the map, skip the whole river generation stage
3) Pair up the sources and the destinations with closest distances
4) For each pair, use the A* algorithm with custom weighting to find the path connecting the source and the destination. Store the path in a stack

Terrain	Weight
Boundary	Infinity
Stream, River, Ocean, Swamp	1

Desert, Plains, Grassland, Forest, Jungle	2
Hillock, Rocks	4
Hills	8
Mountains	16

Table 2: Weights used for A* for rivers generation

5) For each path found, and foreach tile along the path, pop the tile out and then do the following:

Current tile terrain	Action
Boundary or Ocean	Stop and break out
River	Skip overwriting current tile
Stream	Overwrite with River
Others	Overwrite with Stream

Table 3: Logic for overwriting the tiles along the river path

The reason for having this logic is that sometimes the paths may be partially overlapped, and this is to ensure that the segments where the paths overlap become rivers (a larger stream), and the rest of the paths become streams.

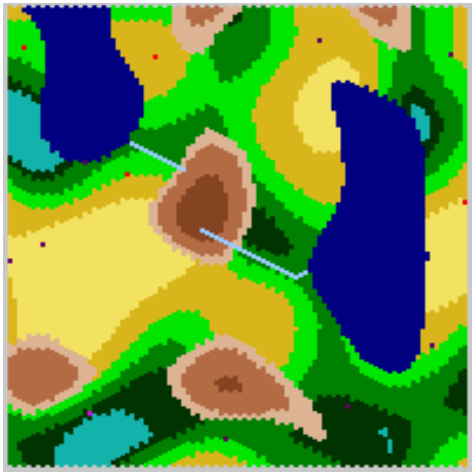


Figure 13: An example map generated with streams (light blue lines)

As seen from the above map, the generated river is a bit straight. Further possible improvement will be discussed in Section 3.1.

Cities generation

As mentioned in the previous report, we encountered a problem where the cities generated were sometimes overcrowded. We managed to solve the problem by having our own implementation of a famous algorithm called the Poisson Disk. The implementation is as follows:

```
1400    private void PickCities(int num_cities, int min_sep, int max_sep)
1401    {
1402        Debug.Log($"Generating cities: no. of cities to generate: {num_cities}, min. separation: {min_sep}, max. separation: {max_sep}");
1403        bool force_generate_without_checking = false;
1404        while (num_cities > 0)
1405        {
1406            // if not have any cities yet, randomly pick one from available;
1407            if (m_cities.Count == 0 || force_generate_without_checking)
1408            {
1409                CubeCoordinates c = m_flatLands[Utilities.Random.Next(m_flatLands.Count)];
1410                m_cities.Add(c);
1411                _ = m_flatLands.Remove(c);
1412                force_generate_without_checking = false;
1413                num_cities--;
1414            }
1415            else
1416            {
1417                Coordinates last = (Coordinates)m_cities.Last();
1418                // test for any cities already generated within min_sep of candidate, if true, pick another one
1419                bool has_cities_within_range = true;
1420                while (has_cities_within_range)
1421                {
1422                    // pick one
1423                    int rand_dist = Utilities.Random.Next(min_sep, max_sep);
1424                    double rand_theta = Utilities.Random.NextDouble() * 2 * Math.PI;
1425
1426                    int candidate_x = (int)(last.X + rand_dist * Math.Cos(rand_theta));
1427                    int candidate_y = (int)(last.Y + rand_dist * Math.Sin(rand_theta));
1428
1429                    // out of bounds
1430                    if (!(candidate_x > 0 && candidate_y > 0 && candidate_x < Width && candidate_y < Height))
1431                    {
1432                        continue;
1433                    }
1434
1435                    CubeCoordinates candidate = (CubeCoordinates)new Coordinates(candidate_x, candidate_y);
1436                    if (!m_flatLands.Any(s => s == candidate)) // not a flatland
1437                    {
1438                        continue;
1439                    }
1440                    has_cities_within_range = candidate.GetNeigbours(min_sep).Intersect(m_cities).Any();
1441
1442                    // be it passing the test or not, it won't be suitable: if it passes, it will be turned into a city, if not then obviously not a suitable tile for any other city generation
1443                    _ = m_flatLands.Remove(candidate);
1444                    if (!has_cities_within_range)
1445                    {
1446                        m_citie    (local variable) bool has_cities_within_range
1447                        num_cities--;
1448                        break;
1449                    }
1450                }
1451            }
1452        }
1453    }
```

Figure 14: Code for cities generation

The code shown above does the following:

- while the number of cities to be generated is larger than 0:
  - if there aren't any cities generated, generate a city by randomly picking a tile from flatlands (desert, plains, grassland, forest and jungle), minus 1 from number of cities to be generated
  - else get the last city generated, read its coordinates
    - pick a random value between minimum and maximum separation
    - pick a random angle of rotation
    - get the coordinates of the potential candidate of city
    - if the coordinates are out of boundaries, repick
    - if the potential candidate is not a flatland, repick
    - if there is at least 1 city within the minimum separation of the potential candidate, repick
    - else generate the city, minus 1 from number of cities to be generated

2.2.2. Gameplay

2.2.2.1. User Interface and User Experience

The UI elements are based on Unity UI library, a GameObject based UI system. Each GameObject has its transform component which defines the hierarchy of GameObjects in the scene. Functionalities are provided by attaching C# scripts to the GameObjects as components. The functionalities are achieved by first designing the layout with the aid of the preview function of Unity Editor, followed by attaching appropriate components from Unity UI library and custom scripts.

To serve the purpose of creating an interactive environment for players to manage their units and cities, the UI system is complex and bulky. This report will highlight the mechanisms that are essential to the project when implementing different parts of the UI system instead of enumerating the elements created.

Scene Management

The project is divided into three scenes, *Menu.unity*, *Loading.unity* and *Game.unity*.

Menu.unity	Responsible for Main Menu
Loading.unity	Responsible for Lobby while assets of battles are loading
Game.unity	Responsible for demonstrating the actual game in battles

Table 4: Scenes in Project

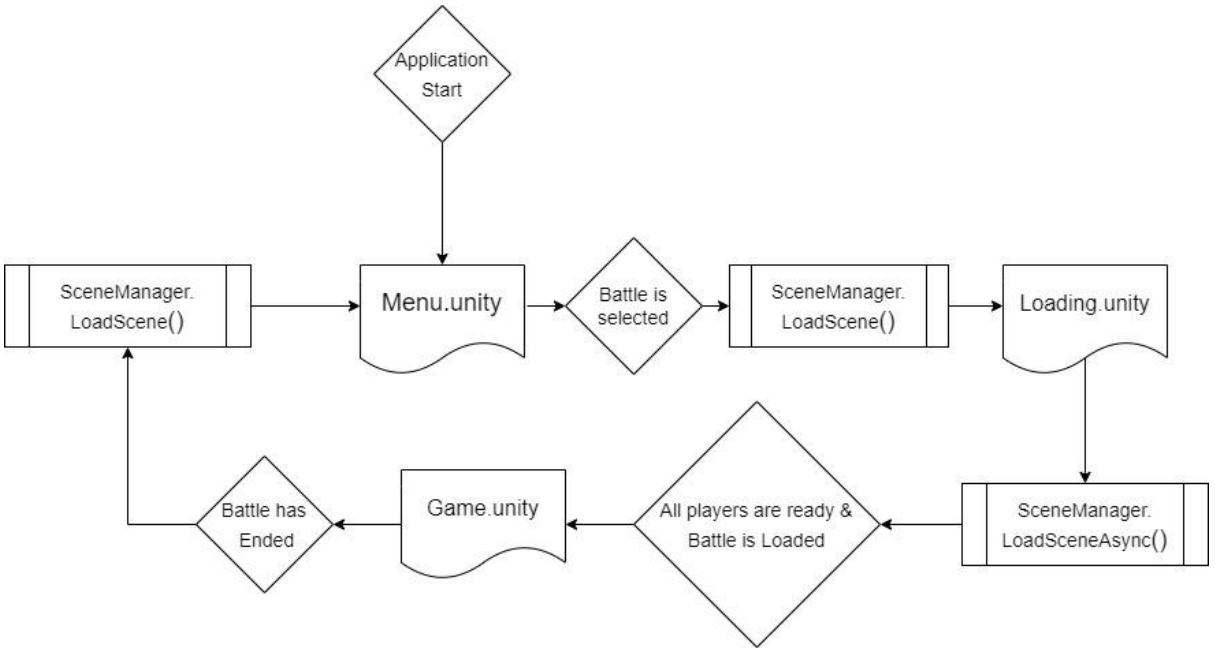


Figure 15: Flowchart of Scene Management

To perform scene switching, *SceneManager.LoadScene()* and *SceneManager.LoadSceneAsync()* under namespace *UnityEngine.SceneManagement* can be used[11]. In order to reduce the loading time, these two methods are used in different scenarios. As *LoadScene()* runs in the main thread and is blocking, it is used only when loading light-weight scenes, which includes *Menu.unity* and *Loading.unity*.

Async method *LoadSceneAsync()* will stop after loading 90% of content when *allowSceneActivation* is set as false. Our scene management algorithm takes advantage of this to control the actual timing for switching the scene and simultaneously avoiding a heavy task blocking the main thread, which would result in laggy user experience if not properly handled.

Threading and Coroutine

In order to avoid thread blocking when running resource intensive tasks, threading and coroutine are two approaches to solve the problem. An asynchronous task that runs on another thread can be created with *Task* class under *System.Threading.Tasks* provided in C# library. *Coroutine* is provided in Unity library. The principle of the Coroutine is to pause and resume the task at set points to avoid blocking. This concurrency approach allows huge tasks to be disassembled to run in multiple frames and cross-frame tasks to be easily implemented. These two approaches have their unique strength and weakness and are suitable in different scenarios.

Threading	Coroutine
runs in another thread	runs in main thread
need to be managed manually	managed by UnityEngine
not attached to a GameObject	attached to a GameObject

Table 5: Comparison between Threading and Coroutine

The flexibility of creating an asynchronous task is higher as it does not attach to any GameObject. It is suitable for running background tasks that are not related to GameObjects. In this project, it is used in generating huge map instances, loading battle assets and sending/receiving packets over the network. This reduces the load in the main thread which solves the significant frame per second (FPS) drop when those tasks are running.

The fact that coroutine runs in the main thread is not always a disadvantage. When running Unity Engine methods that are related to GameObject and graphics, a proportion of them must run in the main thread. It acts as a workaround to increase performance of the game in these situations. In our project, coroutine is used to delay the operation of the resize method in *Resize.cs* for certain frames to avoid the unintended behavior when working with layout groups. Coroutine can also be used with the above-mentioned asynchronous task as handlers when the task is finished. It is used in handling the end of loading battle assets tasks and multiplayer packets transmitting tasks.

Camera

A camera controller to control the movement of the camera is implemented so that players can easily focus on a certain target when changing the angles of the camera.

For a typical first-person-perspective camera, moving a small angle would result in a great degree of view change and thus is not suitable in a strategy game if the player would easily lose the target. A simple implementation would be fixing the angles of the camera but it is not satisfactory as the freedom in controlling the camera allows players to have a better understanding on the map.

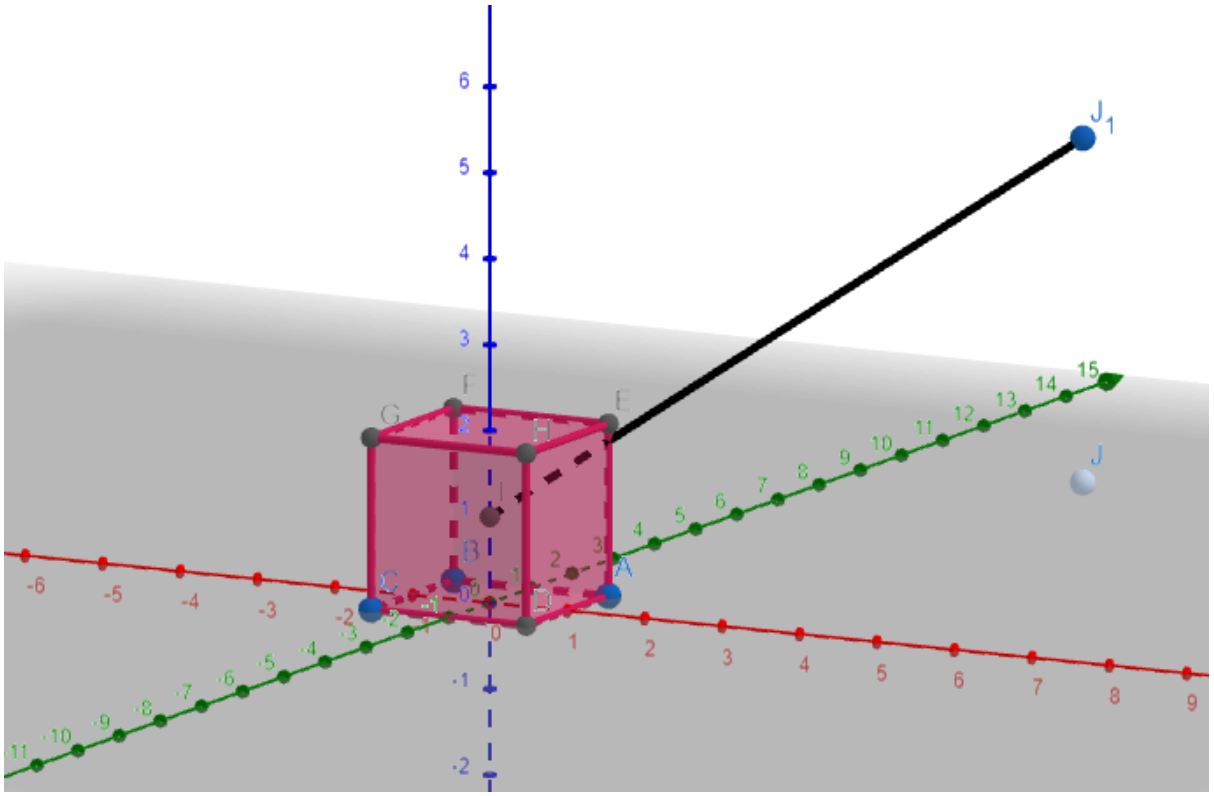


Figure 16: Camera Demonstration - Perspective 1

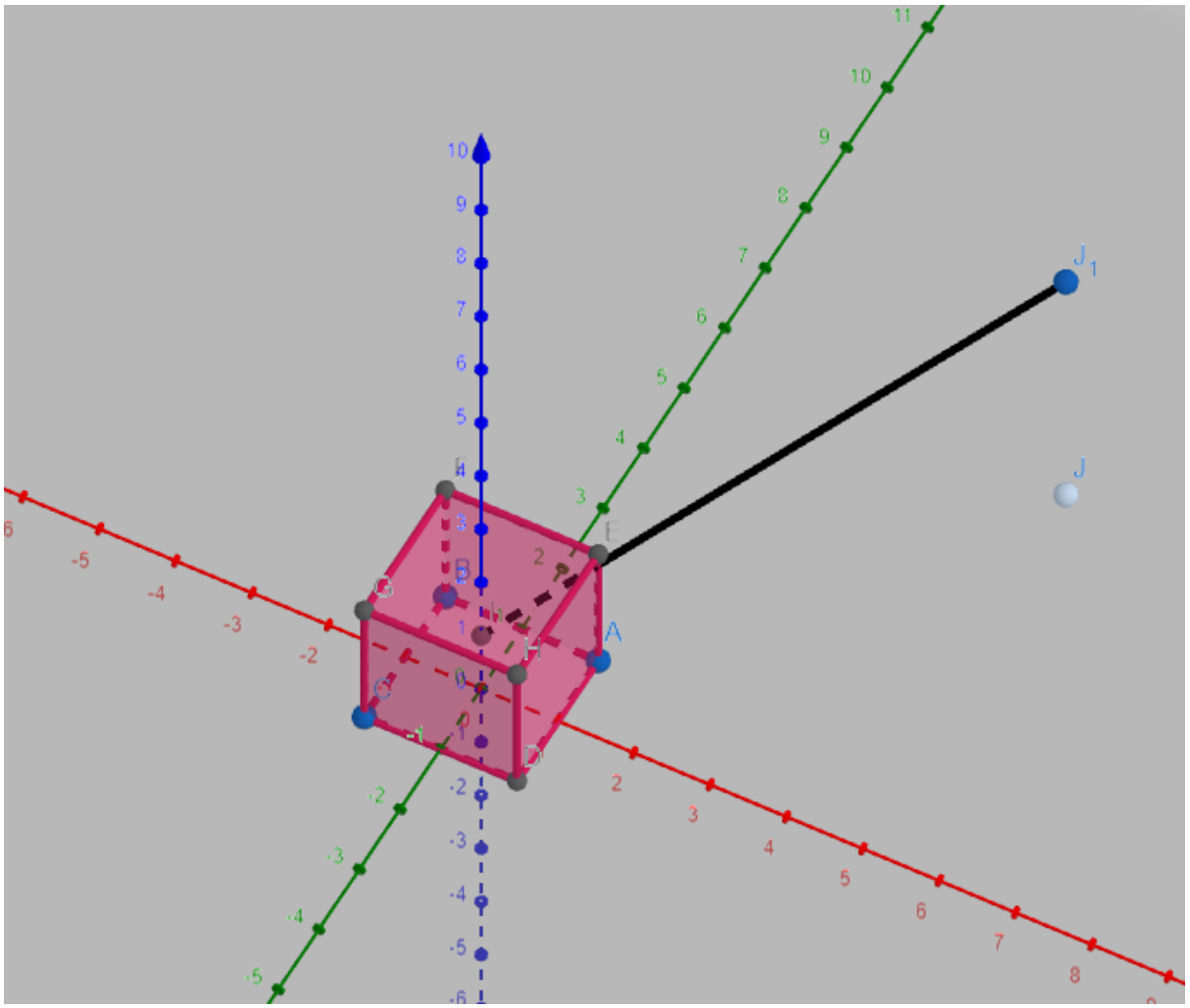


Figure 17: Camera Demonstration - Perspective 2

Let the red axis as x-axis, green axis as z-axis and blue axis as y axis. The target is the center of the cube, point I. The camera is set at J_1,facing the direction of $\vec{J_1 I}$. The target could be fixed when the camera is rotating about y-axis or line $x = 0 \; and \; y = 2$ which meets the requirement that the target is fixed when the camera is rotating. The following table shows the conversion rules of user control and backend camera calculations.

moves left/right/forward/backward	translate I on xz-plane
rotates up/down	alter angle of $\angle JIJ_1$
rotates left/right	alter the angle between IJ and xy-plane
zoom in/out	alter length of IJ_1

Table 6: Conversion Rule from User Control to Camera Calculation

There are 4 variables to control the camera.

- Vector2: position of target (z is fixed as 0)
- float: length between camera and target
- float: z-rotation in degree
- float: x-rotation in degree

To limit the movement area to avoid the player's camera not pointing to the playmat, maximum and minimum value of these four variables can be set. The implementation of this camera controller does not limit the z-rotation intentionally.

In order to smoothen the camera movement, linear interpolation is performed every frame to the four variables to execute a proportion of the change. This algorithm creates a smooth curve of motion which acts as a good simulation of camera movement in the real world.

Animation with Tweening

Tweening approach is chosen to implement simple animations for UI elements. Tween Animations are created by defining two keyframes of the animations and interpolating the frame in between. LeanTween library is used for this task[12]. LeanTween library allows tweens and settings to be chained together which increases the productivity of creating animations. The LeanTween GameObject can also be destroyed automatically after the animation is completed which

reduces memory usage. These features make this library stand out for creating Tween Animations.

Linking mechanism between GameObject and Backend Prop

UnityEngine is a GameObject-based engine, which means that any script needs to be attached to a GameObject to execute. In order to link backend *Prop* and the corresponding GameObject, *PropObject* class is defined and inherits MonoBehaviour so that it can be attached to the GameObject while providing all necessary information needed by UI elements. A unique identifier, *MeshName* is generated for each Prop and the value of MeshName is set to be the name of the GameObject on the scene. With this unique identifier, the corresponding GameObject can be found with *GameObject.Find(string gameObjectName)* provided in Unity library and the corresponding Prop can be located by calling *Map.GetProp(GameObject gameObject)* which is defined in *Game.cs*.

Event trigger

Event Trigger is a class under *Unity.EventSystems* in Unity library for triggering the functions after the events are invoked. Unity provides a predefined Event Trigger component for attaching to GameObjects which can be easily managed via the Unity Editor.

The approach of adding the predefined component is adapted when creating UI elements for productivity. For *PropObject*, we decided to implement class *PropEventTrigger* in *CustomTypes.cs* which inherits the Event Trigger Class for providing a custom feature, toggling the activation of triggered functions. The implemented class is shown in the graph below.

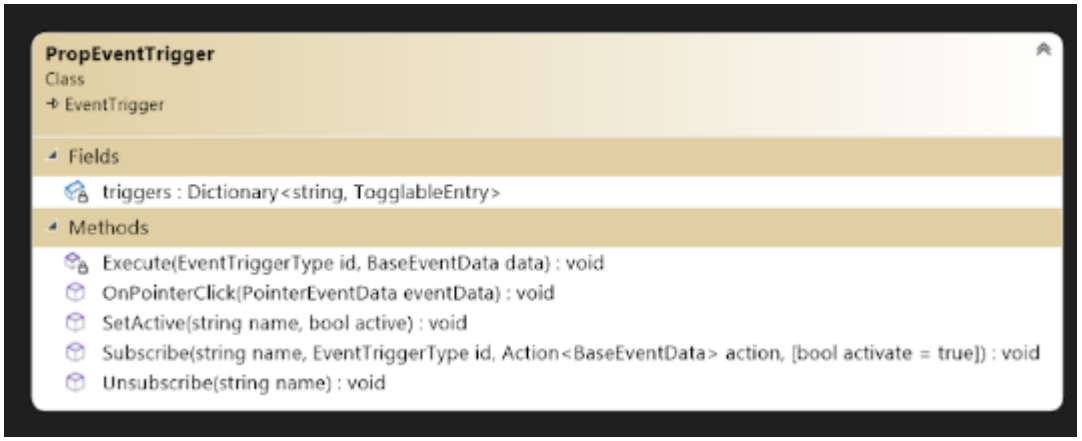


Figure 18: Class Diagram of PropEventTrigger (See Appendix C)

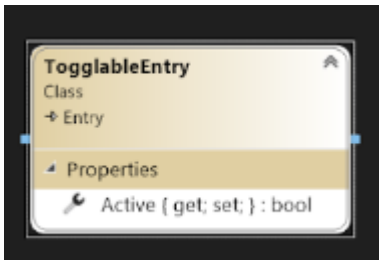


Figure 19: Class Diagram of TogglableEntry (See Appendix C)

TogglableEntry class inherits EventTrigger.Entry for building a dictionary that stores the unique identifier and the event entry. The event entry can be attached to the actual function to be run during runtime. This custom class helps relieve the performance issue created by subscribing and unsubscribing to events repeatedly. As the number of PropObjects scales with the map size that can be customized by players, the overhead generated by the subscription and unsubscription would become significant when the size of map increases. The toggle mechanism converts the expensive operations to less expensive assignment operations which contributes to providing a smooth gaming experience.

2.2.2.2. Bot Algorithm

In order to create an interesting game, which is one of the important goals in our project, we implement a bot algorithm that can help players learn some new strategic skill inside the single player mode.

For the bot logic algorithm, we created sets of algorithms which provide assistance for the bots when they are executing their designated tasks. The bot algorithm is designed based on the methods and classes that are created inside the project. Some of the methods and classes that are used inside the algorithm are shown below :

Prop Class

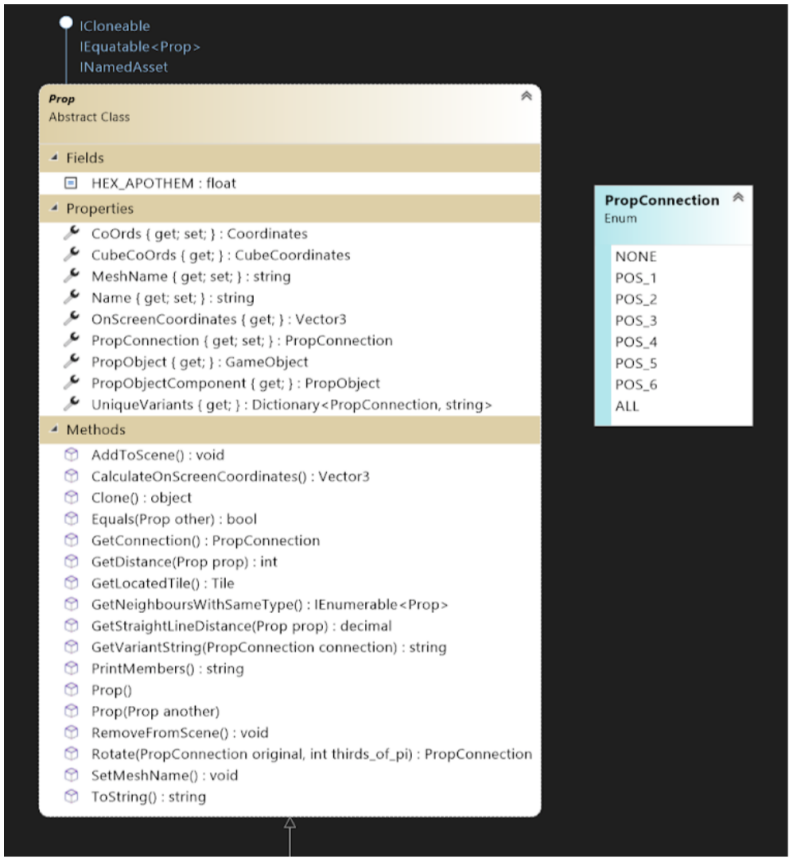


Figure 20: Class Diagram Prop (See Appendix C)

From the Prop class, we use the *CoOrds* and *CubeCoOrds* for the Building and Unit placing. Essentially, the *CoOrds* contains 2 integer variables while *CubeCoOrds* contains 3 integer variables. *CoOrds* is only used for Building, Cities, and Unit coordinates placement, While *CubeCoOrds* is used not only in the coordinates placement, but also used in determining the direction and path. Moreover, *CubeCoOrds* lets us easily calculate the distance between 2 places.
Some of the methods that are used inside the bot algorithm are *GetDistance* and *GetLocatedTile*

Map Class

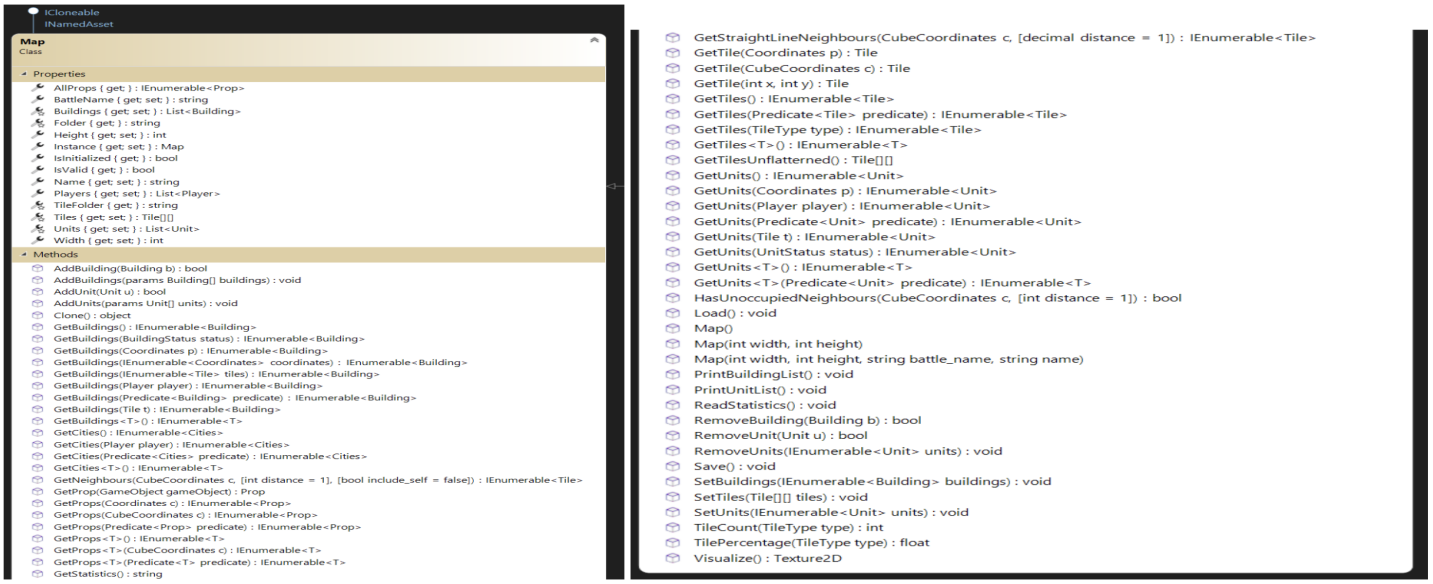


Figure 21: Class Diagram Map (See Appendix C)

From the Map Class, we use this to retrieve Cities, Buildings, Tile, and Units. It is used to retrieve ally props or enemy props. Some of the methods we use inside the bot algorithm are *GetCities(...)*, *GetNeighbour(...)*, *GetTile(...)*, and *GetUnits(...)*.

Unit Class and the Methods

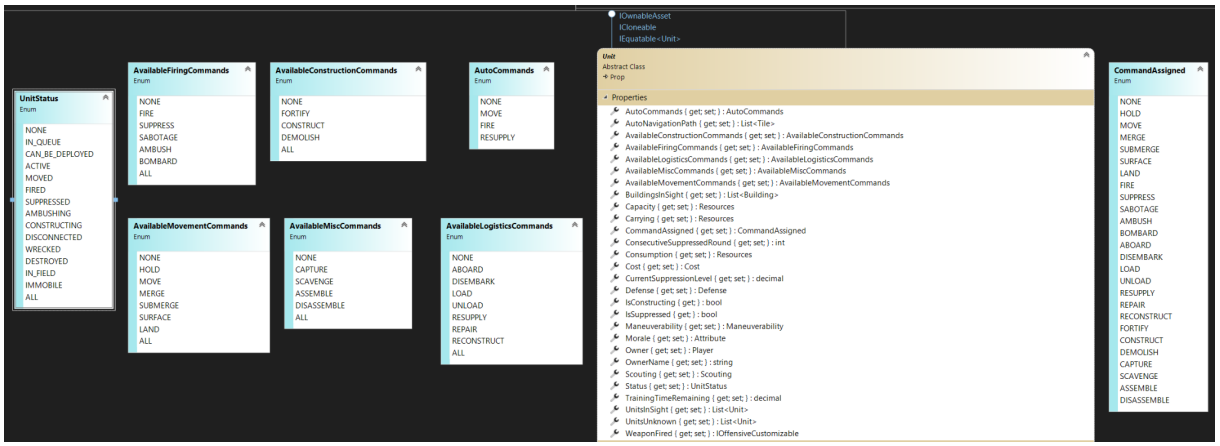


Figure 22: Class Diagram Unit (See Appendix C)

Figure 23: Methods Diagram Unit (See Appendix C)

Inside the bot algorithm, we use most of the methods and properties from the Unit Class as most of the algorithms are used to manage the bot's units movement.

Building Class

Figure 24: Class Diagram Building (See Appendix C)

UnitBuilding Class

Figure 25: Class Diagram UnitBuilding (See Appendix C)

From the Building Class and UnitBuilding Class, we use these classes with the Map Class in order to get friendly or hostile buildings in the area using *IsFriendly()* and *IsHostile()*. From the UnitBuilding Class, which is the subclass of Building, we use some properties inside. We use TrainingQueue and ReadyToDeploy properties to get the number of units being queued and which units are ready to deploy inside the UnitBuilding.

The algorithms will help the bot adjust accordingly depending on the resource generated each round, enemy player's movements and enemy player's units. We divide our algorithms into several parts, a sample general flowchart of our bot algorithm is shown below.

Figure 26: Bot Algorithm flow

For the training algorithm, we first check and deploy any deployable unit for each Unit Building. After deploying, based on the resources acquired each round and the unit train priority, we want our bot to train a specific type of unit accordingly. The current unit train priority consists of the number of same type units and enemy units composition. The sample flowchart of the training algorithm is shown

below.

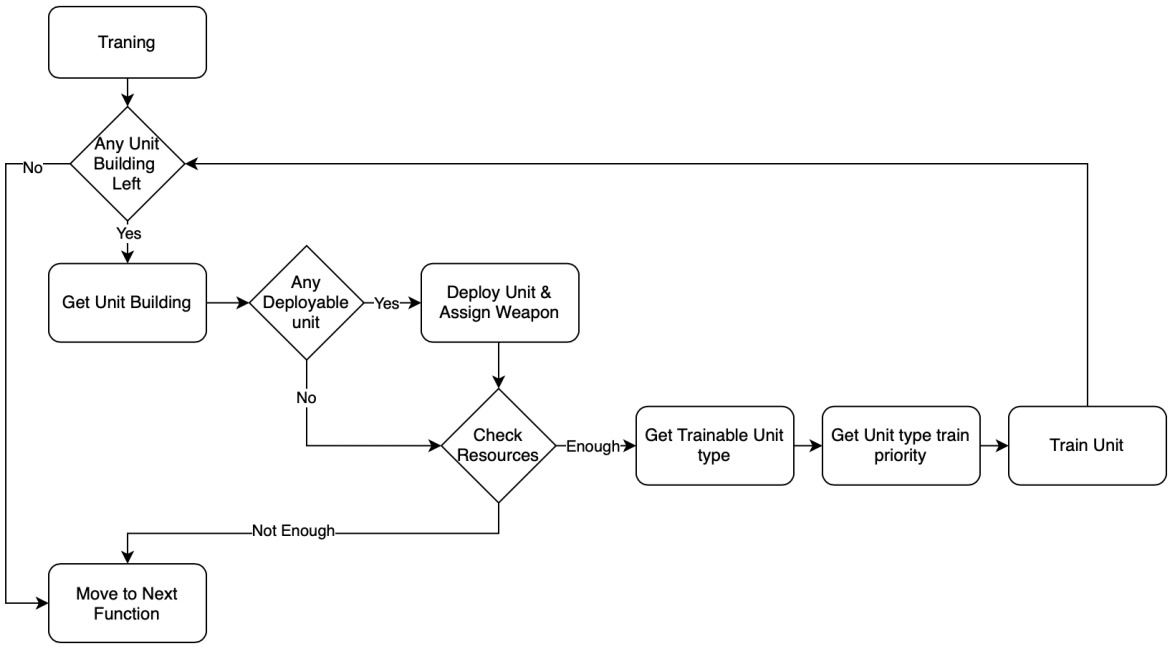


Figure 27: Training Logic

For the capture city logic, at the beginning, the bot will get the nearest neutral city from the metropolis. Then, it will assign a unit which is closest to the neutral city to move there using the recommended path from the pathfinding algorithm. If the assigned unit has arrived, it will capture the city instead. The sample flowchart of the algorithm is shown below.

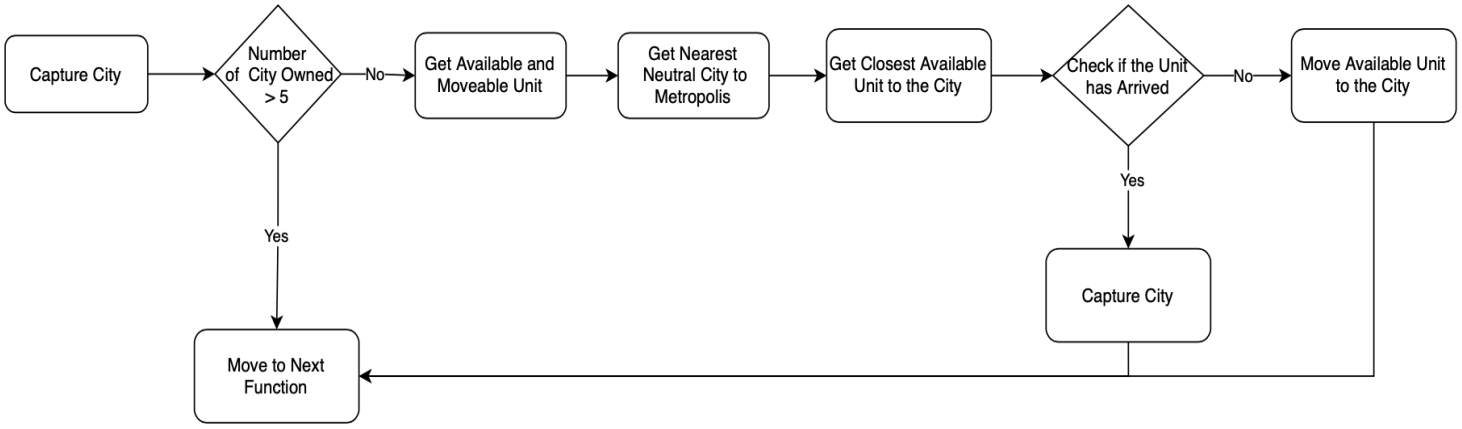


Figure 28: Capture City

For the supply, for each round, the bot will get the units inside the range of the owned city or outpost. The algorithm will assign the supply needed according to capacity of each unit type. The sample flowchart is shown below.

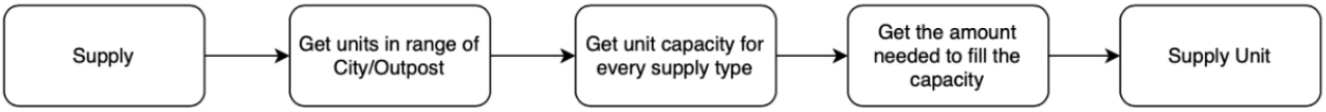


Figure 29: Supply

On the construction part, we limit the amount of building that can be constructed in each round to prevent any excessive resource usage. Then, for the Unit Building or Resource Building, we first check the type of building that has been built around the city. From those, we let the algorithm choose which type of building should be built according to the construction priority. As for the outpost, for every 2 owned cities, we will assign an engineer type of unit to go into the middle point of those 2 cities. The unit will construct an outpost when it arrives at the midpoint.

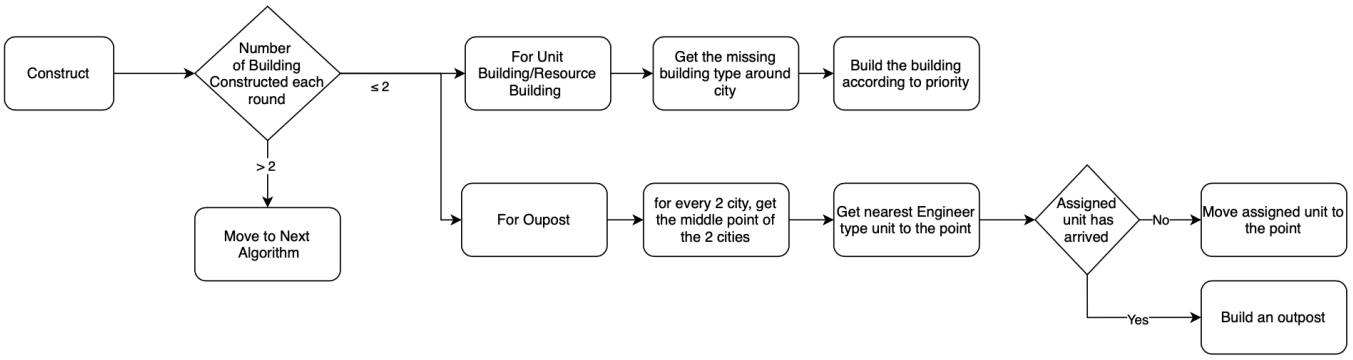


Figure 30: Construct

For the movement part, we want the bot to achieve a certain point before doing some movement. We want to make sure if the bot's unit composition is fully prepared before raiding enemy cities. If the condition has been fulfilled, it will assign a group of 8 units to move toward the closest enemy city. When the units arrive at the midpoint, they will build an outpost in order for them to resupply their resources. When the enemy city is in range, the units will try to kill all enemy units around the city before trying to sabotage the opponent city.

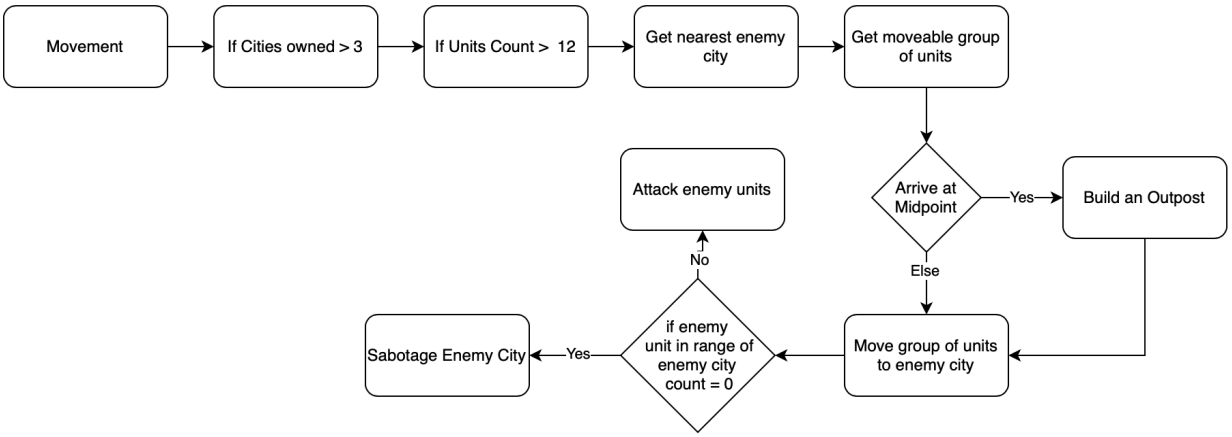


Figure 31: Movement Logic

For the combat parts, we want the bot's units to win as much as possible. Every time any unit found the enemy inside the attack range, we compare the number of enemy units in range with the number of friendly units. If the number of enemy units are more than the friendly units, it will retreat to any available friendly unit in range. If it is not possible, it will attack instead.

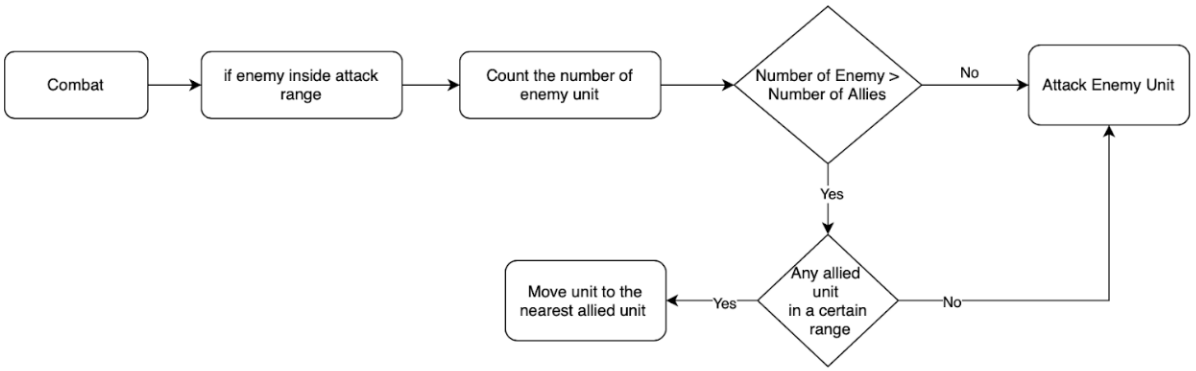


Figure 32: Combat

2.2.3. Multiplayer

An important goal for our project is to create a playable multiplayer game. For a multiplayer game to exist, the host server must be able to communicate with the clients. This allows the server that is running the game to send vital information to the clients that are connected to the game. Some assets such as the map are not known by the client when they first join and need to be sent to each connected client.

In the framework that we have chosen, Netcode for GameObjects [8], there are 2 ways for the client and the server to communicate. The first of these are Named Messages, which allow for large data transfer between the client and the server. The second of these are Remote Procedure Calls which as the name implies, allow procedures to be called remotely, in this case on the server. Each of these communication methods has its own niche, and need to be used in different scenarios.

Additionally, before a client even connects to a server, the server must perform some safety checking of the client for security and quality of life. The client must ask the server to connect, and pass an approval check before the client is allowed into the game.

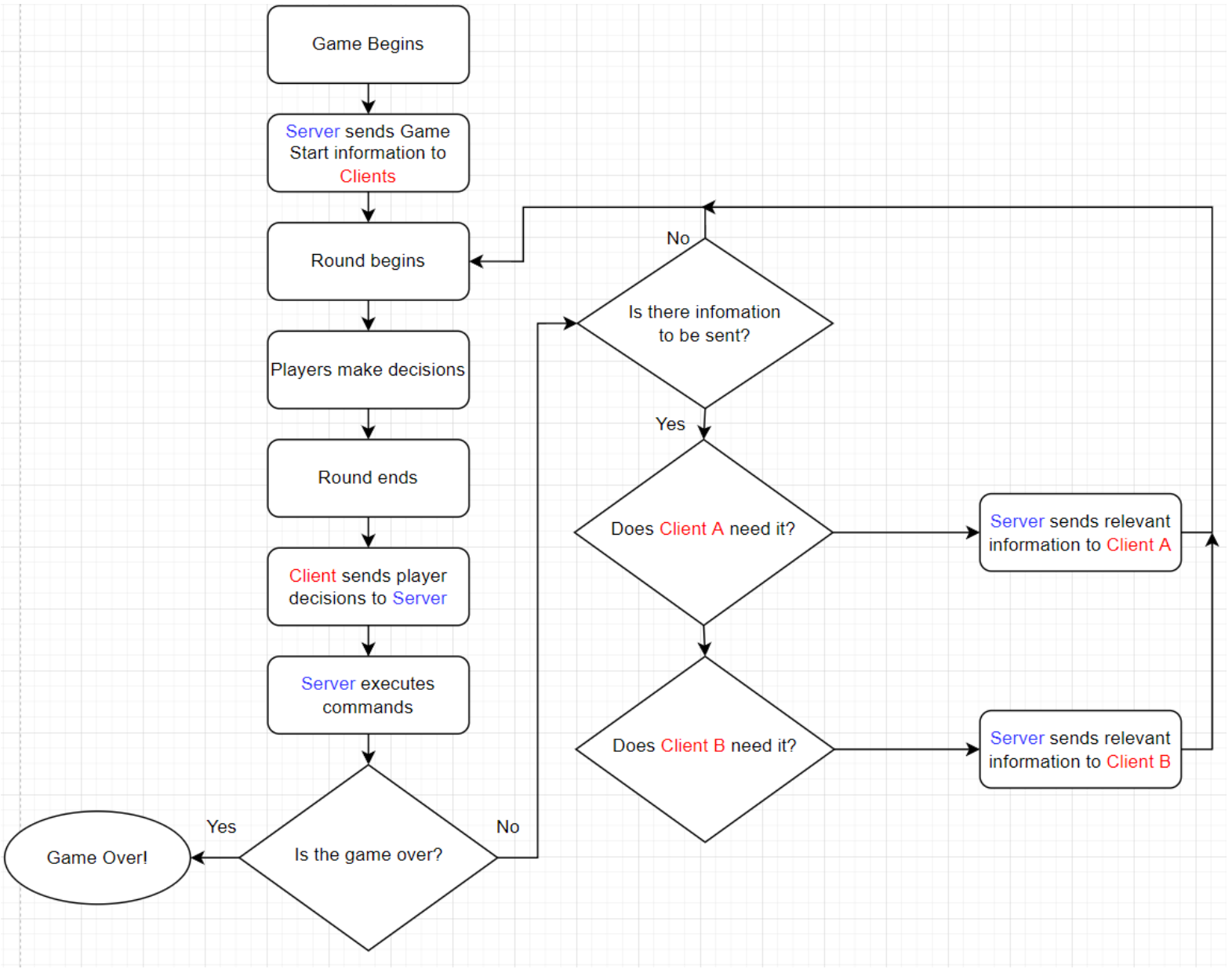


Figure 33: Flowchart of server - client communication during the game

With the exception of the large amount of data to be sent at the beginning of the game, the majority of the communication is done through RPCs. However, without the start of game communication, the game would not be able to function. Thus, both methods of server-client communication are very important.

The implementation of both the RPCs and Named Messages are dealt with by the NetworkManager object provided by Netcode for GameObjects. The NetworkManager object is placed into every scene. We also implemented a NetworkUtilities class that inherits from NetworkBehaviour, the network equivalent of MonoBehaviour that the vast majority of other classes inherit from.

This NetworkUtilties class is a helper class that assists with sending Named Messages and RPCs.

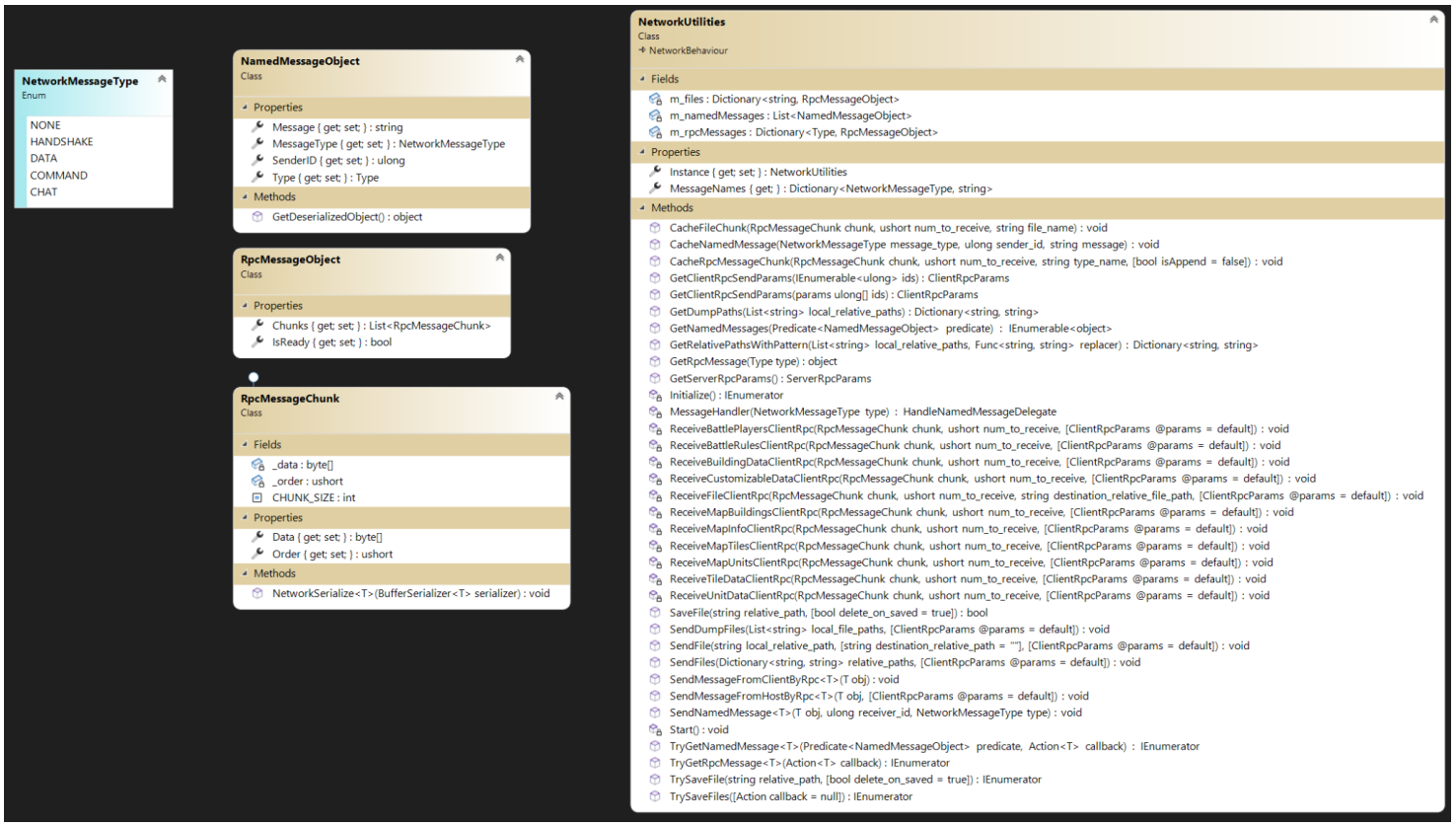


Figure 34: Class diagram of Network Utilities (See Appendix C)

Named Messages

In NetworkUtilities, there are various methods that allow different classes to send a Named Message. In our game, the only time the client needs to receive a Named Message from the server is at the beginning of the game. This can be optimized by sending the data when the client connects. In Unity's Netcode for GameObjects, this can be done easily by attaching a function to the client connection delegate, which fires when the client connection event occurs.

In the Start method of the Battle class, which is executed when the Battle class is initialized, the implemented function OnClientConnected is added to the OnClientConnectedCallback. The OnClientConnected function contains all the different start of game messages to send such as the map, data of the different units, data of the different buildings, data of different tiles, etc., although only the map is sent as a named message. The other data is sent via RPC.

RPCs

Remote Procedure Calls make up the bulk of the communication between the server and the client. There are two types of RPCs, Client RPCs and Server RPCs. The Client RPCs are called by the server, and vice versa.

As mentioned above, the data of the units, buildings and tiles are all sent via RPC. The reason for not sending them via Named Message is because when we attempted to send them via Named Message, there was a problem with the buffer when reading the Named Message. After some frustration and debugging, it was found that the pointer to the fast buffer reader is destroyed before post processing and somehow leads to a null reference error. Although a Named Message usually serves the purpose of one time communication, whereas RPCs are used for short and frequent communication between the server and the client, the Named Message was unable to provide an adequate platform for communication. As such, we used a different strategy to split up each file into chunks and send them via RPC. Although RPC has a smaller message size, by splitting up a message and piecing them back together at the end, large files can still be sent.

```
1 reference
public void CacheFileChunk(RpcMessageChunk chunk, ushort num_to_receive, string file_name)
{
    if (m_files.ContainsKey(file_name))
    {
        if (m_files[file_name].Chunks.Count < num_to_receive )
        {
            m_files[file_name].Chunks.Add(chunk);
        }
        else
        {
            Debug.LogError($"Number of chunks received exceed {num_to_receive}");
        }
    }
    else
    {
        m_files.Add(file_name, new RpcMessageObject()
        {
            Chunks = new List<RpcMessageChunk>() { chunk },
            IsReady = false
        });
    }
    if (m_files[file_name].Chunks.Count == num_to_receive)
    {
        m_files[file_name].IsReady = true;
    }
}
```

Figure 35: Implementation of receiving chunks of large file

However, this method of sending files still has downsides, as after being split up into chunks and sent over, they are not guaranteed to arrive in the same order.

This is due to the Unity Transport Library that Netcode for GameObjects uses. The Unity Transport Library uses User Datagram Protocol (UDP), which does not guarantee that packets will arrive in the same order they were sent. As such, additional work was needed to ensure that all packets were received, then reordered properly. This was implemented in a custom RPC called SendFiles used to split up files into small chunks, then put said chunks back together at the destination using the CacheFileChunk function shown in Figure 35. This SendFiles method belongs to the NetworkUtilities class.

In addition to sending files at the beginning, RPCs are also used in sending information about the player's move on any given round. The commands that a player uses are formatted into a string, which are then passed by RPC to the server. The server then uses a regex match to parse the information from the string and turn it back into different command objects, which can then be executed.

Following the execution of the commands, the server sends back the relevant information after deciding which clients need to have access to the information. This is done by checking whether each unit is visible to each of the other players and if each tile is visible to each player. The information is then again stringified and passed via RPC to the relevant clients. The clients again use regex matches to parse the information and display them on their respective scenes.

Approval Check

The approval check occurs each time a client is about to connect to the game. It is a doorman function that performs a handshake with the server and lets clients in. Similar to the OnClientConnectedCallback delegate provided by Netcode for GameObjects, there is a ConnectionApproval delegate that requires a function to be attached to it.

Our implementation of the ApprovalCheck function is attached to said ConnectionApproval delegate and serves two purposes. The first purpose is to make sure that there is enough room in the game, so that the maximum number of players in a game is not exceeded. The second purpose is to allow a player that has disconnected from the game to be able to rejoin the game where they left. This is a quality of life feature that makes the game less frustrating when players somehow disconnect for unknown reasons.

The implementation of the maximum players in the ApprovalCheck function is done by checking if there is enough room for another player, or if the player is a returning player. Approval is denied if there is not enough room for another player. Each player's data is stored on the server with a unique client ID as well as player name, which can be referenced back if they are rejoining the same server. Since the host client runs the server, the implementation of the host ApprovalCheck skips the host client entirely.

```
2 references
public static void ApprovalCheck(byte[] connectionData, ulong clientId, NetworkManager.ConnectionApprovedDelegate callback)
{
    // TODO FUT. Impl. sanitize the data because it is passed via network
    string player_name = Encoding.UTF8.GetString(connectionData);
    Battle current_battle = Battle.Instance;

    if (clientId != 0)
    {
        bool has_vacancies = !current_battle.IsServerFull;
        bool is_existing_player = current_battle.GetPlayer(player_name) != null;
        bool approve = has_vacancies || is_existing_player;

        if (approve)
        {
            Debug.Log($"Approved connection from client (id: {clientId}, name: {player_name})");
            if (!is_existing_player)
            {
                Player connected_player = new Player()
                {
                    Name = player_name,
                    SerializableColor = (SerializableColor)current_battle.GetRandomAvailablePlayerColor(),
                    Resources = (Resources)current_battle.Rules.StartingResources.Clone()
                };
                current_battle.Players.Add(connected_player);
            }
            current_battle.ConnectedPlayerIDs.Add(clientId, player_name);
        }
        else
        {
            Debug.Log($"Rejected a connection: server is full");
        }
        // note: it creates duplicated player objects on scene, not sure what causing this
        callback(false, null, approve, null, null);
    }
    else
    {
        Debug.Log("Skipped approval check for host client (id: 0)");
        callback(false, null, true, null, null);
    }
}
```

Figure 36: Code implementation of ApprovalCheck function

2.2.4. Models

In order to create an attractive strategy game, visually appealing models and textures are vitally important because they are the objects that players directly interact with. Also, having nice-looking models can give pleasure and enjoyment to the players, which is very important to meeting our objectives.
The implementation of 3D models can be split into 3 parts, namely the creation, textures, and optimization. Some significant techniques used will be discussed in detail in the following section. All models created in Blender were exported as Filmbox files (.fbx) and then imported to Unity.

2.2.4.1. Creation

Modifiers

Blender modifiers are sets of operations which can be applied on a mesh, often replacing the need of repeating tedious operations and saving time in creating the models [13]. The effect of modifiers can be previewed before actually applying them. Using modifiers is one of the essential skills in 3D modeling in Blender. The use of some common modifiers for creating models of this project will be discussed below:

- *Array*: automatic duplication of meshes along axes, around centers of rotations or along a curve

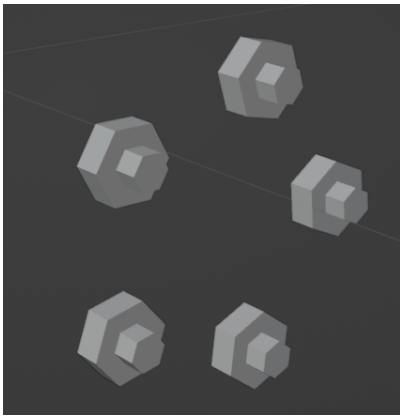


Figure 37: Bolts used on wheels of the utilities model. *Array* was applied on one bolt and rotated 72 degrees around the center of rotation to create 5 bolts distributed evenly on a plane.

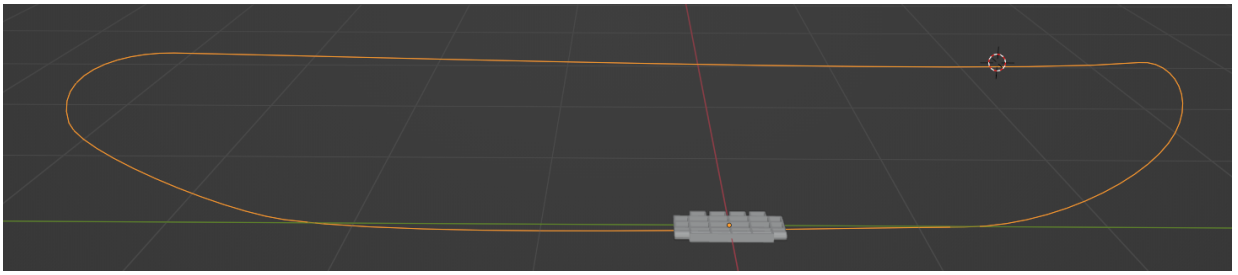


Figure 38: Preparation for creating the link of tank track model: A link piece and a Beizer curve

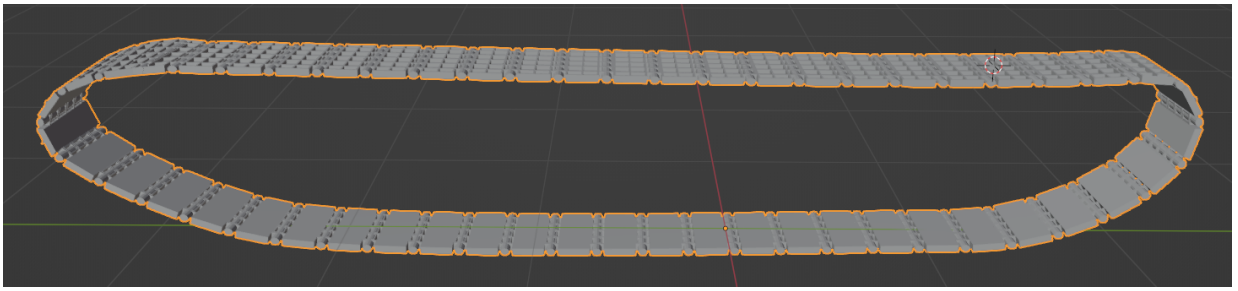


Figure 39: The link created by applying *Array* and *Curve* on the link piece. The Bezier curve in figure 38 was supplied to the curve modifier together

- *Boolean*: meld meshes into one, or create holes on meshes

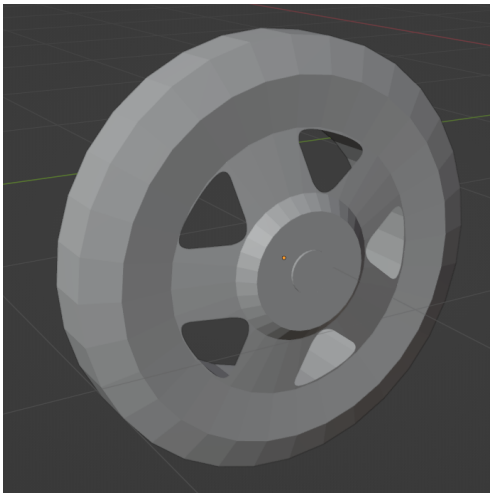


Figure 40: Wheel for utilities unit model. Guitar-pick-shape prisms were created using *Array* (rotation). *Boolean* with “difference” option selected was applied on the wheel with the prisms supplied as the operand object

- *Mirror*: duplicate meshes and reflect them along axes

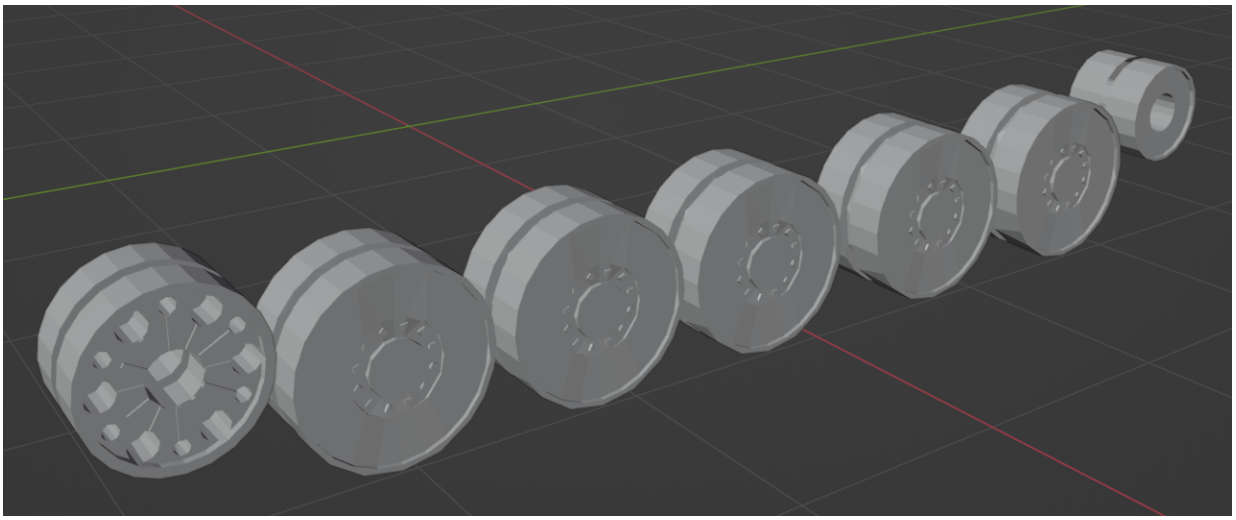


Figure 41: Sprockets and driving wheels of the tank track model. Each of them were created by applying *Mirror* on one half of them

- *Skin*: create surfaces from vertices and edges only

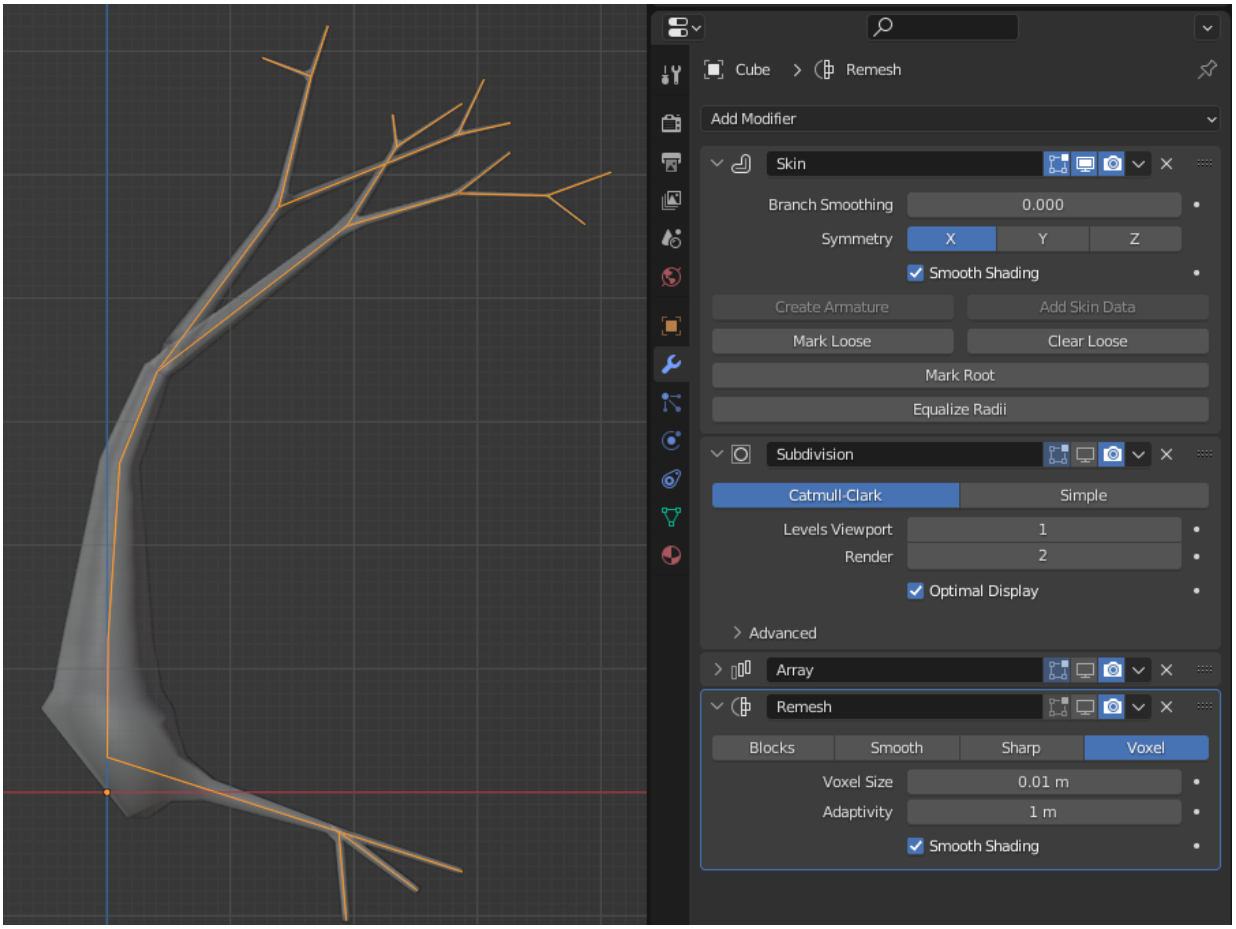


Figure 42: ¼ of the tree trunk model. *Skin* was applied on the highlighted edges to create the mesh. Other modifiers were not rendered.

- *Simple Deform*: Twist, bend, taper or stretch the mesh

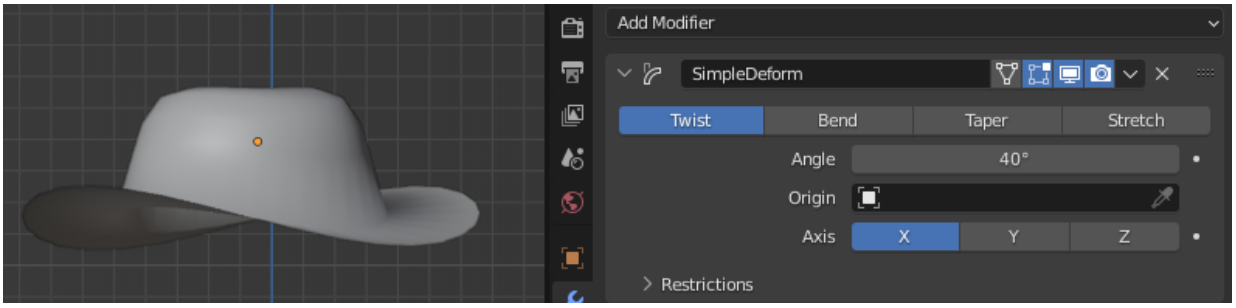


Figure 43: Panama hat used for the Militia unit model by twisting the original mesh for 40 degrees along x-axis

- *Subdivision*: Split mesh surfaces into smaller surfaces
- *Displace*: displace vertices based on textures provided

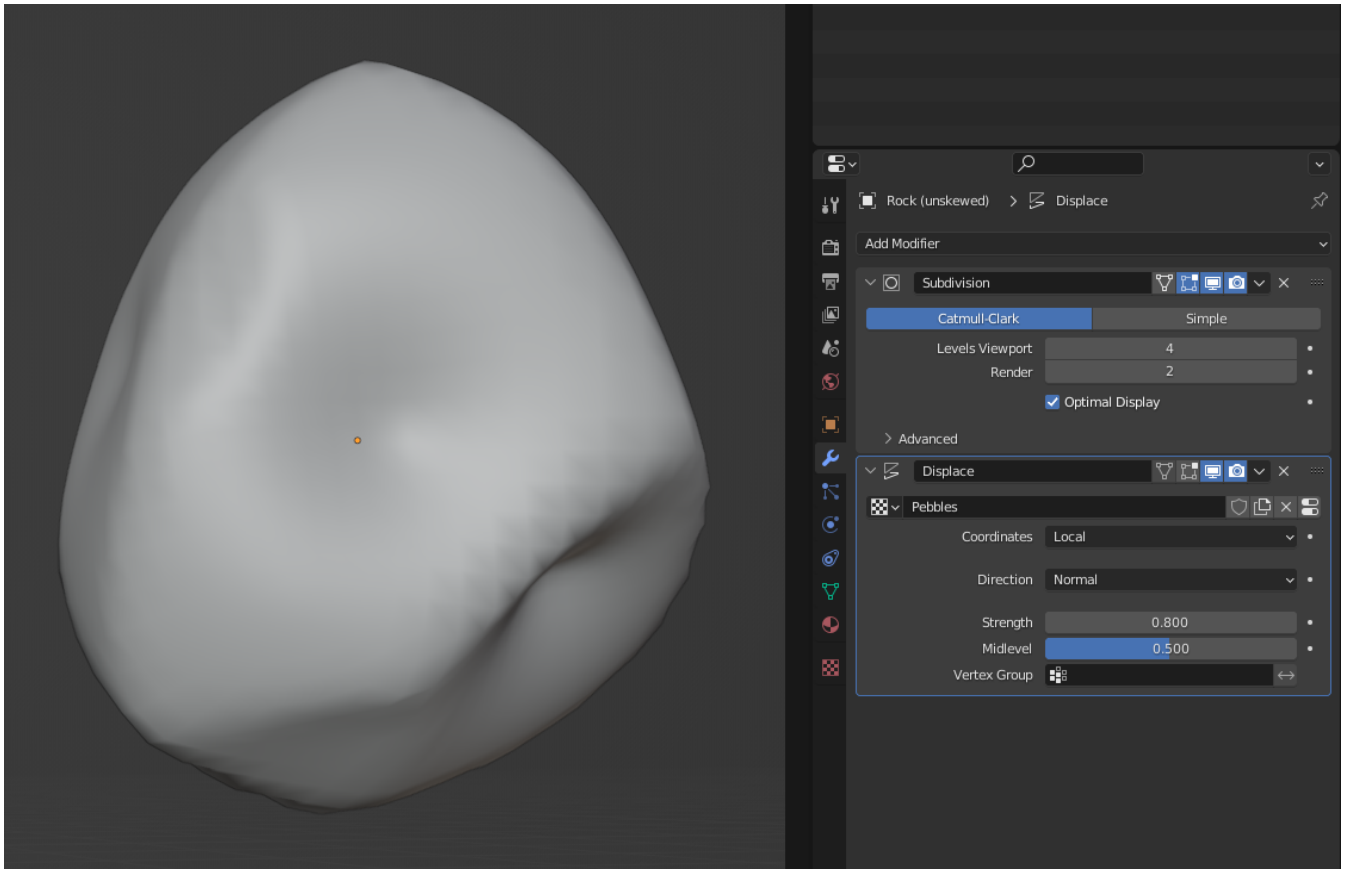


Figure 44: A rock used in the Stream model. Created by applying *Subdivision* and *Displace*. The texture used in *Displace* was a voronoi noise texture generated in Blender

Reference Images

The creation of more complex 3D models for vehicles required frequent uses of the aforementioned modifiers. On top of that, reference images were used for ensuring the models created were in correct ratios. Reference images consist of 2D orthographic projections (top, front and side views) of the objects. The 3D models were then created by matching the overall shapes of individual components in orthographic views in Blender.

Free blueprints online were found and were used as reference images. Some of them are listed here (the colors of original images are inverted for viewing the structures and the outline easier):

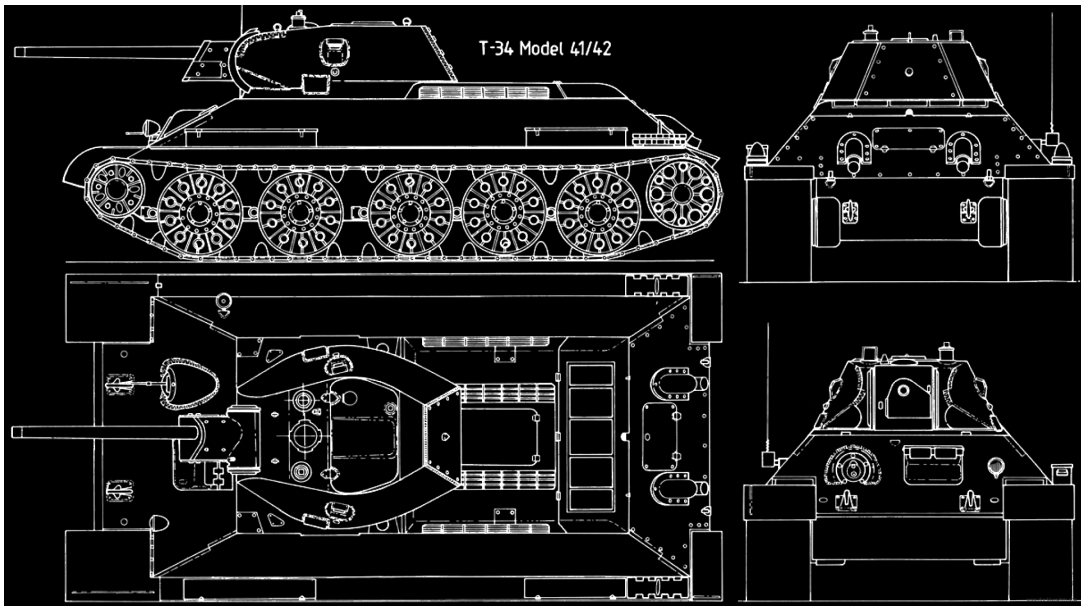


Figure 45: The blueprint of Soviet tank T-34-57, used for the creation of the medium tank unit model

Figure 46: The model created for the medium tank units. Consists of around 25k vertices

Reference images were useful for maintaining the ratio of the models. Yet, solely relying on them was still not enough. During the process of creating the unit models, we found that parts of the objects were blocked in some of the 2D ortho-projections, resulting in incomplete information about the sizes of those parts.

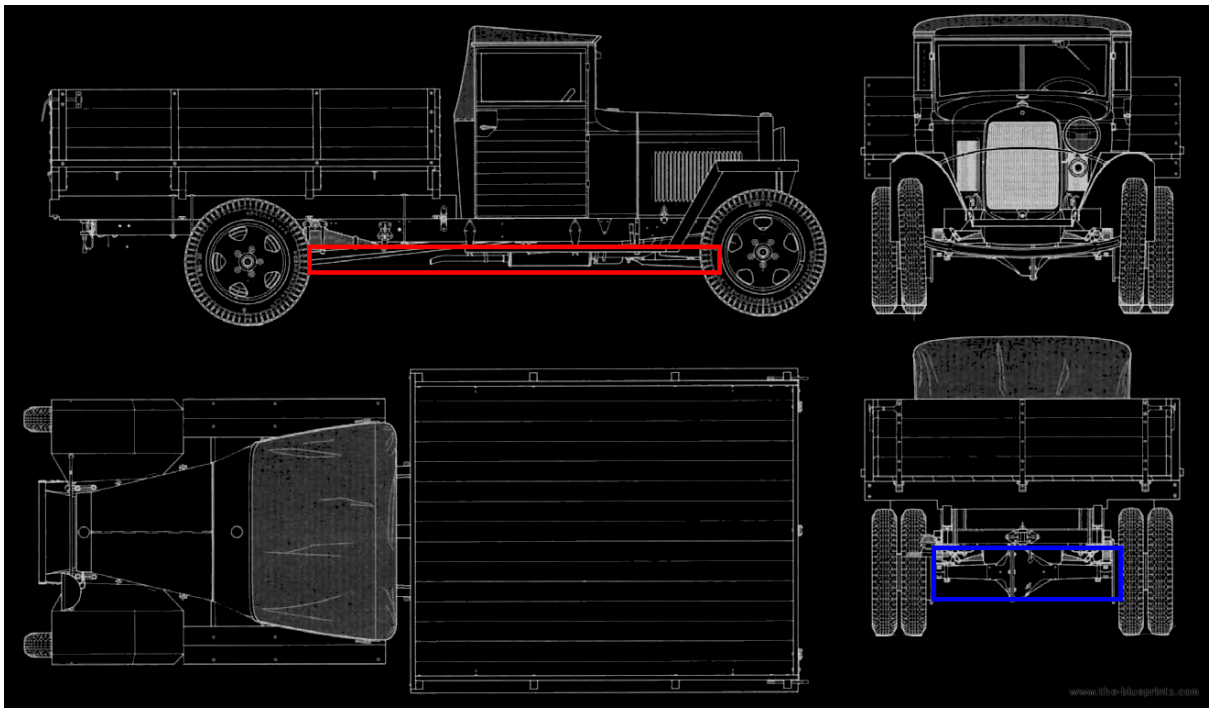


Figure 47: The blueprint of Soviet military truck GAZ-MM (1943), used for creating the utilities unit model. Note that the transmission of the truck (highlighted in red) is completely blocked in the top view (by the compartment and the trunk) and in the rear view (rear drive axle, in blue).

The solution to the above problem was taking a look at images in different perspectives. These images could be found easily on the internet. Yet, as they were all in perspective, they could only serve to assist us to have better insights of the actual shapes of partially blocked parts.

Figure 48: A close-up view of the blocked parts taken from an image online
Source: [14]

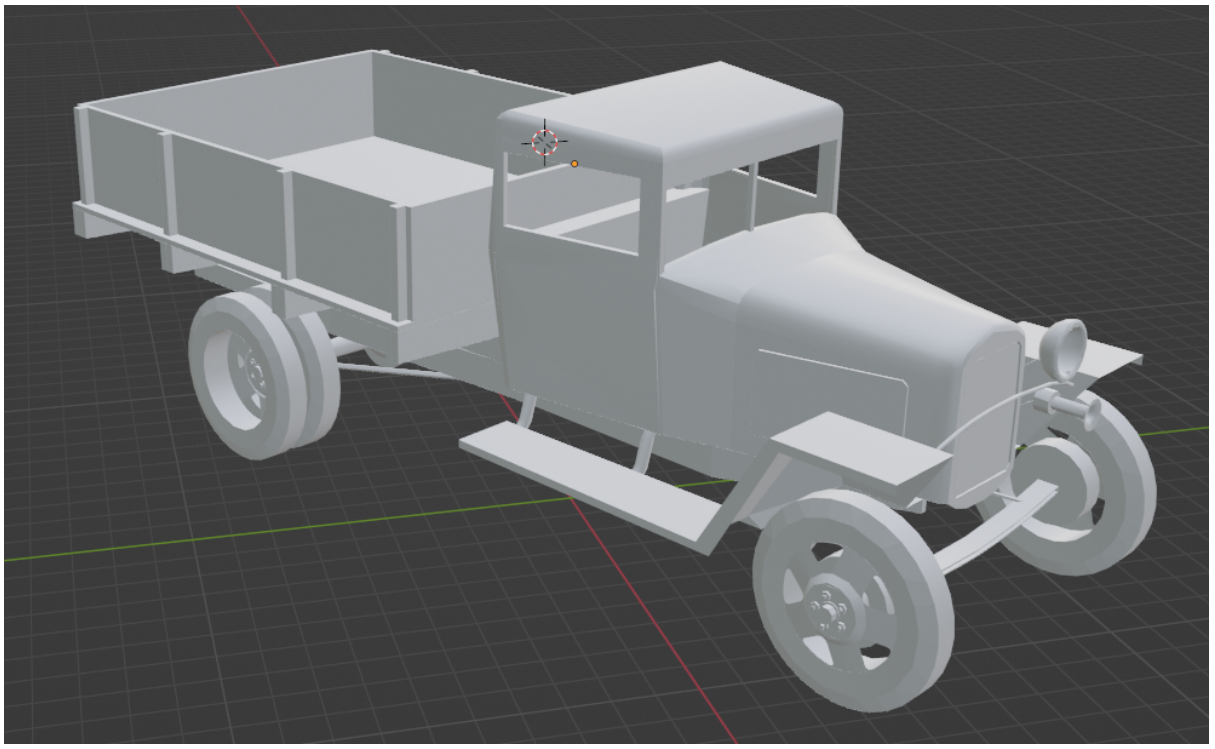


Figure 49: The model created for the utilities unit. Consists of around 8k vertices

Geometry Nodes

Introduced in Blender 2.93, geometry nodes provides node-based operations for changing the geometries of objects [15]. It also allows random placement of objects on a mesh by using the nodes *Distribute Points on Faces* and *Instance On Points*. Instances are not the actual data of objects, but rather references to other objects, or even collections of objects [16]. In our project, this technique was used to create the models of the cities, which consist of different building models. Steps are discussed below:

1) Preparation:
   We firstly duplicated the hexagonal face from our tile 3D model. Geometry nodes would be applied on it.

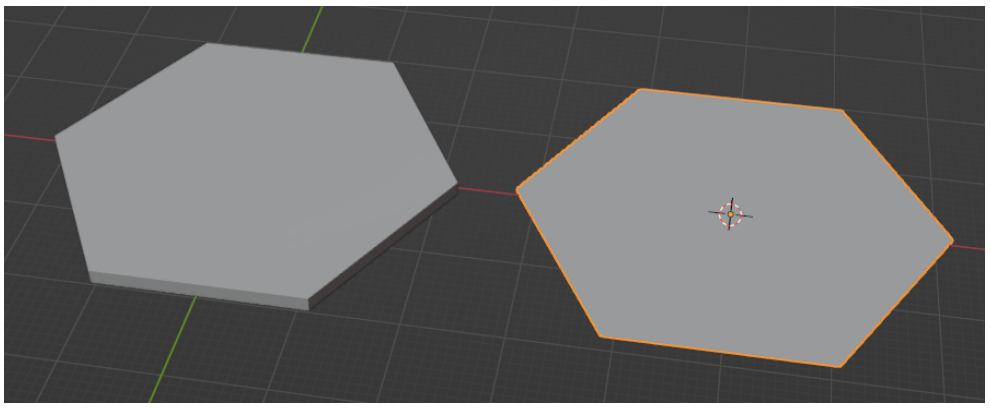


Figure 50: The tile 3D model (left) and the duplicated face (right)

2) Creating buildings variants:
We then created some variants of building models to have more variety and move them into 2 different collections.

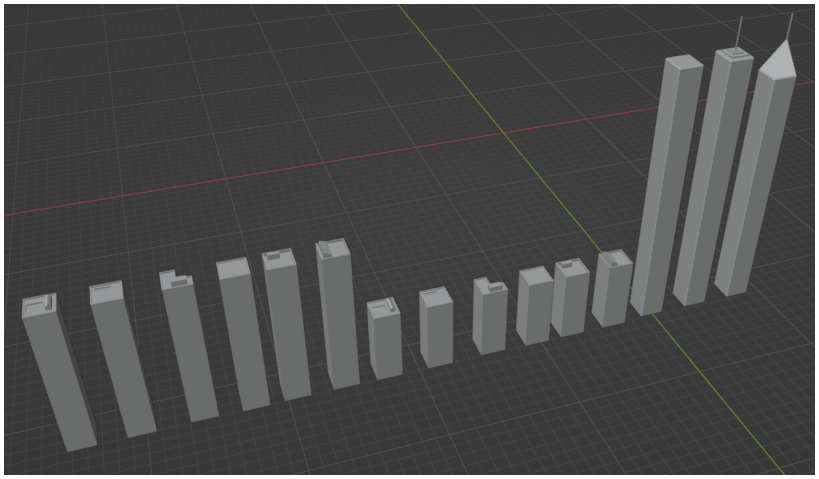


Figure 51: The building variants created by basic mesh operations in blender

Figure 52: The collections of building variants, Buildings_Tall contains the right-most 3 buildings shown in Figure 51

3) Creating the geometry nodes:
Apart from the *Distribute Points on Faces* and *Instance On Points* nodes mentioned above, random value nodes can be added for controlling the scales and rotations of the building models to create even more variety. A portion of the geometry nodes graph is shown below (the complete graph can be found in Appendix D):

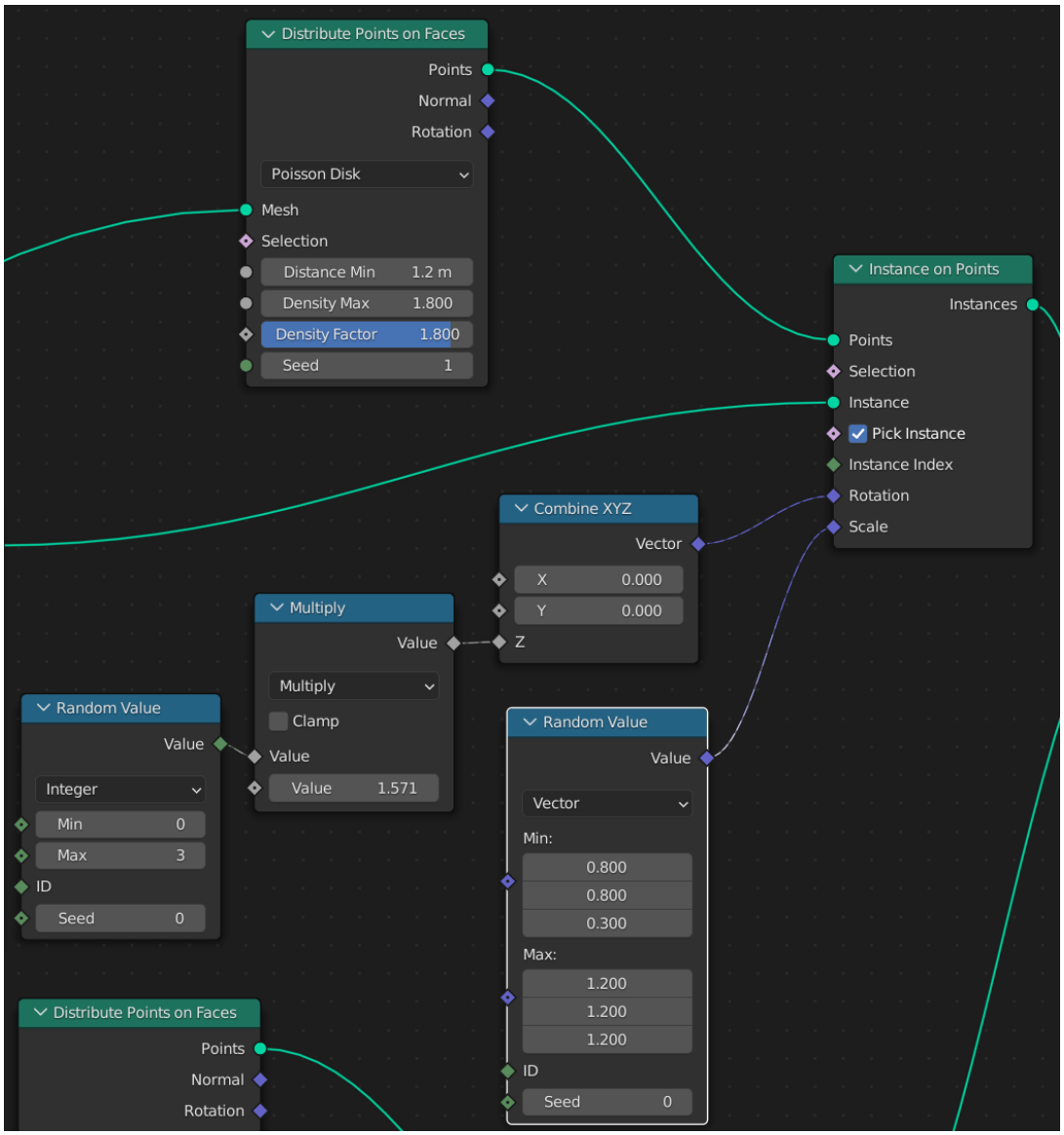


Figure 53: Part of the geometry nodes graph

In this figure, the *Distribute Points on Faces* distributes points on the input mesh by using the Poisson Disk algorithm. The density of points can also be controlled. The nodes connected to the Rotation property of *Instance On Points* restrict the random rotations to either 0, π/2, π or 3π/2 radians along the z-axis. The *Random Value* of type vector connected to the Scale property of *Instance On Points* determines the final scales of the building models. A *Collection Info* node is connected to the Instance property of *Instance On Points*, which is not shown in this figure.

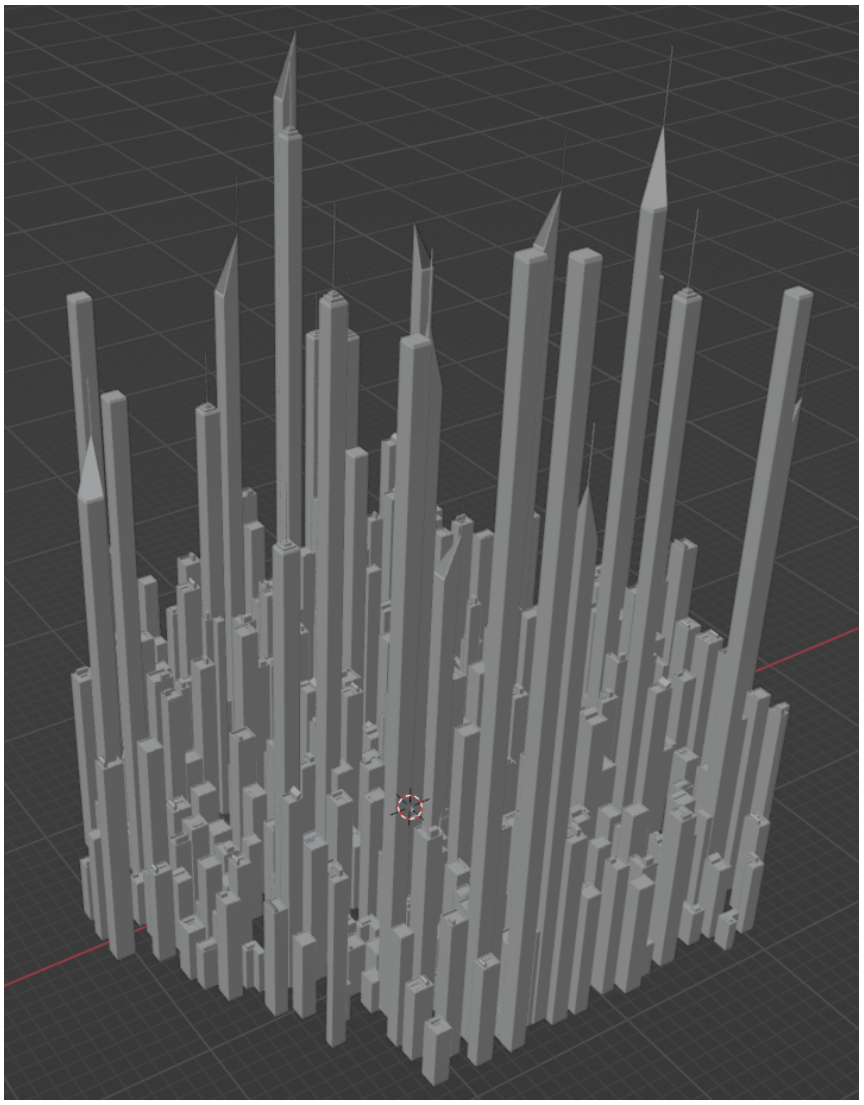


Figure 54: The model generated by geometry nodes for metropolises. Consists of around 20k vertices

2.2.4.2. Textures and Shaders

Textures define the appearance of surfaces of 3D objects. Applying appealing textures to 3D objects can provide aesthetic pleasure to players, which is also important to achieving our project goals. Significant areas of texture implementation will be discussed in this section.

Texture Maps

Seamless texture images could be found online for free. They were used for textures of terrain tiles. Yet, if we directly apply it onto the terrain models, the results would look flat and unreal. In order to produce realistic appearances, different types of texture maps were used. Some of them will be discussed below:

- Bump map

Bump maps store the “height” detail of the texture in grayscale. The brighter the pixel is, the more it appears to be pulling out from the surface, and the darker, the more it appears to be pushing into the surface. Yet, no actual changes would be made on the geometry. In other words, the maps create illusions of depth when applied.

Figure 55: Tree bark texture before applying the bump map (left) and after (mid), and the bump map used (right)

- Normal map

Similar to bump maps, normal maps also create illusions of depth when applied but they work differently. They store orientations of surface normals in RGB format, which correspond to XYZ coordinates in 3D space. The normals are then used in lighting calculations to create bumps on the appearance.

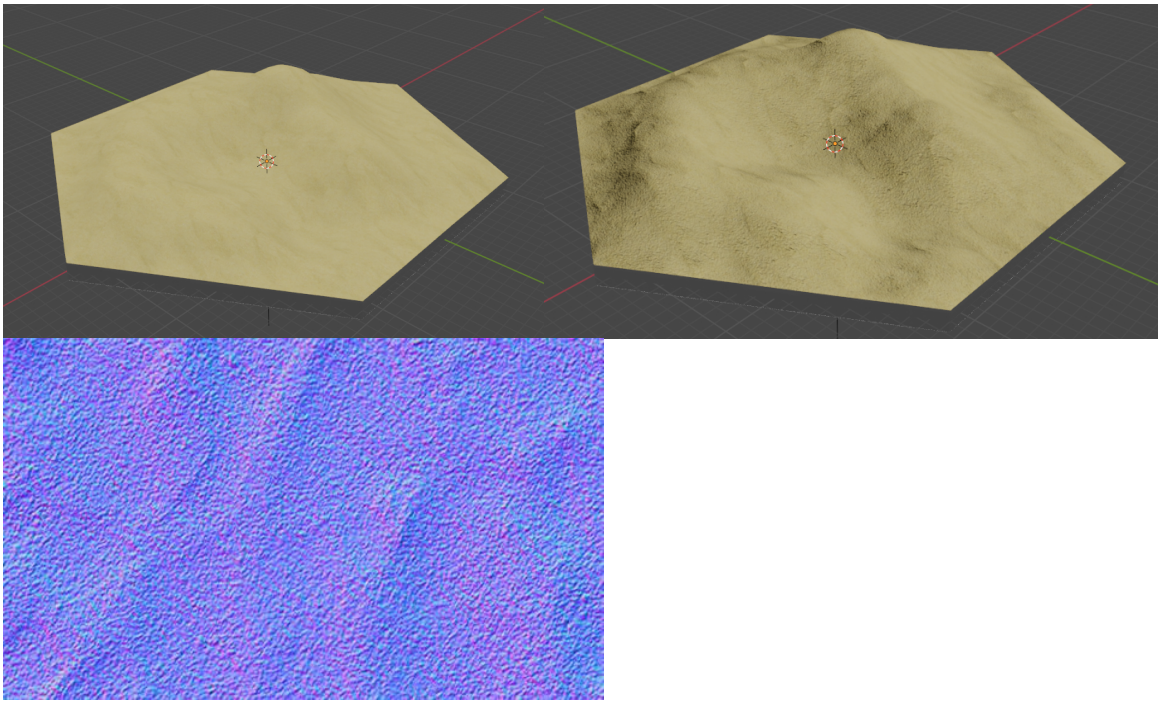


Figure 56: Desert texture before applying the normal map (upper left) and after (upper right) and the normal map used (down)

Basic Shaders

Shaders describe how lightrays interact with surfaces of models or volumes [17]. In Blender, *Shader Nodes* are used for creating shaders. Unity provides convenient features in the editor for creating basic shaders. Yet, the shaders created by both softwares are incompatible. It is because the underlying mathematical functions used for computing the shaders are different in these two softwares. However, *Shader Nodes* in Blender can still serve as previews of final rendering results in Unity because rendering engines in both softwares support Physically-based Rendering (PBR).

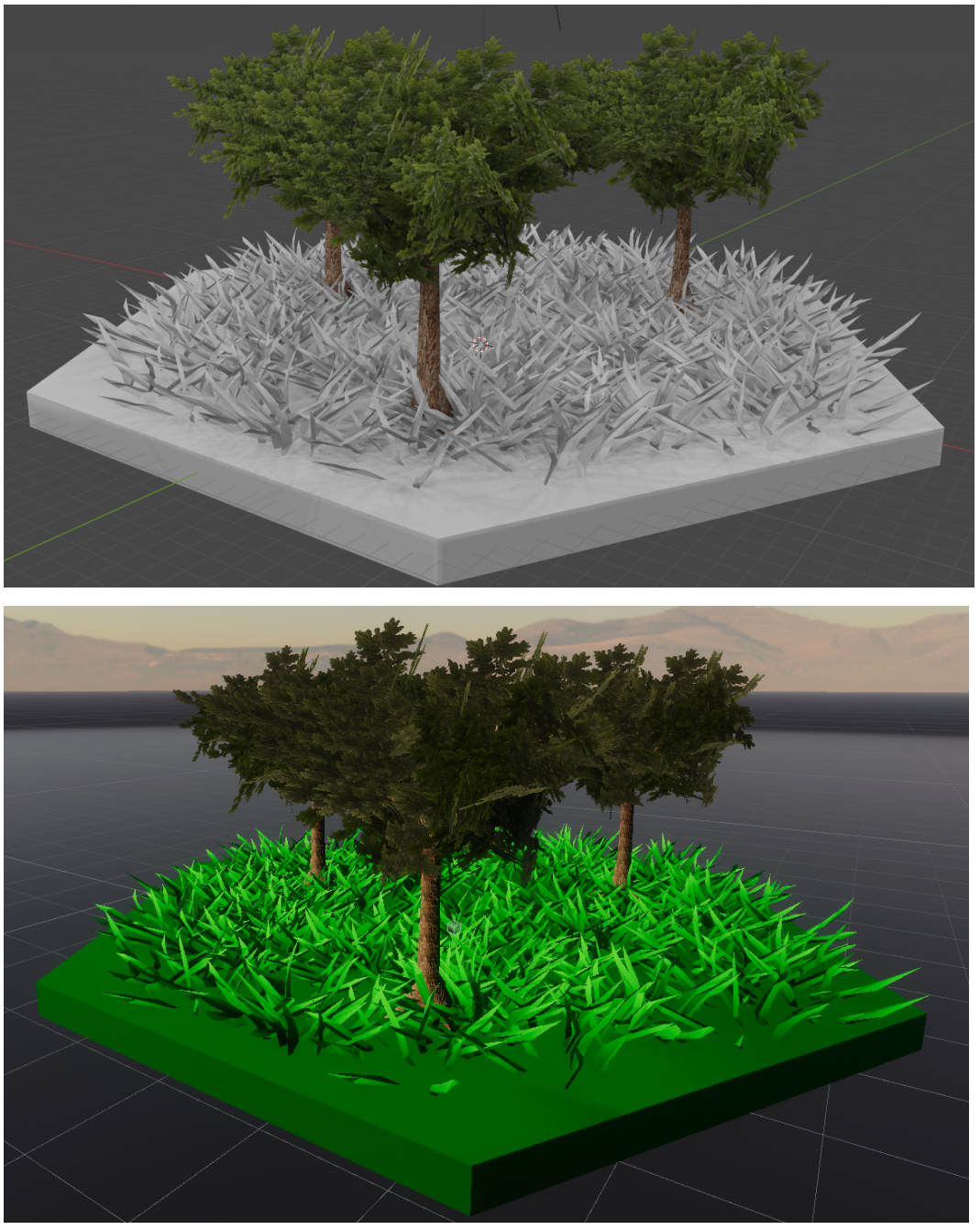


Figure 57: Comparison between the rendering results of the tree models in Blender (upper image) and Unity (lower image). Shader Nodes used for leaves and tree trunks in Blender can be found in Appendix D

Custom Shaders

Unity provides one slot for the base map (texture image) by default. Yet in our project, there was often a need to blend multiple textures together, or having gradient changes of colors, to achieve the desired effect. In short, custom shaders were needed. Unity offers a feature called *Shader Graph* for users to create custom shaders on their own, which in turn can be used in this case. Some examples will be discussed below:

For the Mountains tile, we would like to have snow patches on the rocky surface. Few steps are required to produce this effect:

1) Prepare the model:
   We firstly created the Mountains model in Blender

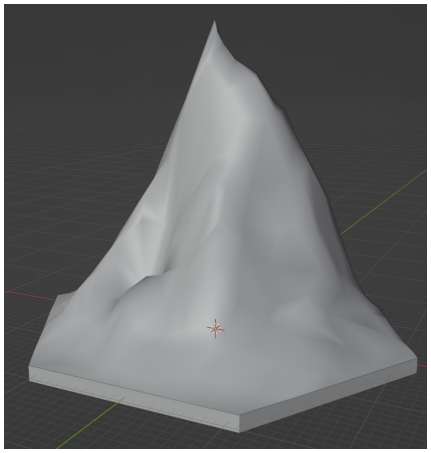


Figure 58: The Mountains model. Consists of around 600 vertices

2) Create the *Shader Graph*:
   In order to instruct Unity about how the rock texture and the snow texture should blend together, the *Lerp* node was used. The time input (T) of the *Lerp* node can be determined by generating simple noise values and a transition value. Some operations on these values were needed as well. The portion of the *Shader Graph* controlling the T input of *Lerp* is shown below (complete graph can be found at Appendix D):

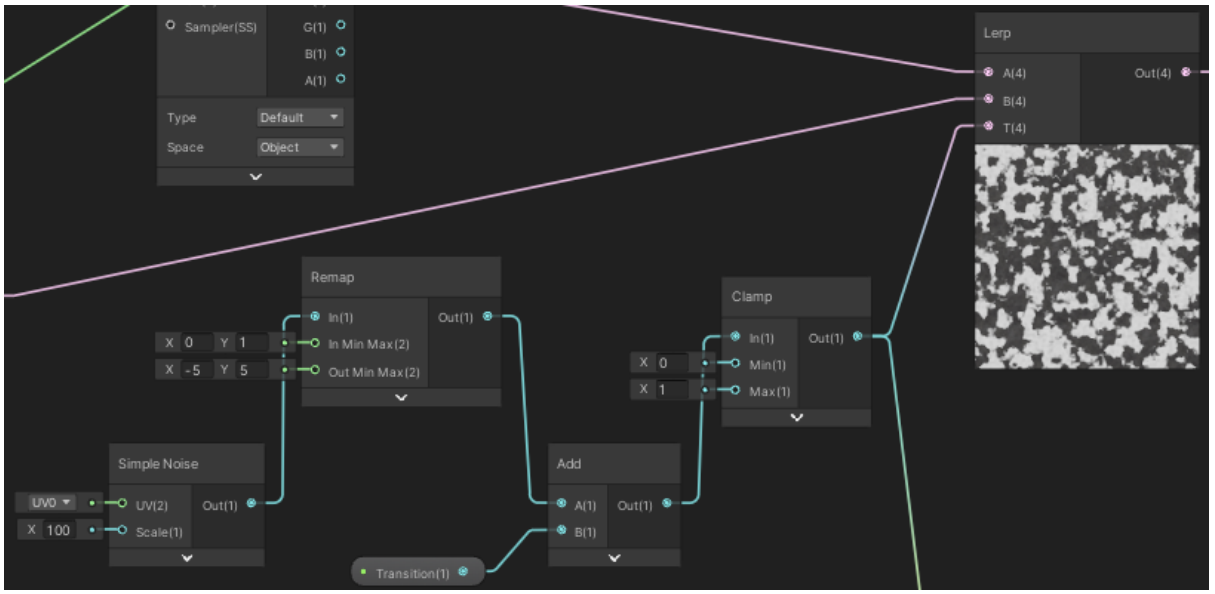


Figure 59: The part of the Shader Graph which controls the T input of Lerp. A and B input of Lerp are connected to the rock and snow textures respectively, and the output of Lerp is connected to base color input of the shader master node, which are not shown in this figure

Simple Noise, *Remap*, *Add* and *Clamp* nodes were used. The transition value was exposed as a property for changing it in the editor. These 4 nodes have the following functions:

- *Simple Noise* : generate noise values based on the input UV (in this case, the UV of the model)
- *Remap* : increase contrast of the values generated by *Simple Noise*
- *Add* : self-explanatory
- *Clamp* : clamp the result from previous nodes to 0 to 1

It is worth-noticing that if negative values were fed to T input of *Lerp*, some part of the texture would turn black, and if values over 1 were fed, some part would be overly-exposed. Therefore, the *Clamp* node is necessary.

Apart from lerping the texture images, the normal maps have to be lerped as well. This was done in a similar way except connecting the *Lerp* output to the normal input of the shader master node.

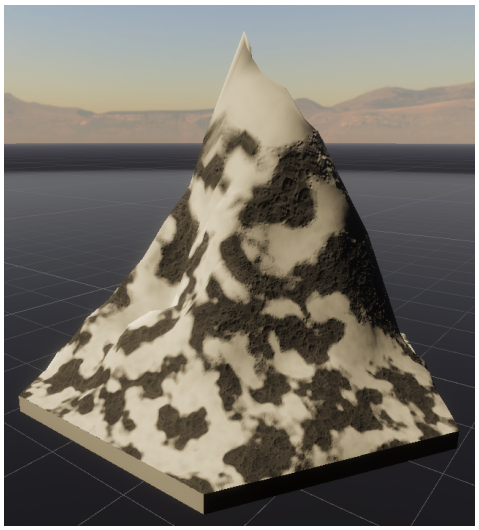


Figure 60: The Mountains model with the custom shader applied

2.2.4.3. Optimization

When we added more details to object models, their polygon counts would increase. Yet, this would bring extra burden to the GPU and probably results in a drop in frame rates. Having low FPS (frames per second) count would definitely bring an unpleasant experience to the players and ultimately undermine our goal. So, optimization is vital even during the implementation process.

Level Of Details

Level of Details (LOD) is a technique to reduce GPU load by dropping unnecessary details which do not cause significant change in the appearance of the objects. When an object is far away from the camera, the apparent details of the object decrease. However, Unity renders the object the same way it does when the object is close to the camera, resulting in a waste of GPU load.

In our project, this technique was applied to the model for grasses. To automatically reduce polygon counts, the ***Decimate*** modifier in Blender with different ratios was applied on the original (LOD 0) model.

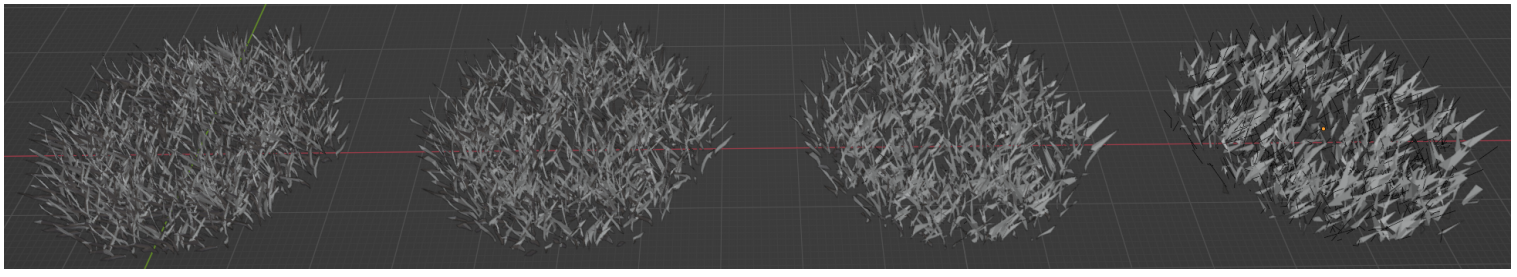


Figure 61: Grasses models for different LOD levels (left to right: 0 (14k verts), 1 (10k verts), 2 (6k verts) and 3 (1.4k verts))

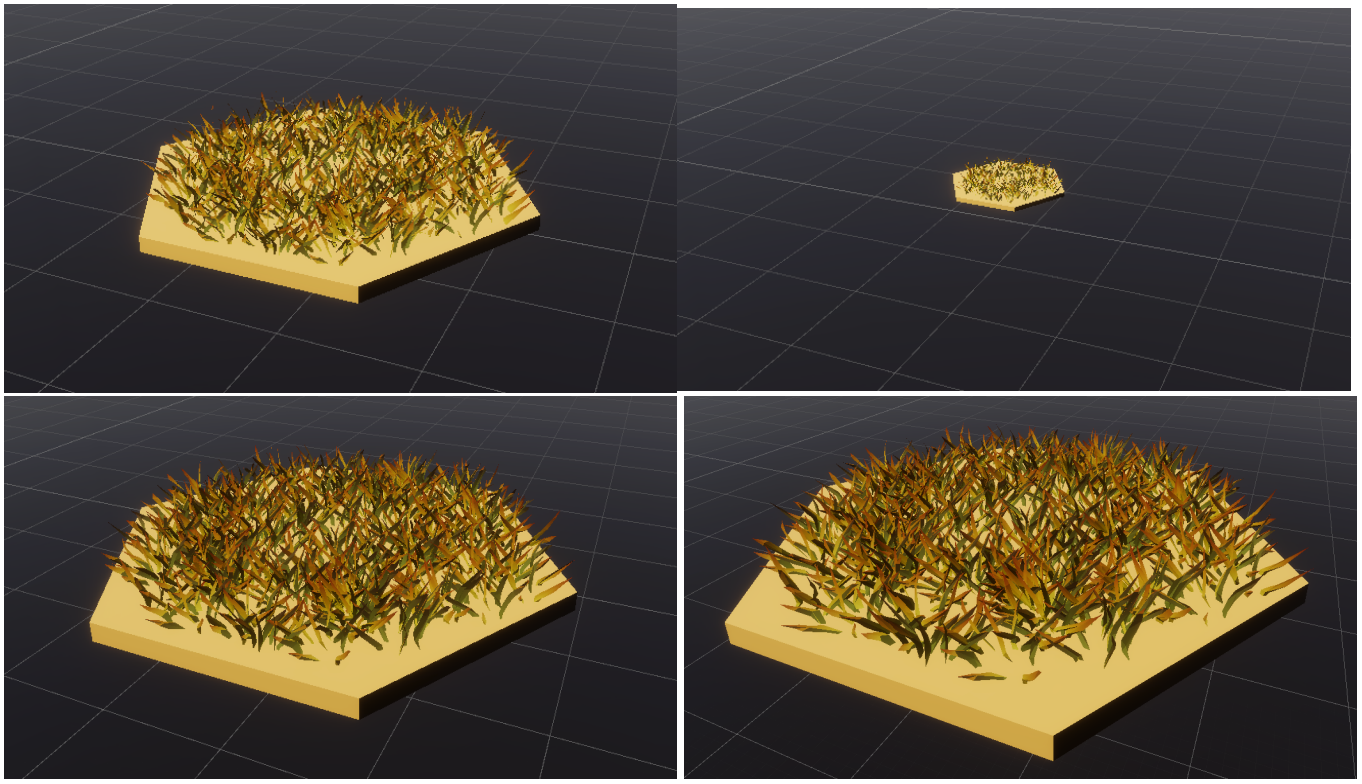


Figure 62: Plains models at different distances in Unity from near to far

2.2.5. Data Handling

In order to provide modability on values of attributes, all of the data storing the values are (de)serialized during runtime and saved as Json files, instead of hardcoding. The *Data<T>* generic abstract class was implemented to provide the structure of data and custom Json converters were also implemented for (de)serialization purposes.

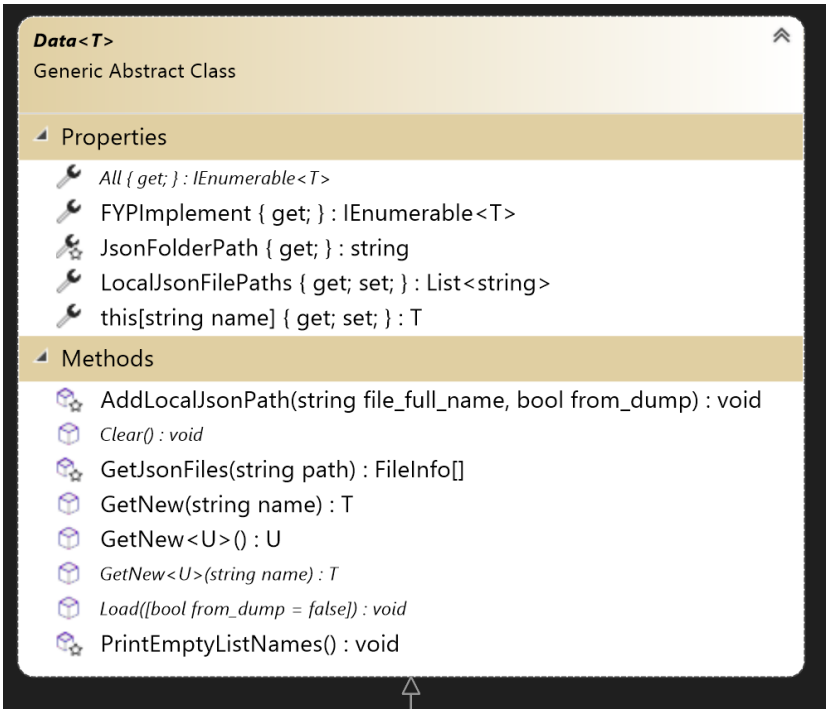


Figure 63: The Data<T> class. Full class diagram can be found at Appendix C

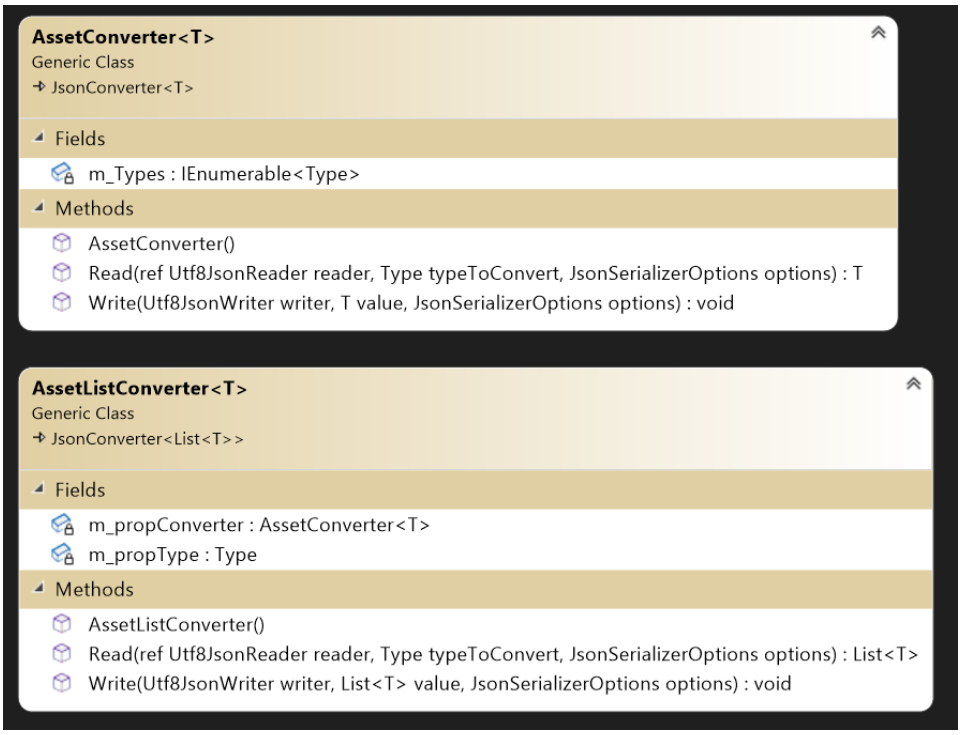
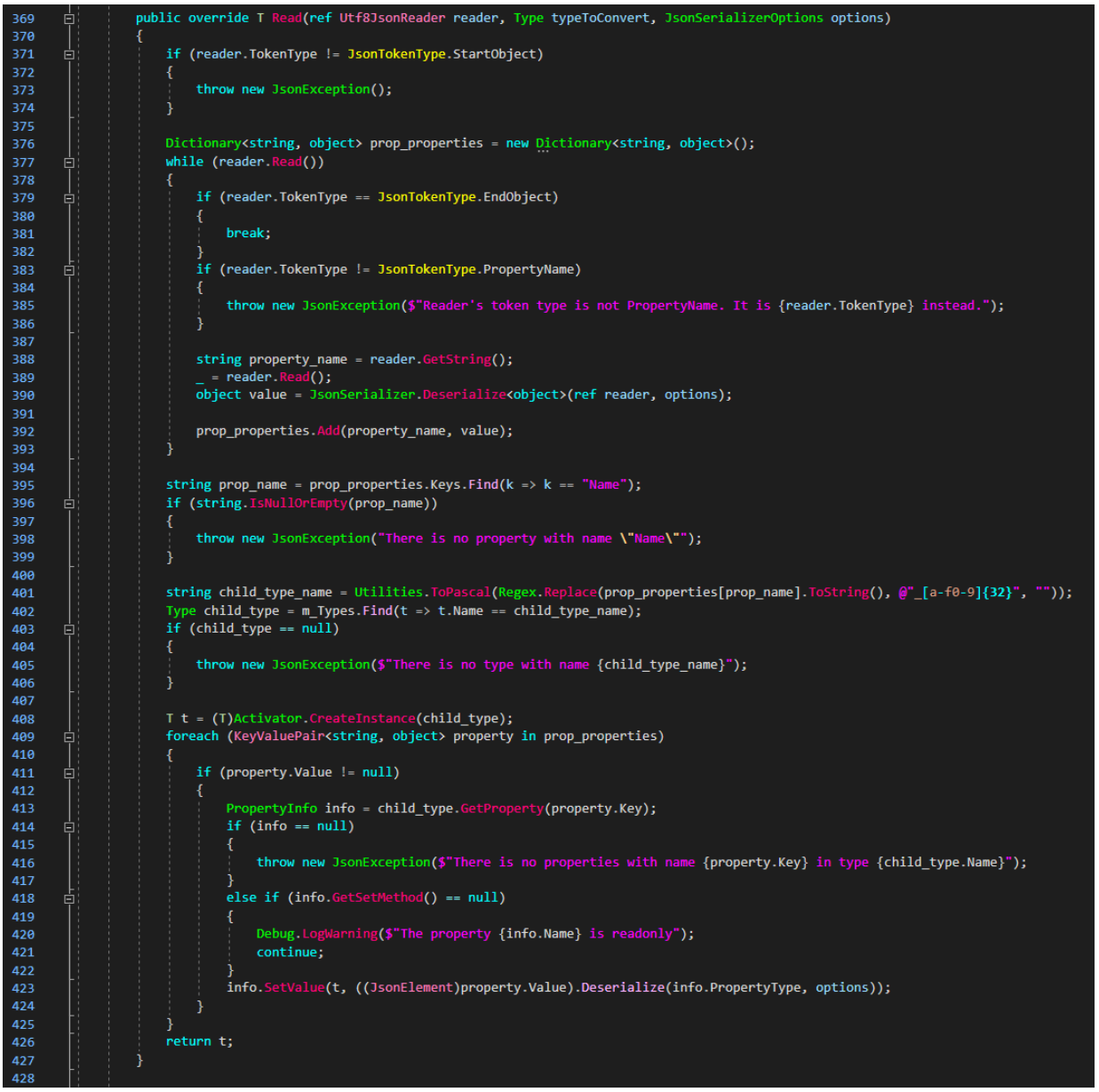


Figure 64: Custom Json converters for asset (de)serialization. See Appendix C

There are a few classes inheriting the *Data<T>* which are the *UnitData*, *BuildingData*, *TlleData*, *CustomizableData*, *ModuleData*, and *GunData*. Each of the classes contains several lists of their corresponding sub-types. For example, *UnitData* contains *List<Personnel>*, *List<Artillery>* and *List<Vehicle>*. Yet, all these three sub-types are abstract classes which constructors cannot be utilized by the default Json deserializer provided by *System.Text.Json* library. So a custom Json converter was needed for this case, or more generally, for all assets.



```
public override T Read(ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
{
    if (reader.TokenType != JsonTokenType.StartObject)
    {
        throw new JsonException();
    }

    Dictionary<string, object> prop_properties = new Dictionary<string, object>();
    while (reader.Read())
    {
        if (reader.TokenType == JsonTokenType.EndObject)
        {
            break;
        }
        if (reader.TokenType != JsonTokenType.PropertyName)
        {
            throw new JsonException($"Reader's token type is not PropertyName. It is {reader.TokenType} instead.");
        }

        string property_name = reader.GetString();
        _ = reader.Read();
        object value = JsonSerializer.Deserialize<object>(ref reader, options);

        prop_properties.Add(property_name, value);
    }

    string prop_name = prop_properties.Keys.Find(k => k == "Name");
    if (string.IsNullOrEmpty(prop_name))
    {
        throw new JsonException("There is no property with name \"Name\"");
    }

    string child_type_name = Utilities.ToPascal(Regex.Replace(prop_properties[prop_name].ToString(), @"_[a-f0-9]{32}", ""));
    Type child_type = m_Types.Find(t => t.Name == child_type_name);
    if (child_type == null)
    {
        throw new JsonException($"There is no type with name {child_type_name}");
    }

    T t = (T)Activator.CreateInstance(child_type);
    foreach (KeyValuePair<string, object> property in prop_properties)
    {
        if (property.Value != null)
        {
            PropertyInfo info = child_type.GetProperty(property.Key);
            if (info == null)
            {
                throw new JsonException($"There is no properties with name {property.Key} in type {child_type.Name}");
            }
            else if (info.GetSetMethod() == null)
            {
                Debug.LogWarning($"The property {info.Name} is readonly");
                continue;
            }
            info.SetValue(t, ((JsonElement)property.Value).Deserialize(info.PropertyType, options));
        }
    }
    return t;
}
```

Figure 65: The *Read* method for AssetConverter<T>

In the *Read* method of the *AssetConverter<T>* class, instead of the ordinary approach used in the default Json deserializer, which reads the properties one by one, all properties were read and cached temporarily. The actual derived asset type was resolved by reading the *Name* property. The actual type names were not stored in Json. Therefore some string processing operations were needed for resolving the actual type correctly. After resolving the type, a new instance of the asset object was created by *Activator.CreateInstance*, instead of calling the constructor directly because the type was unknown and was resolved at runtime. The properties of the asset object were then set by using methods provided by *System.Reflection* library.

2.3. Testing

In order to provide an optimal user experience, we want to minimize the occurrence of bugs. As such, we performed unit testing on the codebase to try to ensure that the code works as intended. In this project there is a very large codebase with many methods in many classes. We performed unit testing on two major parts of the code: Asset Loading and Game Logic.

Using Unity's Test Runner, these unit tests can be written and run in the client with a helpful GUI.

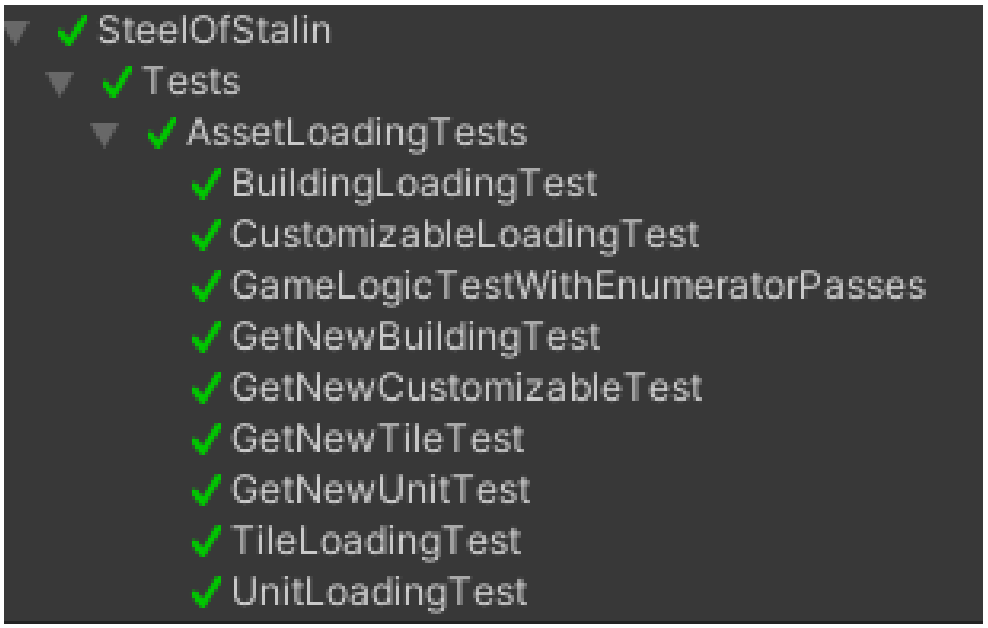


Figure 66: Unity Test Runner

Asset Loading Tests
This section of the tests are mainly to ensure that different assets can be loaded properly, even during game time. This will make sure that the methods used to create a unit behave as intended when called in different areas of the code, whether that may be for player or AI training units, creating buildings, or map generation, etc.

Test Case	**Passed**
Units can be loaded	✅
Buildings can be loaded	✅
Tiles can be loaded	✅

Customizables can be loaded	✅
Random new Unit Data can be gotten	✅
Random new Building Data can be gotten	✅
Random new Tile Data can be gotten	✅
Random new Customizable Data can be gotten	✅

Table 7: Asset Loading Test Cases

Game Logic Tests

This section of tests will ensure that the different methods used by the game and mainly the map class are working as intended. Many of these methods rely on the tests done in the asset loading tests, thus the asset loading tests were done prior to doing game logic tests.

Test Case	**Passed**
Map can add units	✅
Map can add/remove buildings	✅
Map can get specific props at a given coordinate	✅
Map can get tiles	✅
Map can get all props of a given type	✅
Map can get cities	✅
Map can return correct owner of a given unit/building	✅
Units on a Map can return the correct tile they are on	✅
Tiles can return correct neighbouring tiles, with/without the same types	✅
Units can locate the correct path to another tile	✅

Table 8: Game Logic Test Cases

2.4. Evaluation

To evaluate this project, we will mainly compare the objectives that we set at the beginning of this project to the results that we have created.
Our first goal was to create an in-depth logistic management system that allowed the player to make tactical decisions. Our group was able to accomplish this goal by implementing mechanics in the game such as the supplies attribute on each unit. Using commands such as suppression, the player can make decisions that make use of the logistic system to run enemy units out of resources. The logistics system is a core part of the gameplay loop, and as such meets the first goal.

Our second goal was to create an attractive strategy game. Through UI implementation, camera control, and various other strategies, we have created a game that has acceptable graphics (especially since the assets we used are original). Thus, the second goal is met.

Our third goal was to create a multiplayer game. Using Netcode for GameObjects, a connection was able to be created between a host and a client. The client is able to send commands to the host via RPC and the host is able to send information back to the client. These are the fundamentals of a multiplayer game, and though there are other safety measures to implement, and more to be done to connect via Internet, the rudimentary structure of a multiplayer game is created and can be played locally. Therefore, the third goal has been met.

Our fourth goal was to have the game run smoothly. The smoothness of the game is acceptable, as it can run smoothly as a single instance. Though there are some hardware requirements (see section 6.1), the game will run smoothly on most modern devices. Much optimization was done regarding the polygon counts in the models, as well as the rendering. As such, the fourth goal is met.

Although we would have liked to do some UAT, we were unable to do so under the time constraints.

3. Discussion and Future Works

3.1. Map Generation

River distortion

As found out in Section 2.2.1, the rivers generated by the existing solution were spaghetti-like and they aren't really appealing. Some distortions can be added by using 1D Perlin noise. Yet, as the coordinates of tiles are integers, rounding off the distorted coordinates of rivers may create fragments of rivers or zigzag-like patterns which are also undesirable. Further research and investigation on how to overcome this problem will definitely be an interesting topic.

River source picking

The existing way of picking river sources is simply picking random highland tiles. A more advanced method can be used: Firstly, detect the number of highland patches in the map by using the DFS algorithm. Then foreach patch, pick one river source from the patch. The reason why this will be a better method is that the existing method sometimes generates rivers that are concentrated in a small area of the map, because the river sources are very close to each other. The new solution can ensure certain separation of the river sources, because highland patches are much less likely to be closely packed.

Seeded Perlin & Problem with Mathf.PerlinNoise

As mentioned in Section 2.2.1, we used the *Mathf.PerlinNoise* function from Unity and do not have the control of the permutation table. The mitigated solution used right now does not create enough variety if the "seed" provided changes a little, as the "seed" is just an offset value. Also if the "seed" is too large (more than 7 digits to be exact), the *Mathf.PerlinNoise* function will generate the same result for all sampling points, resulting in a monotonous grayscale texture when visualized. The cause of the problem lies in the fact that *Mathf.PerlinNoise* intakes the coordinates of type floating-point single precision, which has an accuracy of around 7 to 8 digits. The details of the problem is shown in the below figures:

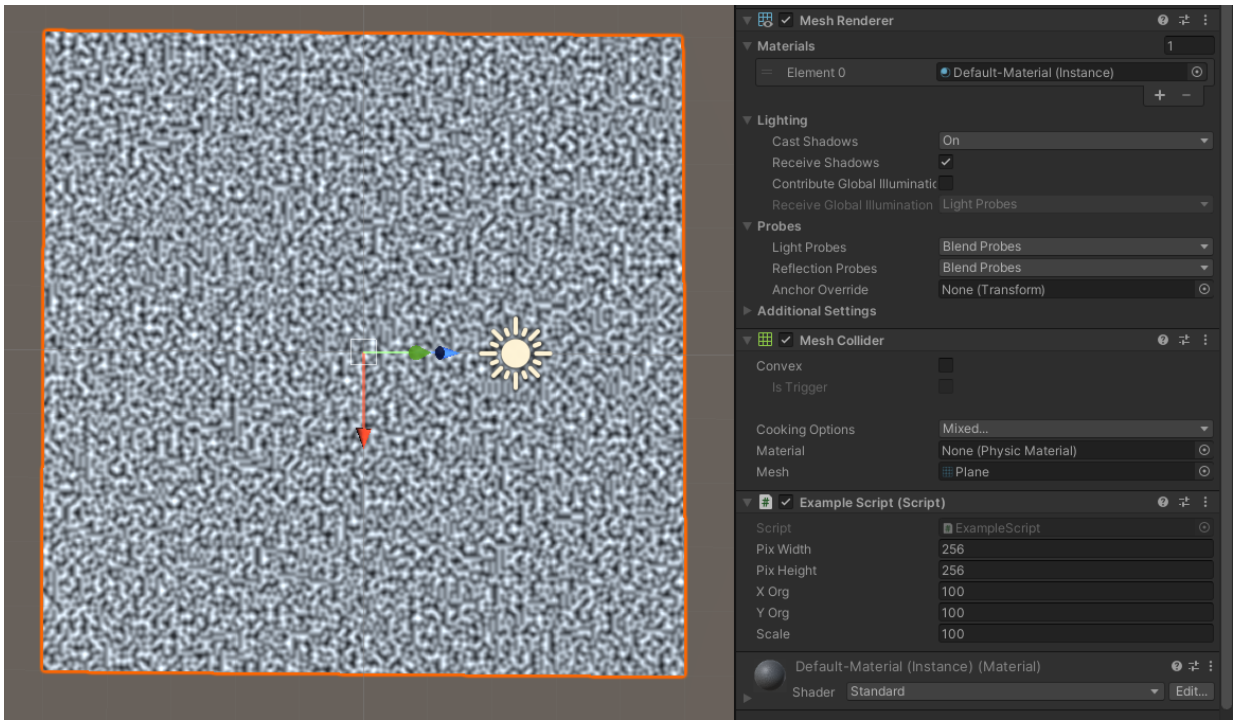


Figure 67: Normal behavior resulted if X Org and Y Org (both used 100) are small

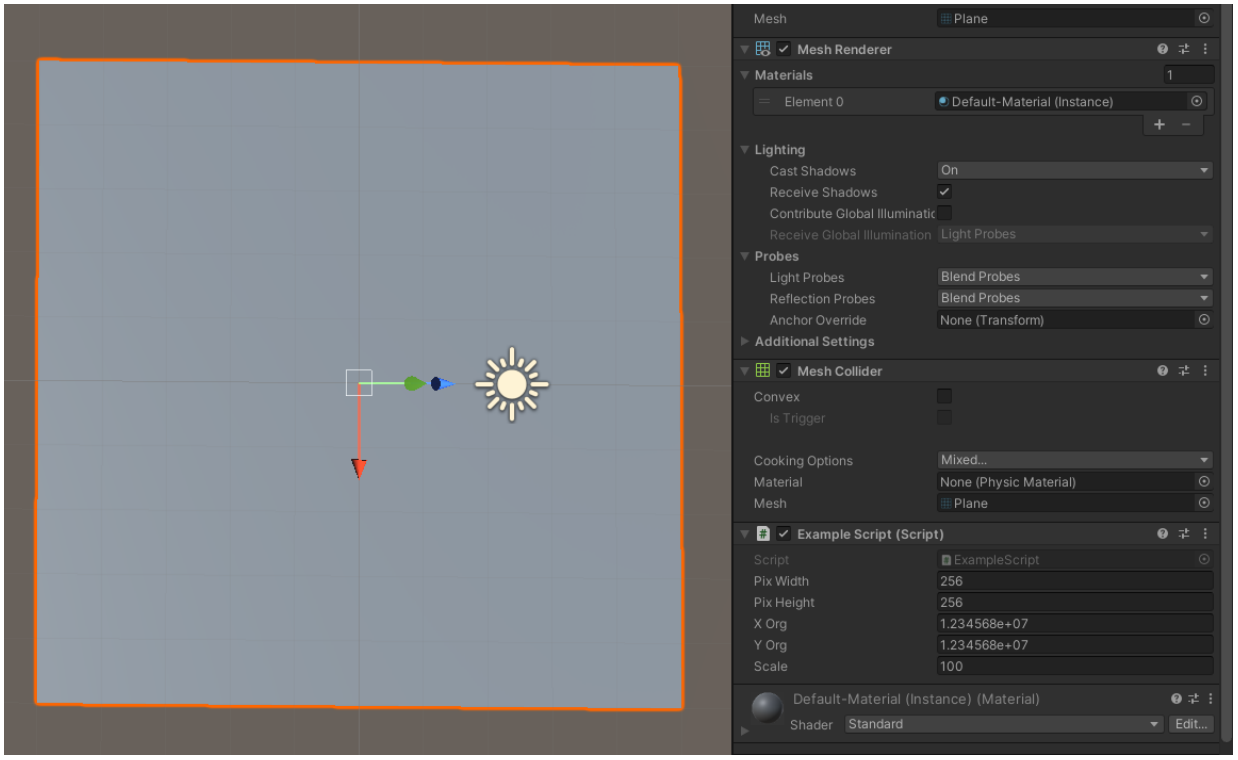


Figure 68: A monotonous grayscale texture generated if X Org and Y Org (both used 12345678 for testing) are too large

The code of *Example Script* was adopted from Unity Manual directly without any modification [18]. Yet, it demonstrated the aforementioned problem. The mitigated solution was to restrict the "seed" to less than 7 digits. Yet for future works, we will definitely need our own implementation of the Perlin Noise algorithm with an actual seed and intake coordinates of type floating point double-precision.

3.2. Gameplay

Further Modability
Currently, we implement the game logic using abstract classes so that any type of props would need to inherit current base classes with all methods being properly implemented. This class hierarchy allows the game logic to be concise and tidy when the amount of content scales. In the future, we will work on implementation of building handful tools for players to add custom models and prop logics to provide players with modability with the hope of bringing the game a high replayability.

More game modes, mechanisms
Historical mode and other game modes are on the roadmap of further development. We chose to drop this part in this project due to the enormous amount of models, sound effects and logics that need to be built on top of the current code base. However, we have done our best to build a base of the game that allows these new features to be included without a need of refactoring the whole code base.

Bot Algorithm Improvements
Currently, the bot algorithms that we implemented inside the game can still be improved especially with the amount of unknown variables that can happen inside the game. With more Advanced algorithms, we can add an additional difficulty for the bots where the user can choose from. With this, players can improve their skills a lot when playing single player mode.

3.3. Multiplayer

Currently, the multiplayer implementation is rather elementary, and does not contain many higher-level features. Further implementation of various higher-level features such as security, data integrity, and connection of the real network are desirable.

Security
As shown in the implementation of the ApprovalCheck function in Figure 36, there is further implementation to be done in sanitizing the data before sending it over the network, to guarantee that no residual data can be recovered. This will enhance the security of the data transfer between the server and the client.

Data Integrity
Another feature that can be implemented in our project is to ensure the integrity of the data. By using a checksum for the files that we send, and checking it at the end after reception of the data, we can ensure the integrity of the files will be preserved throughout the transmission. This will protect the data sent from tampering, and attempt to thwart any attackers that may try to change the data as it is being sent.

Connection over Real Network
Though the connections currently are local to facilitate easier testing, and therefore not exposed to external networks, it is a future goal to achieve connection over the internet rather than just local play. To achieve this goal, we will need to write code for passing through firewalls. In addition, other features such as password checking will ensure that players will not be able to join random networks at will. This password checking will be passed on the connectionData parameter to the ApprovalCheck function, where it will confirm the password of the server. This will allow for lobby privacy when games are open to connection over public networks.

3.4. Models

Texture painting
Right now, the unit models are much more complex than terrain models and they are without textures. It is foreseeable that free textures of complex models would not be available online. Yet, adding textures to those unit models will definitely enhance player experience. To do so, we have to explore more on an advanced technique in Blender called Texture Painting. It allows model designers to paint on the polygons directly when the model is UV unwrapped.

Animations & Visual effects
As the players are taking simultaneous turns in our game, the timing of playing the animations of units will be an interesting topic to explore: if we play all the animations at round start to show the results from the previous round, then it may be too overwhelming to the players. Yet, if we play them right after the players assign the command to the unit in the Planning phase, the animations will not be reflecting the actual changes, as the command is not executed yet.
We had an in-depth discussion for this part before but no conclusion was drawn and we decided to leave it for future works. Right now, some rigging (for

personnel models) is already made for future animations. It will surely be an interesting part to implement them in the future.

3.5. Integration

Code quality & documentations

As always, maintaining code quality is vital to the success of the development of any projects. Code quality can be maintained by setting rules in the .editorconfig file. The file contains rules about the code styles and naming conventions. If the rules are not conformed to, warnings will be generated in the code editor even before compilation. Yet, throughout the development of our project, we often need rules that are not covered in the .editorconfig file. Writing custom rules using Roslyn Analyzer will be a solution to this problem. Some examples of the custom rules needed are listed as followed:

- Forcing copy constructors in abstract class to inherit base class's copy constructors
- No direct use of new constructors for any assets
- Check whether all serializable properties (exclude those without setter) are included in copy constructor
- Force names of ClientRpcs with no ClientRpcParams to have suffix "AllClientRpc"

Documentation also comes into play when the project size grows larger. Clear documentation eliminates confusion of the use cases of methods and functions most of the time. Right now, we have some documentation for some methods. Yet, it is always a best practice to document every method in the project.

Version control

In this project, we use git, which is the most popular version control tool in the world, to increase the productivity for this collaborative project. Yet, the mechanism of tracking documents modification history for git does not work well with Unity Engine. One trouble is that it is normal to have scenes that include a large amount of GameObjects and attached scripts and it causes merge conflicts to happen frequently to the scenes when modification is done in different modules of the game. One workaround of this issue is to put all GameObject initialization logics into backend scripts but this is not a good solution as the difficulty of developing user interfaces would be significantly increased when there is no real-time preview in the development process. Another issue is that the automatic optimization process is done by Unity Engine which causes .asset files to be changing from time to time even if the scene is kept constant. The

change causes git to record unnecessary changes and increase the project size which is not desirable.

In order to solve the problem, further research on Perforce and Plastic SCM, which are two version control tools that support Unity integration, will be conducted.

4. Conclusion

At the outset of this project, our group had four main goals:

1. Creating a logistics management system in gameplay
2. Creating an attractive strategy game
3. Developing multiplayer features
4. Implementing the above objectives into a smooth game

Having reached the end of this project, we have achieved all of the goals set out at the beginning. Through the various gameplay systems implemented into the game, we have created a game with strategic depth especially regarding the logistic management of a player's army.

As avid game enjoyers, our game was created with a large focus on user experience and unique gameplay. Thus, our project features a blend of different features in other strategy games such as fog-of-war and simultaneous turns, creating a special experience for the player.

Although we achieved our goals, there is still much to be further implemented into our game to make it more complete and robust. Features such as but not limited to additional game modes, network security, and improved animations will make our game better and more enjoyable.

As a group, we have learned much about different aspects of game programming as well as problem solving. We look forward to continuing to improve our game.

5. Project Planning

5.1. GANTT Chart

Task	Jul	Aug	Sep	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May
Select game engine	■										
Design game mechanism	■	■	■	■	■						
Design Graphical User Interface			■	■	■	■	■	■	■		
Design game maps					■	■	■	■			
Design and develop algorithm for bots							■	■	■		
Design and create models					■	■	■	■	■		
Develop multiplayer mode					■	■	■	■	■		
Design background music						■	■	■			
Perform unit testing							■	■	■	■	
Write proposal		■	■								
Literature survey		■	■								
Write monthly report				■	■	■	■				
Write progress report							■	■			
Write final report									■	■	
Prepare for presentation										■	
Design video trailer										■	■

Table 9: Project Planning GANTT Chart

5.2. Division of work

Task	LEUNG, Ho Man Max	Marcus	Nathan	Hubert
Select game engine	**L**			
Design game mechanism	**L**			
Design Graphical User Interface		**L**		
Design game maps	**L**			
Design and develop algorithm for bots				**L**
Design and create models	**L**			
Develop multiplayer mode			**L**	
Design background music		**L**		
Perform unit testing	**L**			
Write proposal			**L**	
Literature survey			**L**	
Write monthly report			**L**	
Write progress report			**L**	
Write final report			**L**	
Prepare for presentation	**L**			
Design video trailer		**L**		

*L = Leader, everyone else assisting

Table 10: Division of Work Chart

6. Hardware and Software Requirements

6.1. Hardware

Hardware	Requirement
Personal Computer	Operating System: Windows 7 (64-bit) or higher
Graphics Card	Nvidia GTX 1050 or higher

Table 11: Hardware Requirements

6.2. Software

Software	Version	Usage
Unity	2020.3.17f1 or later	Game development
Visual Studio 2019	16.11.3 or later	Code development
Blender	3.0 or later	Models design
Adobe PhotoShop	CC 2017 or later	Texture maps creation
MuseScore 3	3.6.2 or later	BGM design
LMMS	1.2.1 or later	BGM design
Audacity	2.3.3 or later	BGM design
Figma	N/A	UI design

Table 12: Software Requirements

7. References

[1] C. Kelly, P. Johnson, and S. Schuler, “The new face of gaming,” *Accenture*, 27-Apr-2021. [Online]. Available: https://www.accenture.com/us-en/insights/software-platforms/gaming-the-next-super-platform. [Accessed: 12-Sep-2021].

[2] A. Stern and T. Murray, “Top 10 Esports games of 2020 by total Winnings – archive - the Esports Observer,” *The Esports Observer*, 03-Jan-2021. [Online]. Available: https://archive.esportsobserver.com/top10-games-2020-total-winnings/. [Accessed: 12-Sep-2021].

[3] Japaneseretrogames, 2011. *Thumnail of Advanced Daisenryaku Video.* [image] Available at: https://www.youtube.com/watch?v=CBO-ZWgBZws [Accessed: 12-Sep-2021].

[4] G2A, *Screenshot of Hearts of Iron IV Gameplay* [image] Available at: https://www.g2a.com/it/hearts-of-iron-iv-cadet-edition-steam-key-europe-i10000017487006 [Accessed: 12-Sep-2021].

[5] alternatehistory.com, 2015, *Standard map of the game of Diplomacy.* [image] Available at: https://www.alternatehistory.com/forum/threads/return-of-horrible-educational-maps.375167/page-20 [Accessed: 12-Sep-2021].

[6] Blizzard Entertainment, *Versus Mode* [image] Available at: https://starcraft2.com/en-us/ [Accessed: 12-Sep-2021].

[7] “Unity UI: Unity user interface: Unity Ui: 1.0.0,” Unity UI | 1.0.0. [Online] Available at: https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/index.html [Accessed: 20-Apr-2022].

[8] “Unity Multiplayer Networking: Netcode for GameObjects,” *Unity*. [Online]. Available: https://docs-multiplayer.unity3d.com/. [Accessed: 17-Sep-2021].

[9] Making maps with noise functions. [Online]. Available: https://www.redblobgames.com/maps/terrain-from-noise/#elevation. [Accessed: 20-Apr-2022].

[10] I. Quilez, “warp,” Inigo Quilez :: computer graphics, mathematics, shaders, fractals, demoscene and more. [Online]. Available: https://iquilezles.org/articles/warp/. [Accessed: 20-Apr-2022].

[11] U. Technologies, “Scenemanager,” Unity. [Online]. Available: https://docs.unity3d.com/ScriptReference/SceneManagement.SceneManager.html. [Accessed: 20-Apr-2022].

[12] LeanTween. [Online]. Available: http://dentedpixel.com/LeanTweenDocumentation/classes/LeanTween.html. [Accessed: 20-Apr-2022].

[13] “Introduction,” Introduction - Blender Manual, 20-Apr-2022. [Online]. Available: https://docs.blender.org/manual/en/latest/modeling/modifiers/introduction.html. [Accessed: 20-Apr-2022].

[14] gaz_mm-v_093_of_130.jpg (2240×1680). [Online]. Available: http://data3.primeportal.net/trucks/yuri_pasholok/gaz_mm-v/images/gaz_mm-v_093_of_130.jpg. [Accessed: 20-Apr-2022].

[15] “Introduction,” Introduction - Blender Manual, 20-Apr-2022. [Online]. Available: https://docs.blender.org/manual/en/latest/modeling/geometry_nodes/introduction.html. [Accessed: 20-Apr-2022].

[16] “Instances,” Instances - Blender Manual, 20-Apr-2022. [Online]. Available: https://docs.blender.org/manual/en/latest/modeling/geometry_nodes/instances.html. [Accessed: 20-Apr-2022].

[17] “Introduction,” Introduction - Blender Manual, 20-Apr-2022. [Online]. Available: https://docs.blender.org/manual/en/latest/render/shader_nodes/introduction.html#:~:text=The%20output%20of%20all%20surface,Emission%20shader. [Accessed: 20-Apr-2022].

[18] U. Technologies, “Mathf.PerlinNoise,” Unity - Scripting API: Mathf.PerlinNoise. [Online]. Available: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html. [Accessed: 20-Apr-2022].

8. Appendix A: Meeting Minutes

8.1. Minutes of the 1st Meeting

Meeting 1 - Sept 4th, 2021

Attending:

- Nathan
- Marcus
- Max
- Hubert

- Today mainly do Division of Labor (3.2)
- Asterisks in 3.2 mean optional features
- Props:
    - Free assets? Make our own? Buy assets?
    - For now use default/free assets
- BGM:
    - Marcus will try to do
- Multiplayer server & network
    - Agreement that it would be very preferable that multiplayer is implemented
- Map
    - Perlin noise algorithm
- Achievements
    - Yes
- Chat function
    - Save till last
- Record & Replay
    - Yes

Development Schedule:

- Testing/Rebalancing/Bug fixes: Early March 2021

Use Max's Server

To do:
GANTT chart, table for 3.2, 4.1, 4.2
Flowchart of game

Tentative date of next meeting: Sept 18th, 2021 9pm

8.2. Minutes of the 2nd Meeting

Meeting 2 - Sept 8th, 2021 with Prof Arya

Attending:

- Max
- Marcus
- Nathan
- Prof Arya

Main Questions regarding:

- Objectives
    - What sort of game we're developing
    - How will it be different (**unique features**)
    - Motivations for developing the game
    - Inspiration from different games, in what way we're inspired and what features we want to include
- Lit Survey
    - Comparing similar products
    - Comparing weaknesses and strengths
    - Go into more detail
    - The more we explore the better
    - its flexible
    - Can read both papers and look at other products
- Methodology

- Make sure the game is interesting and fun to play
- But also needs to be a little technical, uses graphics, ai, collision detection etc. (something more advanced, requires original thought)

Fog of war

Maybe group units to move them

Researching during a battle may not seem historically accurate

Historical mode - upgrade during battle

Multiplayer mode - certain number of points to put into upgrades, cannot change during match

8.3. Minutes of the 3rd Meeting

Meeting 3 - Sept 13rd, 2021

Attending:

- Max
- Hubert
- Marcus
- Nathan

Things discussed:

- Game Mechanics
    - How to win: Destroy all enemy bases (starting capital + cities/villages)
    - Logistics
        - Has seperate layer (view)
        - Supplied from outposts
        - Player chooses roads (draws them)
        - Going over different terrain makes it slower (3,2,1)
        - Enemy routes are hidden
        - Killing convoys lets you restock

        - Logistics units running around
        - Can build infrastructure for transport units
            - Roads (trucks)
            - Rails (trains)
        - Maybe have map generate some road parts
        - Maybe logistics units can go over rough terrain, some supply lost, takes longer
    - Unit Resource Consumption (supply)
        - Resource consumption based on tiles traveled, but even not moving takes supply
        - If a unit is on a transport, it does not consume supply
    - Resources
        - Occurs when a player owns Capitals/Cities/Villages
        - Resources added each turn
        - Can construct resource buildings to increase production (marginal return is less)
        - Types of resources
            - Ammo: Produced from munitions factory
                - Can be later split into cartridges/shells

            - (discuss later)
        - Weather system
            - Seasons
            - Terrain affected (eg. rivers freezing over, muddy swamps making it harder to travel)
- Overall flow of the game
    - Start with a capital
    - (discuss later)
- Combat
    - Each unit can either move, fire, or hide(ambush mode)
    - Ambush mode
        - Takes a turn to set up
        - When dealing damage, you deal damage first
        - Moving out of ambush your range is halved
    - If a unit has range 2, and is shooting at a moving target:
        - If the moving target is moving to a tile still in range, damage will be lessened (25%?)
        - If the moving target is moving to a tile not in range, damage will be further lessened (50%?)
- Objective of Project
- Methodology (Part 2)
    - Design
        - Describe structure of game
        - Game flowchart
    - Implementation
        - Classifying what we're going to do into small parts
        - Explain what we're going to do for each part and what tools
    - Testing
        - Content:
            - Mechanisms to test
        - Self testing
        - Automated tests?
        - Send to friends
    - Evaluation
- Hardware and Software Requirements

Things to do by Wednesday:

Max: Lit Survey (1.3) for the first 3, Implementation (2.2), Hardware and Software (4.1, 4.2) Requirements
Nathan: Finish Overview (1.1), Objectives (1.2), add Starcraft inspiration to 1.3, Design (2.1)
Marcus: Testing (2.3)
Hubert: Evaluation (2.4)

8.4. Minutes of the 4th Meeting

Meeting 4 - Sept 18th, 2021

Attending:

- Max
- Nathan
- Hubert
- Marcus

Class diagram to be sent by Max tomorrow
Game class includes all props

Talk further about game mechanics and how to implement them

Hearts of Iron AI (simple approach):

- How to determine which unit to train?
- Probabilistic actions

Rule-based AI

Possibly have units with abilities?

Discussion about Timeframe of Project
Keep management simple first: the battle comes first

Todo:

- Search for assets from Unity assets store
- Next Wednesday Max gives us basic files

8.5. Minutes of the 5th Meeting

Meeting 5 - Oct 17th, 2021

Attending:

- Marcus
- Nathan
- Hubert
- Max

Discussion of implementation:

- Big game class that saves gamestate of each turn
- Another interface for each player that indicates what info is shown to that exact player
- AI can build a bot on top of that interface

Todo:

- Meet up soon to finalize
- Marcus works on sample buttons in PS
- Nathan works on multiplayer
- Hubert works on AI
- Max working on initial implementation

8.6. Minutes of the 6th Meeting

Meeting 6 - Feb 20th, 2022

Attending:

- Marcus
- Max
- Nathan
- Hubert

Catching up on what we've done

Discussion on gameplay mechanics:

- As long as a unit moves into the range of an outpost, it gets resupplied, doesn't have to stop in range (enables smoother gameplay)

Multiplayer:

- Anybody can be host
- Calculations made server-side, send info back to client
- Server sends:
    - Filter map, events to different clients
    - Map updates
    - Events
- Client needs to:
    - Render map updates
    - Render events

Next Meeting: Next weekend

8.7. Minutes of the 7th Meeting

Meeting 7 - Feb 27th, 2022

Attending:

- Marcus
- Max
- Nathan
- Hubert

Discussed:

- FooServerRPC, C# Primitives
- Start by Ping pong between server & client
- Classes no longer inherit Monobehaviour
    - Helps serialization
- Keep track of on-screen events, do logic behind
- Game design:
    - Drop planes, no time for textures
- Coding shaders

Next Meeting: March 1st, 2022 9:30pm

8.8. Minutes of the 8th Meeting

Meeting 8 - Mar 12, 2022

Attending:

- Marcus
- Max
- Nathan
- Hubert

Discussed:

- RPC calls
- More on classes not inheriting from MonoBehaviour
- Map class
- Start some testing now
- Object models finished soon
- Game launch -> Menu -> Game Start -> Instantiating Map, etc. -> Battle Scene
- Merging UI branch on Github
- More info on AI

8.9. Minutes of the 9th Meeting

Meeting 9 - Mar 26th, 2022

Attending:

- Marcus
- Max
- Nathan
- Hubert

Discussed Topics:

- Help Hubert with his AI stuff, Max meet him individually on Monday
- Show same scene in Unity? How to do fog of war?
- Talked about splitting functions into client and server rpcs
- Max meet with Nathan tomorrow again to discuss unit tests
- Max try out some multiplayer rpc simple stuff first

8.10. Minutes of the 10th Meeting

Meeting 10 - April 2nd, 2022

Attending:

- Marcus
- Max
- Nathan
- Hubert

Weekly update:

- Game Logic Testing
- AI Train method
- UI updates

8.11. Minutes of the 11th Meeting

Meeting 11 - April 9th, 2022

Attending:

- Marcus
- Max
- Nathan
- Hubert
- Noor Liza

Discussed Topics for Final Report:

- Skip straight to methodology
- Appendix: Reference point for the reader to quickly understand the units
- Class diagram is for different purpose
- Keep the developer audience in mind

- How the algorithms are modified, A*, Perlin Noise, Poisson Disk
- What would you do differently?

Main Things to write about:

- Big Pillars:
    - Map Generation
        - Algorithm, but before we get to that:
        - Talk about the relevance of algorithm:
            - Ie. explaining tiles
        - Poisson Disk
    - Units/Assets
        - Movement
        - Pathfinding Algorithm (related to river generation)
        - Command implementation
    - Multiplayer
    - Graphics
        - Tiles
- Integration of these pillars

Normal Meeting:

- Multiplayer
    - How to fetch configs and maps from host
    - 3 important things:

        - How to send command to server
        - In the UI: After play button -> Connect / Host
    - More Unit Tests (Prop.cs)
- Camera Controllers
    - FPS drops? Should be ok
- AI Training method

8.12. Minutes of the 12th Meeting

Meeting 12 - April 16th, 2022

Attending:

- Marcus
- Max
- Nathan
- Hubert

Final Report:

- Do your parts
- Mainly write about:
    - What you have learned
    - How to implement it
- Multiplayer - Nathan
- Gameplay & UI - Hubert and Marcus
- Add diagrams! - use draw.io, caption the diagram
- Add an intro, add diagrams
- Ask Max if you need any help!

- Update Table of Contents -> Nathan
- **Finish on or before 19th, then we can proofread**
- Fix up Meeting Minutes - Nathan
- Glossary - Nathan

9. Appendix B: Glossary

9.1. Basic Terms

- **Asset:** Anything which is interactable with the player. Consists of props and customizables.
- **Attribute:** An object that consists of a modifier and an underlying value, representing an aspect of an asset. For the full list of attributes, see section 9.2.
- **Battle:** Where players confront each other to compete for being the only winner. Consist of multiple rounds.
- **Buildings:** Grant the player various advantages if built. Can be fortified, demolished and sabotaged. Divided into 5 categories:
  - **Production:** boost production in general
  - **Defensive:** slow down enemies' advancement or give them surprise attacks
  - **Infrastructure:** general purpose, from setting up temporal bases to granting access to inaccessible terrains
  - **Transmission:** boost friendly units' spotting and signaling abilities, or hinder that of enemies'
  - **Training Grounds:** Training ground for units, can use for rearming as well
- **Cities:** Special type of terrain and players' objectives. Destructible. Capturable. Produce resources each round. Come with a barracks and an arsenal for each. Three tiers (from high to low):
  - **Metropolis:** Main objective of a player is to defend their sole metropolis and destroy the others'. If destroyed, the player is defeated. Game ends when there is only one undefeated player left, which is the winner.
  - **City:** Resource producing city of medium size. Larger and more valuable than suburbs.
  - **Suburb:** Lowest tier city, provides small amount of resources.
- **Command:** An action which the player can do, or can assign to his unit. Each unit can only be assigned with one single command each turn. For the full list of commands, see section 9.3.
- **Customizables:** Provides customizations to units which would directly affect their attributes. There are 3 categories:
  - **Firearms:** Equipped by personnel, can have primary and secondary
  - **Modules:** Parts of any units other than personnel. Each of them has their own "health", namely Integrity. If the integrity reaches zero, the module will be out of order and requires repair to work again. Some modules are listed below:
    - Main Gun (aka the cannon)
    - Heave Machine Gun

        - Engine
        - Suspension
        - Radio
        - Periscope
        - Fuel Tank
    - **Shells:** Different armaments have different compatible sets of shells. Some properties of shells like penetration vary among shells, serving for different purposes.

➔ **Fog of War:** Feature that hides information from players unless the area has been explored /

➔ **Game:** Refers to this project

➔ **Map:** Where the battle takes place, contains hexagonal tiles with different terrains.

➔ **Modifier:** A numerical value which can alter the underlying value of an attribute. The effect of altering can be both permanent (last for the whole battle) or temporal (one to few rounds).

➔ **Phase:** The order of execution of commands of the same category. Categories of commands are listed as subheadings of section 9.3.

➔ **Prop:** Basically anything on the battle scene. Including tiles, units and buildings.

➔ **Round:** Where players make decisions by assigning commands to their units.

➔ **Spot:** To detect a unit belonging to another player, or to have your unit detected by another player

➔ **Signal:** An invisible chain of command, if a unit cannot be signaled, it becomes disconnected

➔ **Stream:** Terrain that mimics real world streams.

➔ **Terrain:** An abstraction of real-world terrains like grassland, forest, river, etc. represented in 3D models. Each type of terrain has types of modifiers of different values to affect values of attributes of units.

➔ **Tile:** Represented by standard hexagonal prisms of apothem 18.9m in Blender measurement units.

➔ **Unit:** Think of it as a chess piece, but can be trained each round by consuming resources, and there are 3 major categories of units:

- **Personnel:** Ground units. Sole class of units that can capture cities.
- **Artilleries:** Ground units. Provide supportive fire from long range. Weak in confrontations or direct fights. Some are designated for countering vehicles.
- **Vehicles:** Ground units. Used for covering advancement of personnel in breakthroughs in general. But some are designated for logistics.

9.2. Attributes

9.2.1. Basics

- ➔ **Cost:** Resources consumed when performing the corresponding command.
  - ◆ **Base:** Base cost of training a unit, equipping a unit with customizable or constructing a building
  - ◆ **Repair:** Cost of repairing a module
  - ◆ **Fortification:** Cost of fortifying a building
- ➔ **Resources:** Vital to all operations in the game, like training units, movement of units, constructing buildings. There are 5 types of resources:
  - ◆ **Money:** Most used in constructions.
  - ◆ **Steel:** Most used in producing units except personnel.
  - ◆ **Supplies:** Rations to support the lives of soldiers. Most used in producing personnel and movement of units.
  - ◆ **Cartridges:** Used by firing most of the firearms.
  - ◆ **Shells:** Used by firing some kinds of firearms and all kinds of guns
  - ◆ **Fuel:** Used by movement of vehicles and firing some kinds of firearms

9.2.2. Maneuverability (determines how maneuverable a unit is)

- ➔ **Mobility:** Determine how mobile a unit is, usually has negative correlation with Size
- ➔ **Size:** Determine how bulky a unit is
- ➔ **Speed:** Maximum number of tiles a unit can move each turn
- ➔ **Weight:** Determine how heavy a unit is, has positive correlation with Size

9.2.3. Defense (determines the ease of being eliminated by hostiles)

- ➔ **Strength:** The maximum amount of damage the unit can take
- ➔ **Resistance:** The effective armor value of vehicles to withstand penetration of shells firing from hostiles
- ➔ **Evasion:** The chance of completely dodging the incoming attack, no damage will be taken if the unit successfully evaded the attack.
- ➔ **Hardness:** How “hard” the unit is, the more man the unit is made up of, the softer. The more steel the unit is made up of, the harder.
- ➔ **Suppression:** Consists of threshold and resilience. Not to be confused with the command suppression, suppression is also a unit defensive attribute.

- **Threshold:** The minimum level of suppression the unit must have before being suppressed.
- **Resilience:** The drop in level of suppression each round.

9.2.4. Offense (determines the ease of eliminating hostiles, weapons specific)

- **Handling:** How easy the weapon can be used
- **Damage:** How much damage the weapon can inflict on hostiles
- **Accuracy:** How accurate the weapon is
- **Suppression:** Determines how much suppression a specific weapon can inflict. The suppression value attribute of a weapon is not to be confused with the unit defensive attribute or command.
- **Range (min/max):** The effective firing range of the weapon

9.2.5. Scouting (determines the ease of being spotted and spotting hostiles)

- **Reconnaissance:** The maximum range that hostiles are detectable by the unit
- **Concealment:** How well it can remain hidden from the hostiles
- **Detection:** How well it can spot the hostiles
- **Communication:** The maximum range of transmitting signals to friendlies

9.3. Commands

9.3.1. Movement-related

- **Hold:** Do nothing
- **Move:** Move to a specific destination on the map

9.3.2. Attack-related

- **Ambush:** Assign the unit into ambush mode, fire preemptively before the start of next round if any hostile units is within range. Cancel ambush status after the preemptive strike. Cannot move when ambushing but gains considerable concealment.
- **Fire:** Fire in normal mode, can choose which firearm/armament to use
- **Suppress:** Suppress the enemy, accuracy is greatly reduced
- **Sabotage:** Command for firing at buildings

9.3.3. Logistics-related

- **Repair:** Repair modules of adjacent units
- **Resupply:** Retrieve resources from nearest outpost

9.3.4. Construction-related (Available in city menu or for Personnels)

- **Construct:** Construct a new building on an available tile.

- ➔ **Demolish:** Completely remove a building. Usually used for freeing up available tiles for other more important buildings.
- ➔ **Fortify:** Upgrade an existing building for higher durability

9.3.5. Training-related (Available only in training ground menus)

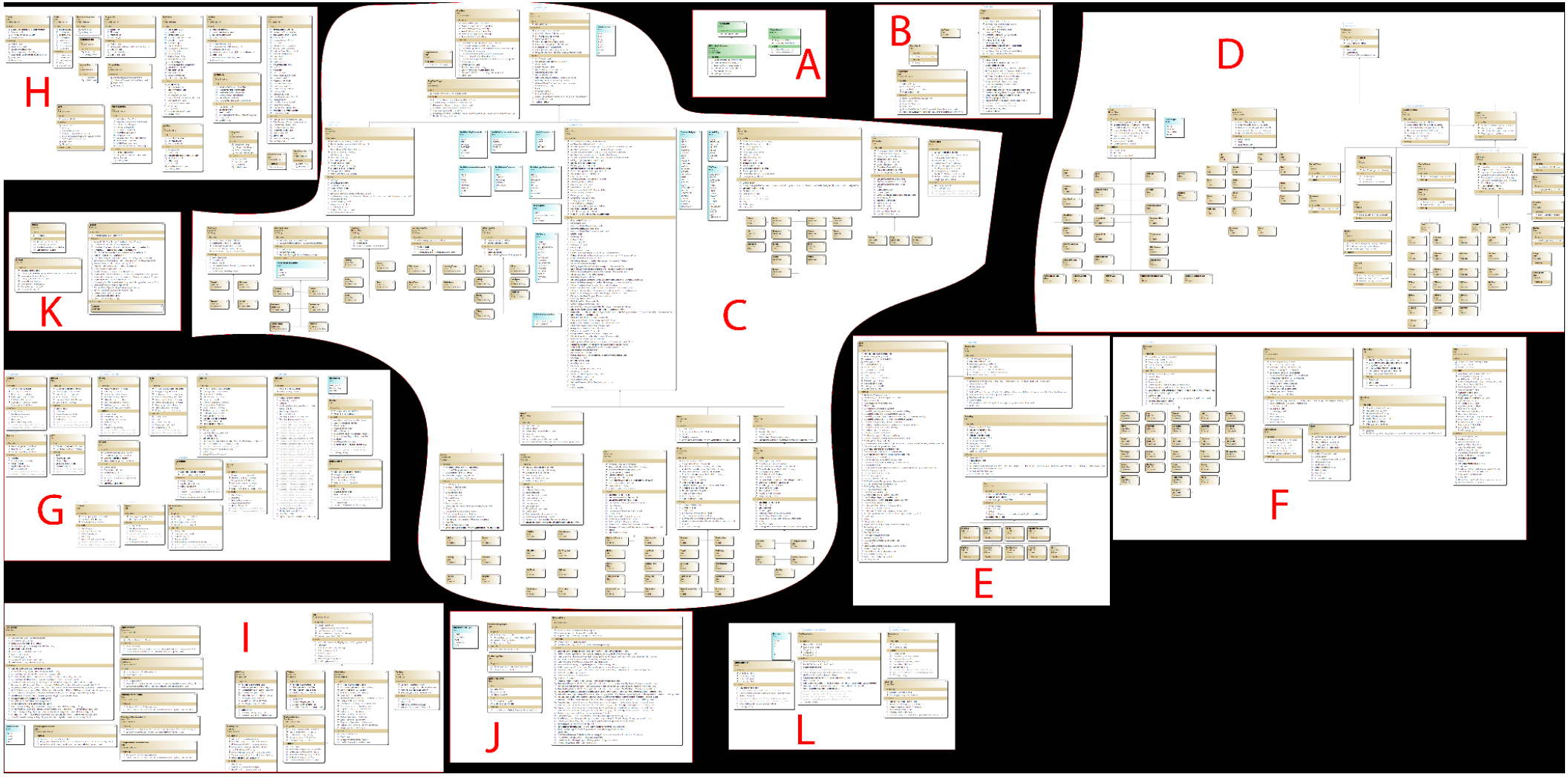
- ➔ **Deploy:** Deploy a trained unit to any adjacent tiles of the training ground
- ➔ **Train:** Train a unit, consuming corresponding cost of training and then add the unit to the training queue. Cannot train more units if the training queue is full.

9.3.6. Miscellaneous

- ➔ **Assemble:** Assemble gun pieces so that the unit can fire. For specific types of artilleries only. Gains speed reduction when the gun is assembled.
- ➔ **Capture:** Reduces the morale of a city, if the morale drops to 0, the owner of the city will be the owner of that unit.
- ➔ **Disassemble:** Dissemble gun into pieces so that the unit can move faster. For specific types of artilleries only. Cannot fire if the gun is disassembled.

10. Appendix C: Class Diagrams

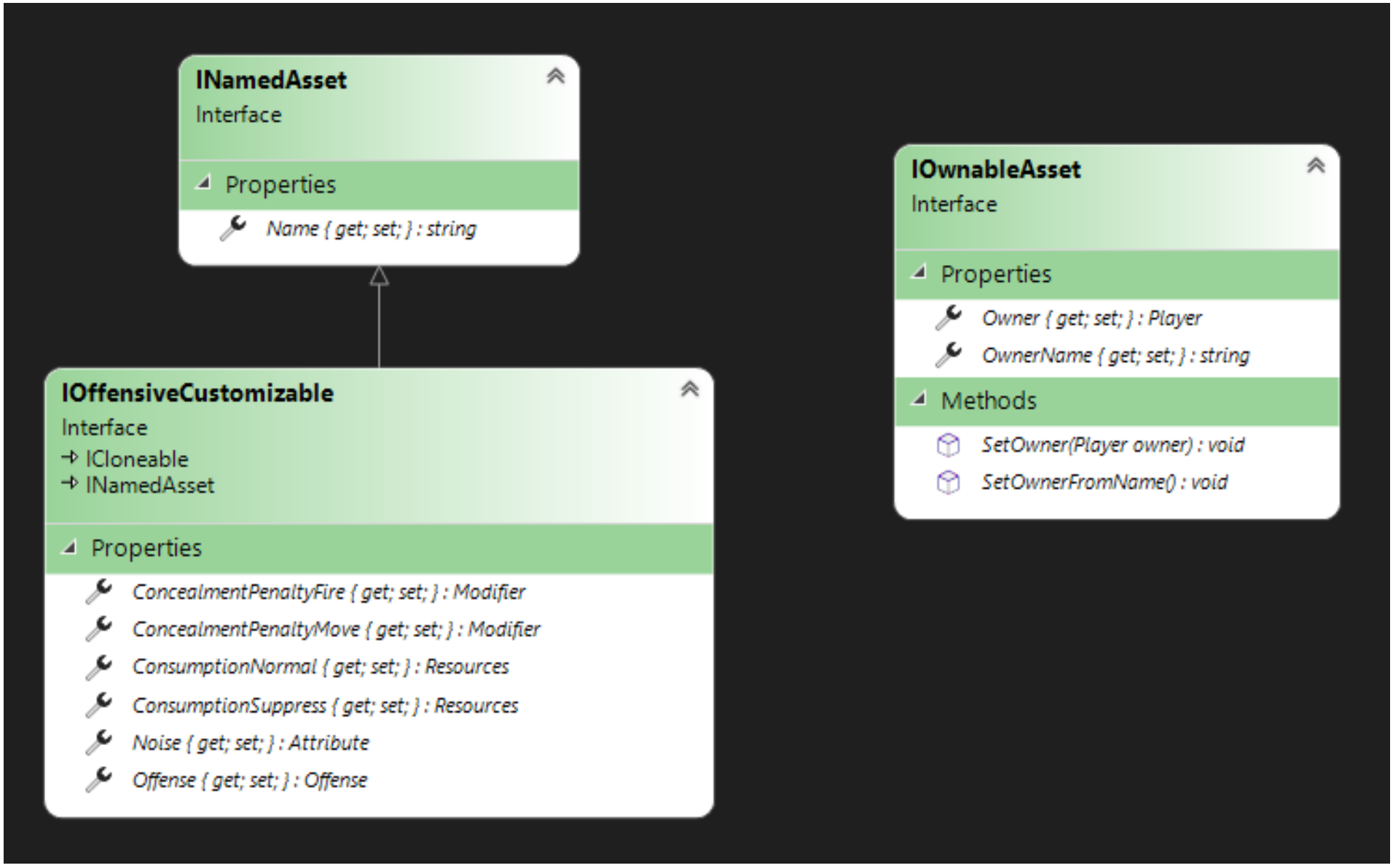
10.1. Overview



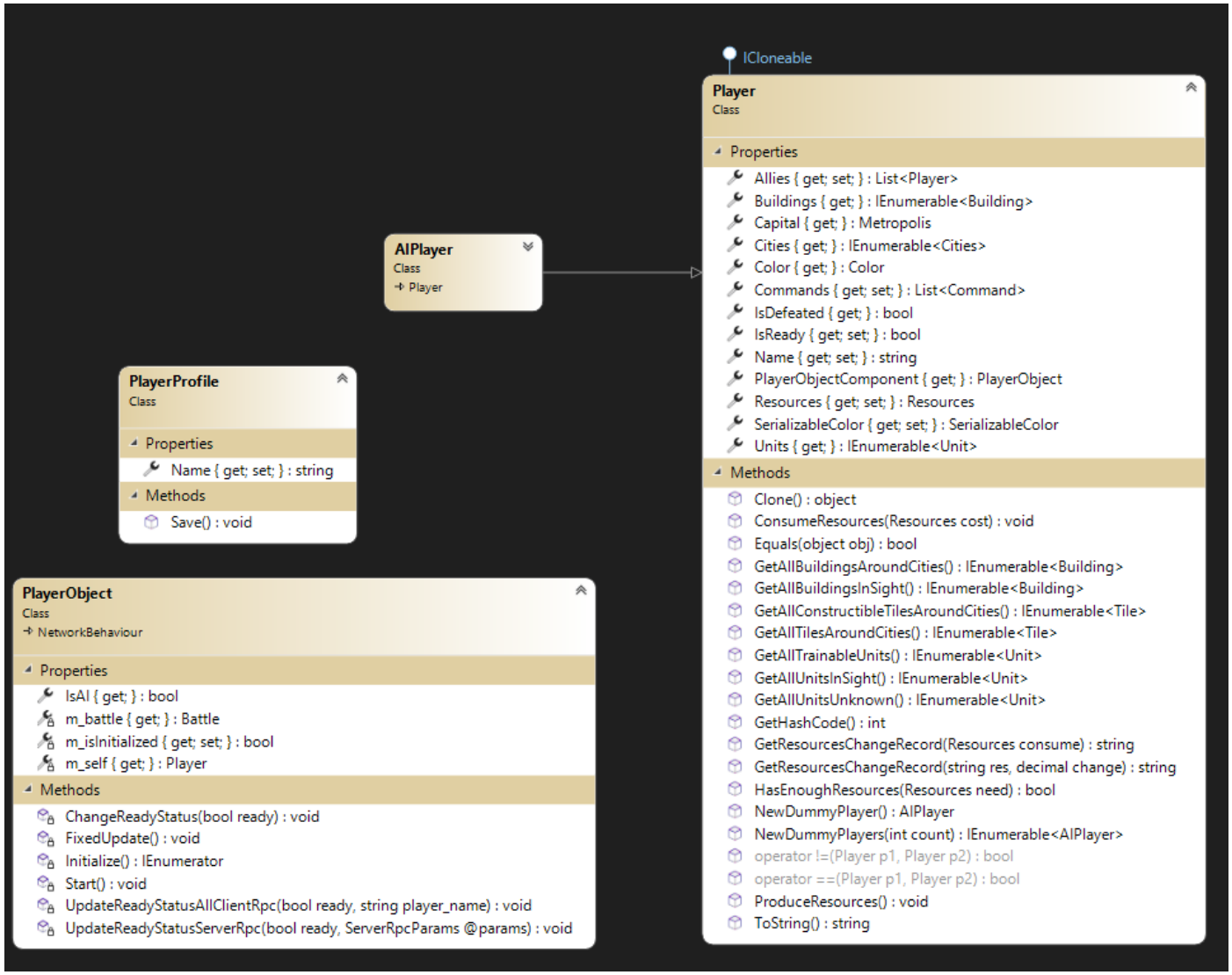
Legend:

A: Base interfaces; B: Player classes; C: Prop classes; D: Customizable classes; E: Map classes; F: Game flow classes; G: Attribute and modifier classes; H: UI classes; I: Data classes; J: Networking classes; K: Utility classes; L: Custom types

10.2. Base Interfaces



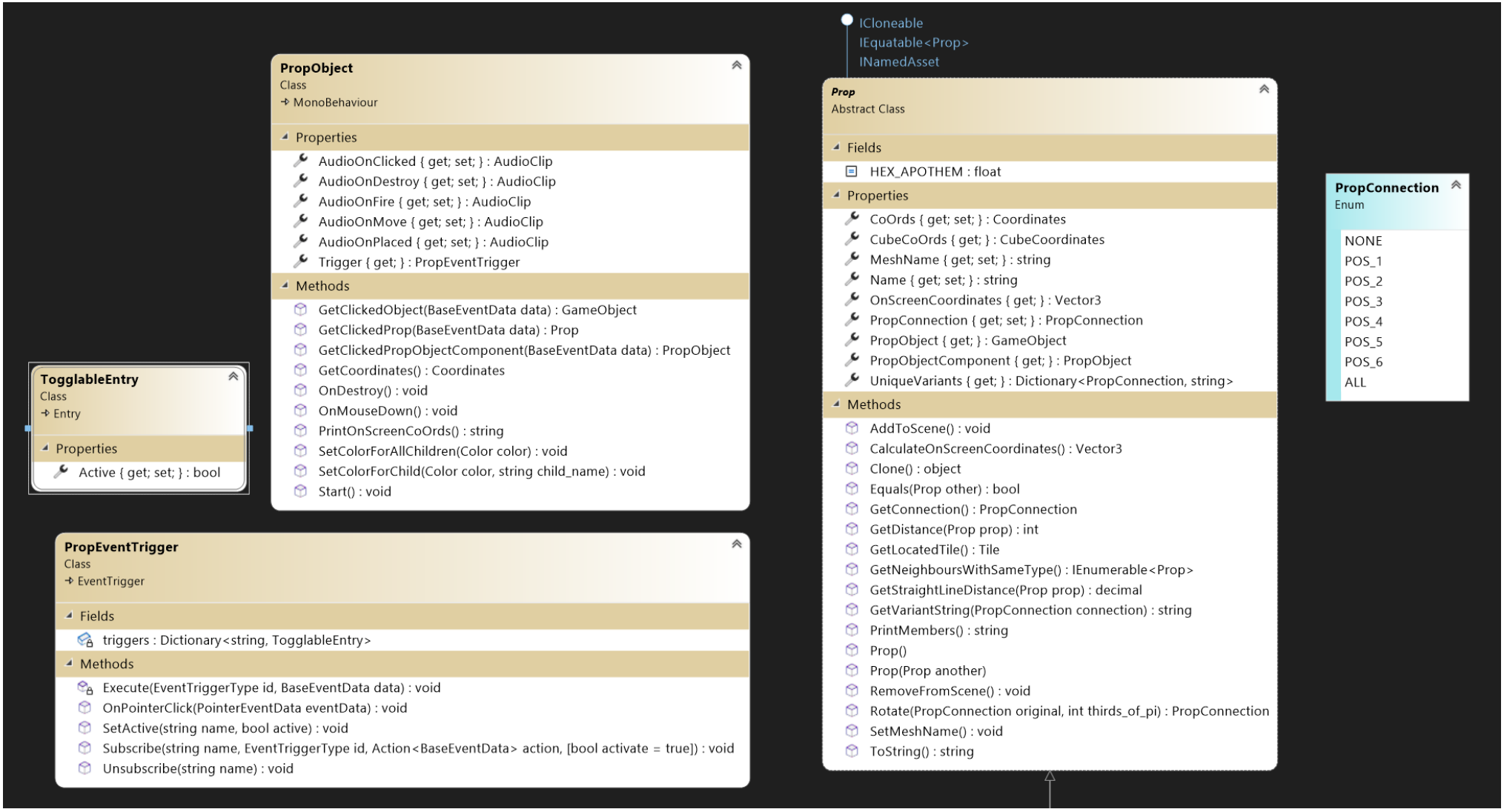
10.3. Player Classes



All classes related to a player. The PlayerObject class is used for communicating with the server (if connected as a client).

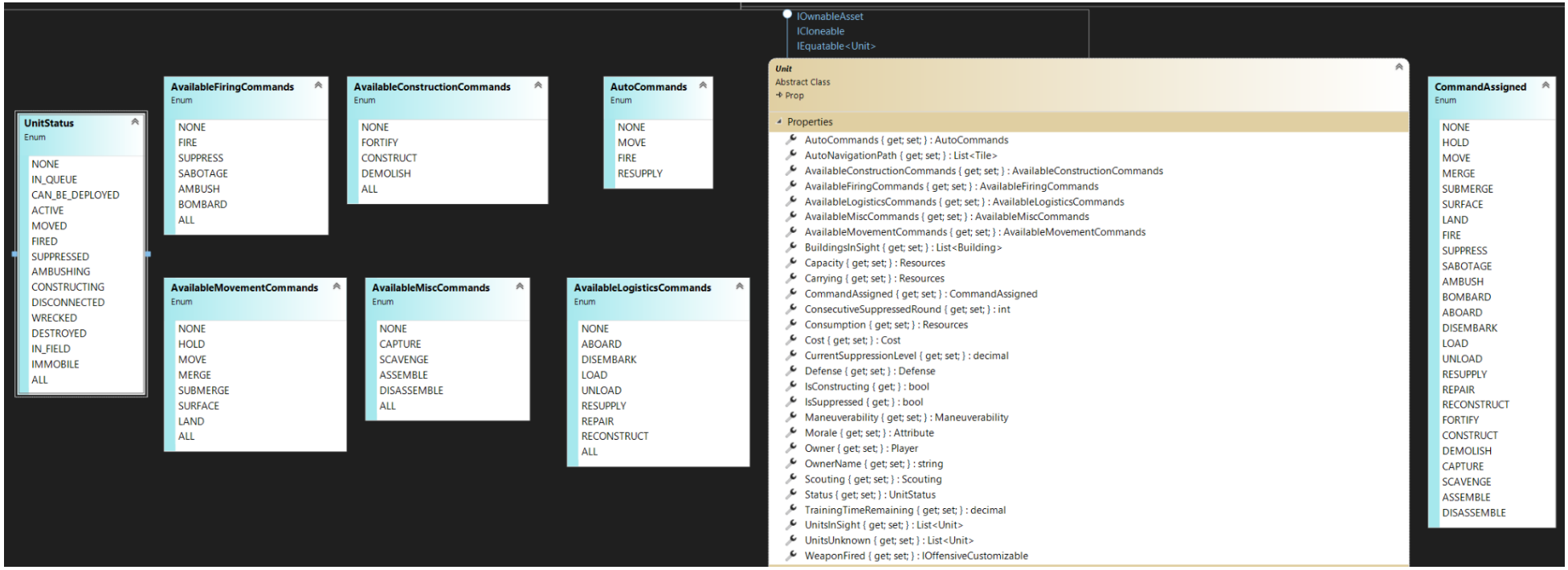
10.4. Asset Classes

10.4.1. Prop Classes

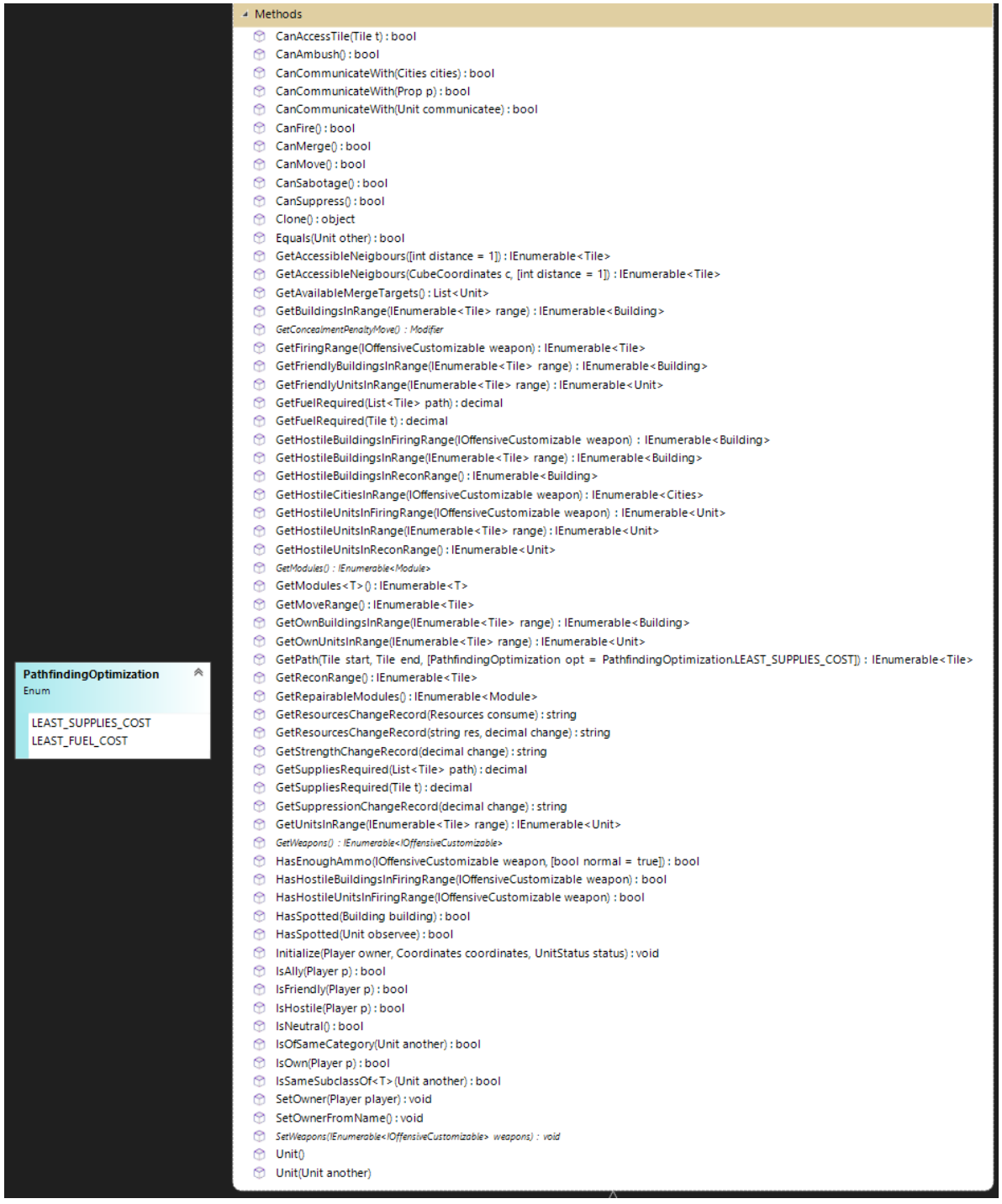


All classes related to a prop. PropObject class is the representation of the game object instantiated on the scene. Its name is set to be the MeshName of the Prop (C# object, base class of all props) so as to link them together.

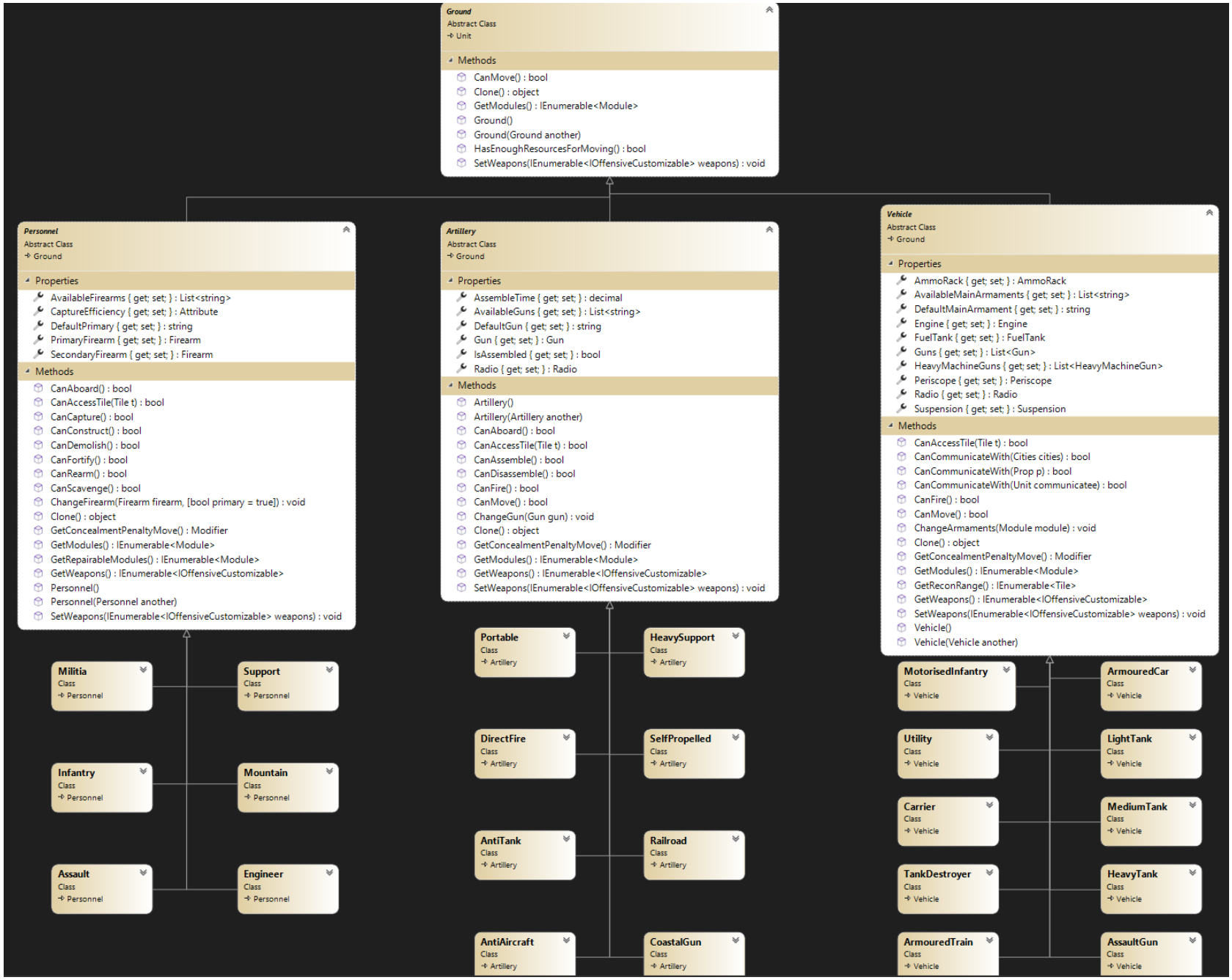
10.4.1.1. Unit Classes



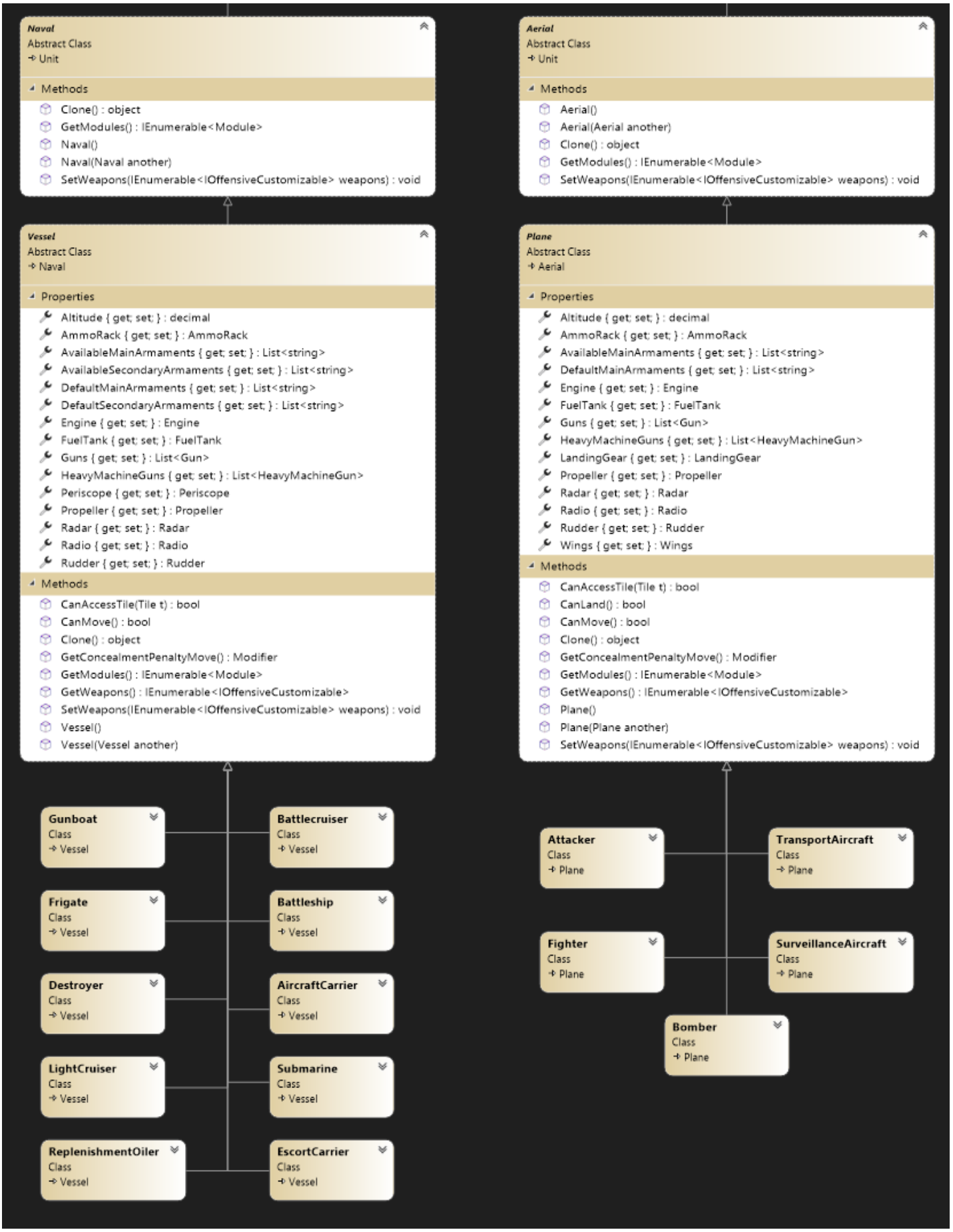
Properties and enums of Unit class (base class of all units)



Methods of Unit class

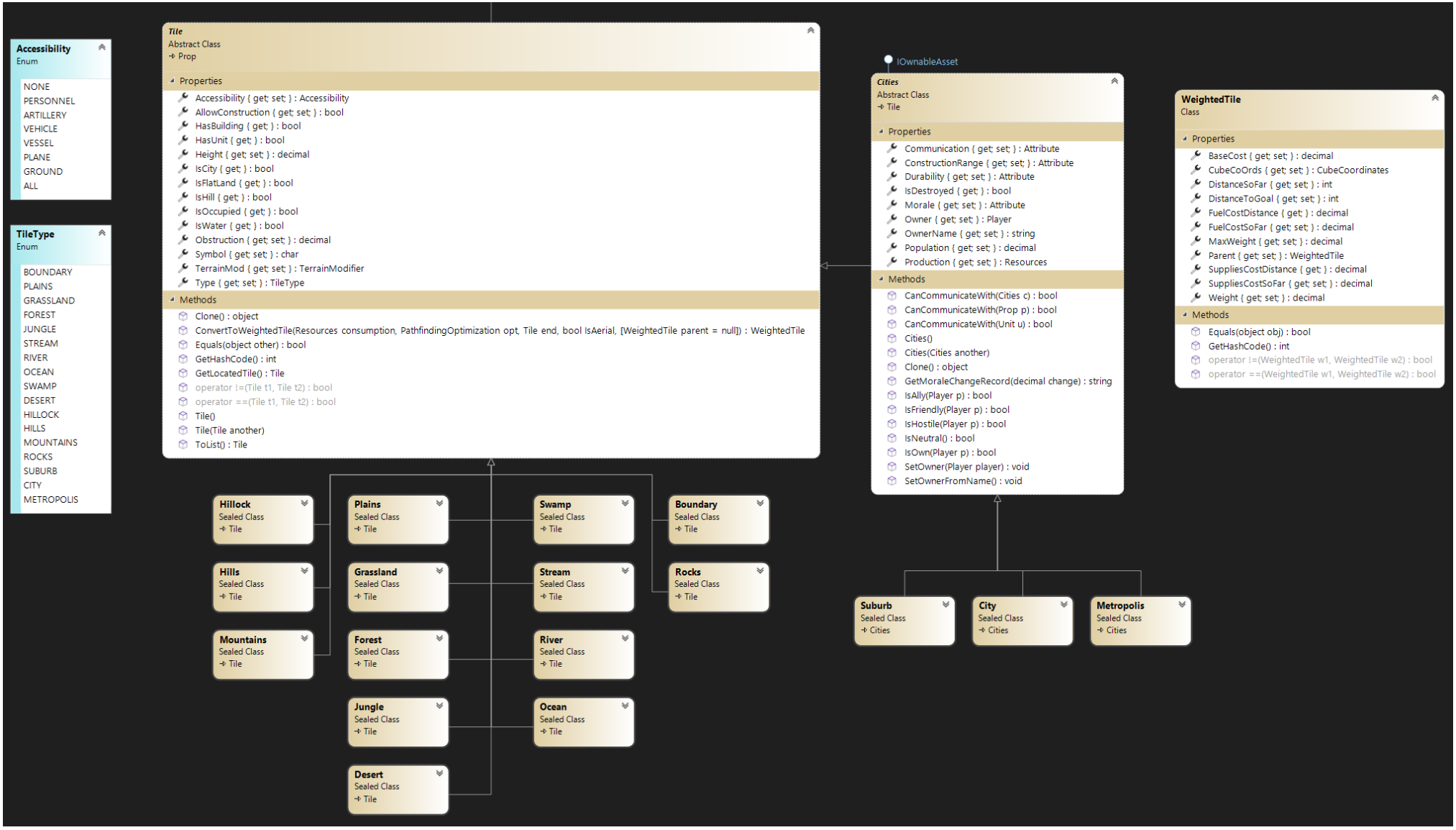


Subclasses of Ground Unit class (inherit from the Unit class)



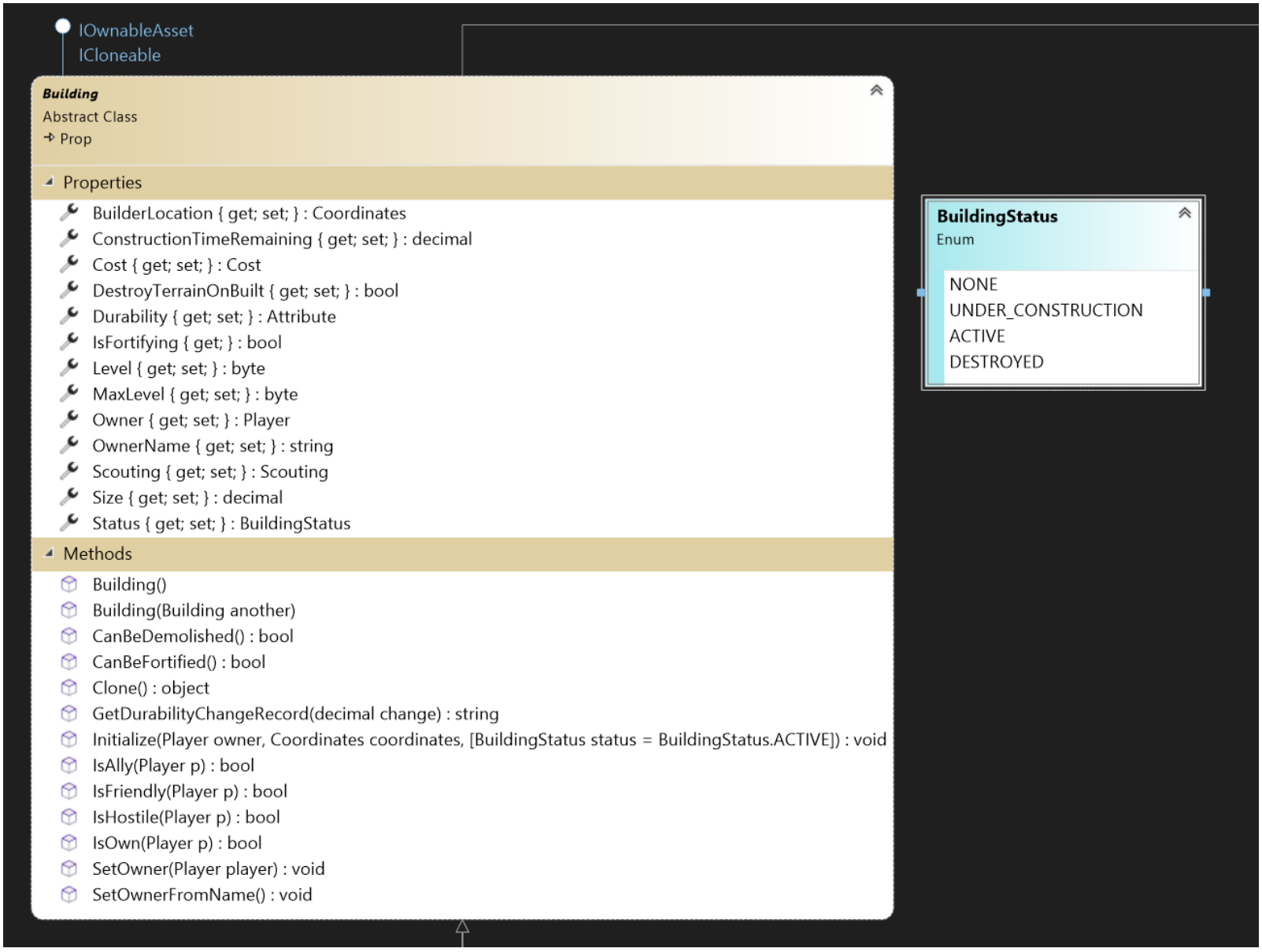
The Naval and Aerial classes and all of their subclasses (implemented for future use).

10.4.1.2. Tile Classes

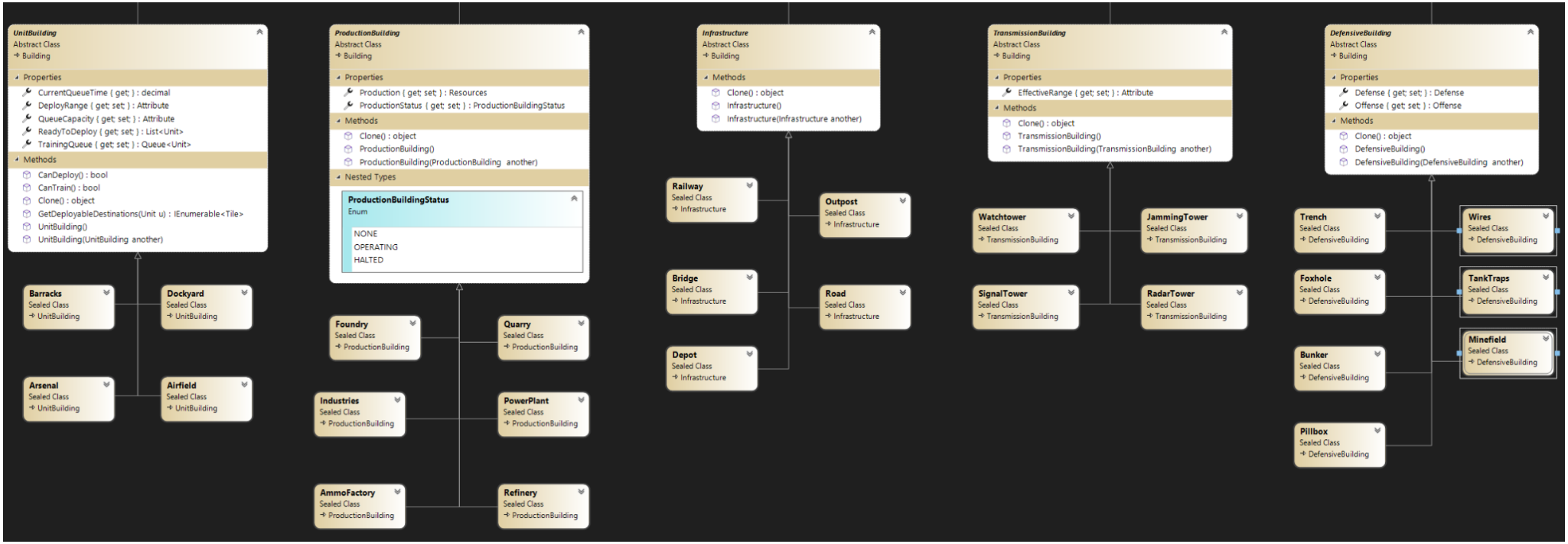


All classes related to a tile. The WeightedTile class is used for A* pathfinding with custom weighting.

10.4.1.3. Building Classes

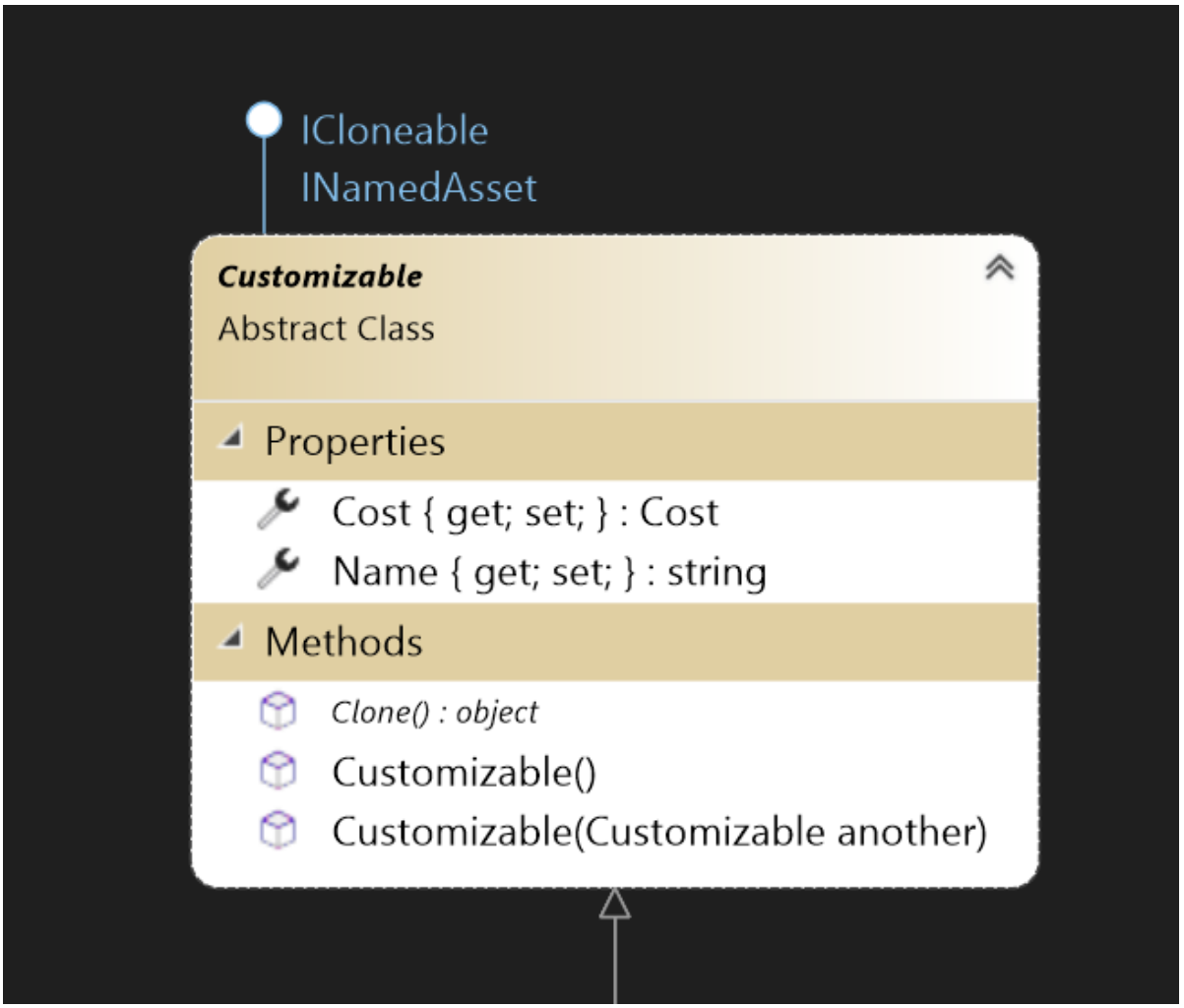


The Building class (base class of all buildings)



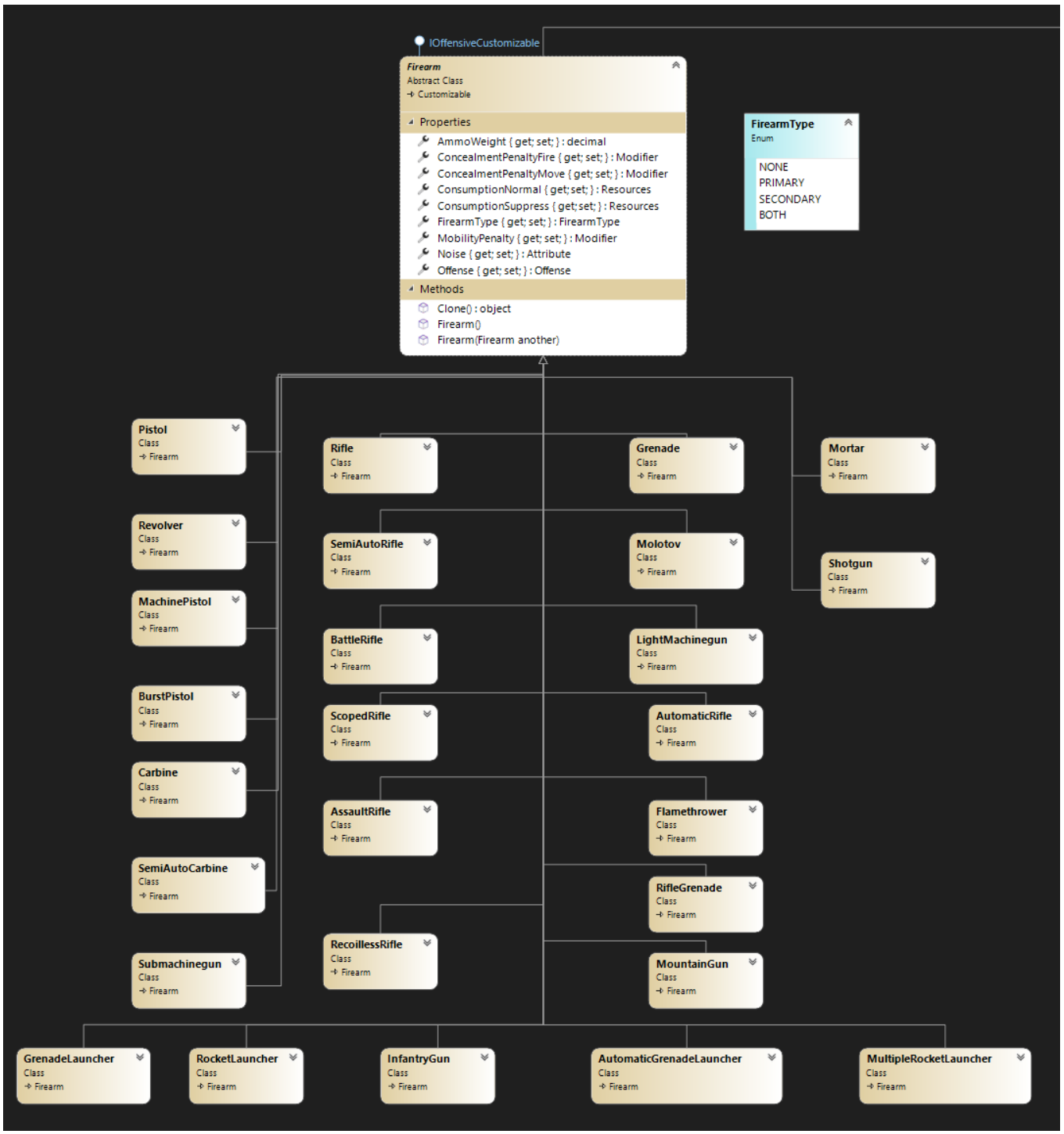
All subclasses of the Building class

10.4.2. Customizable Classes



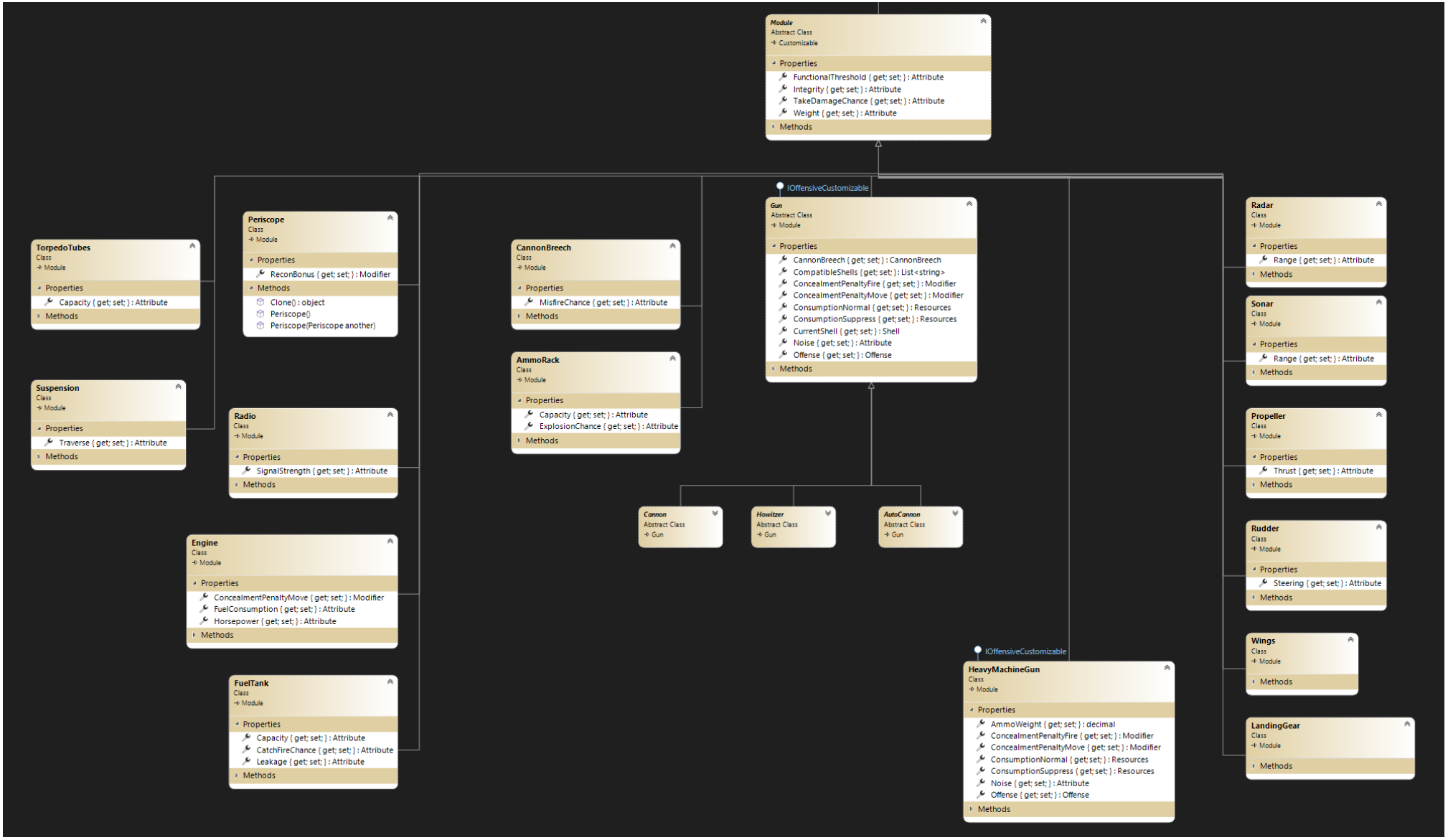
The Customizable class (base class of all customizables)

10.4.2.1. Firearm Classes



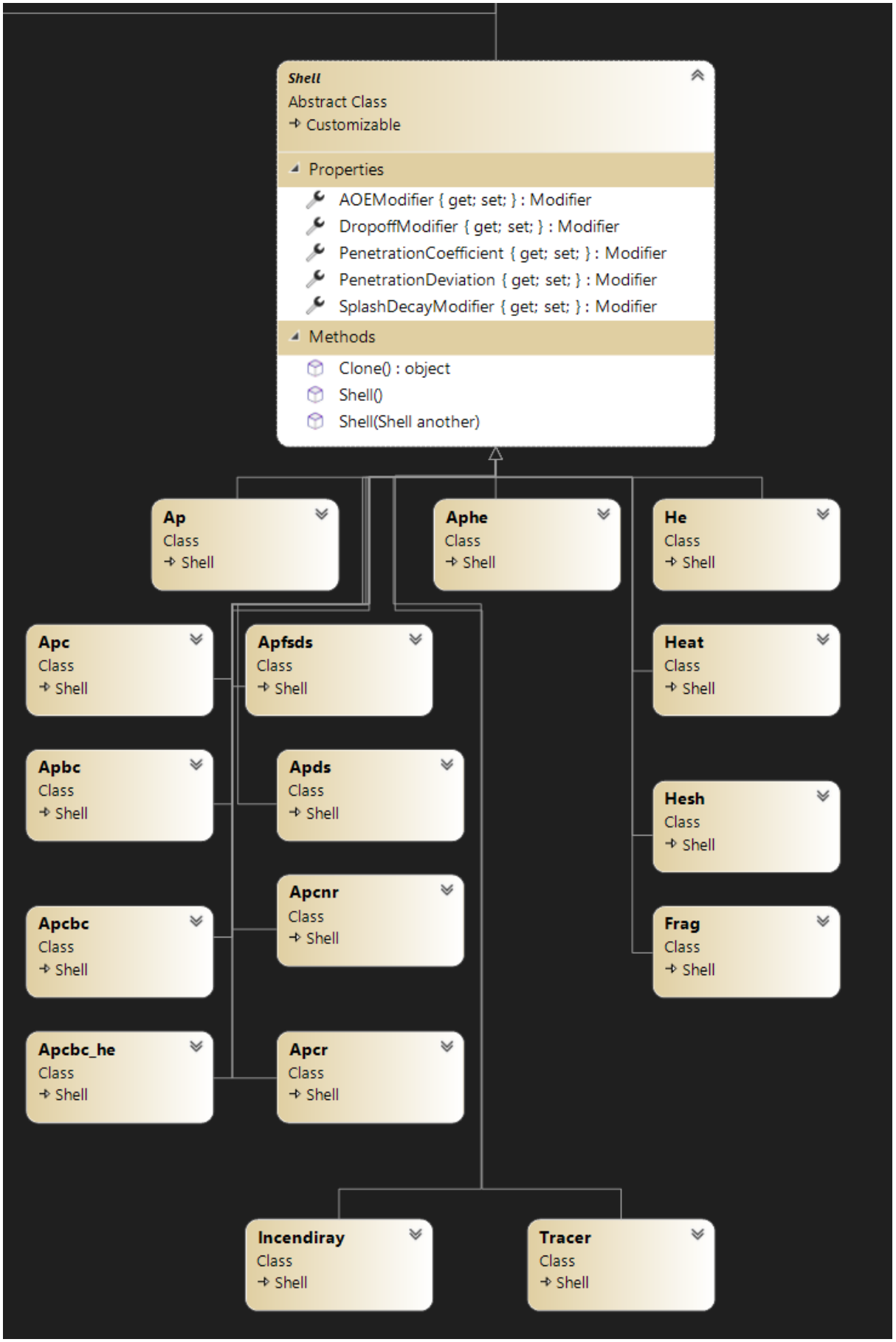
The Firearm class and all of its subclasses

10.4.2.2. Module Classes



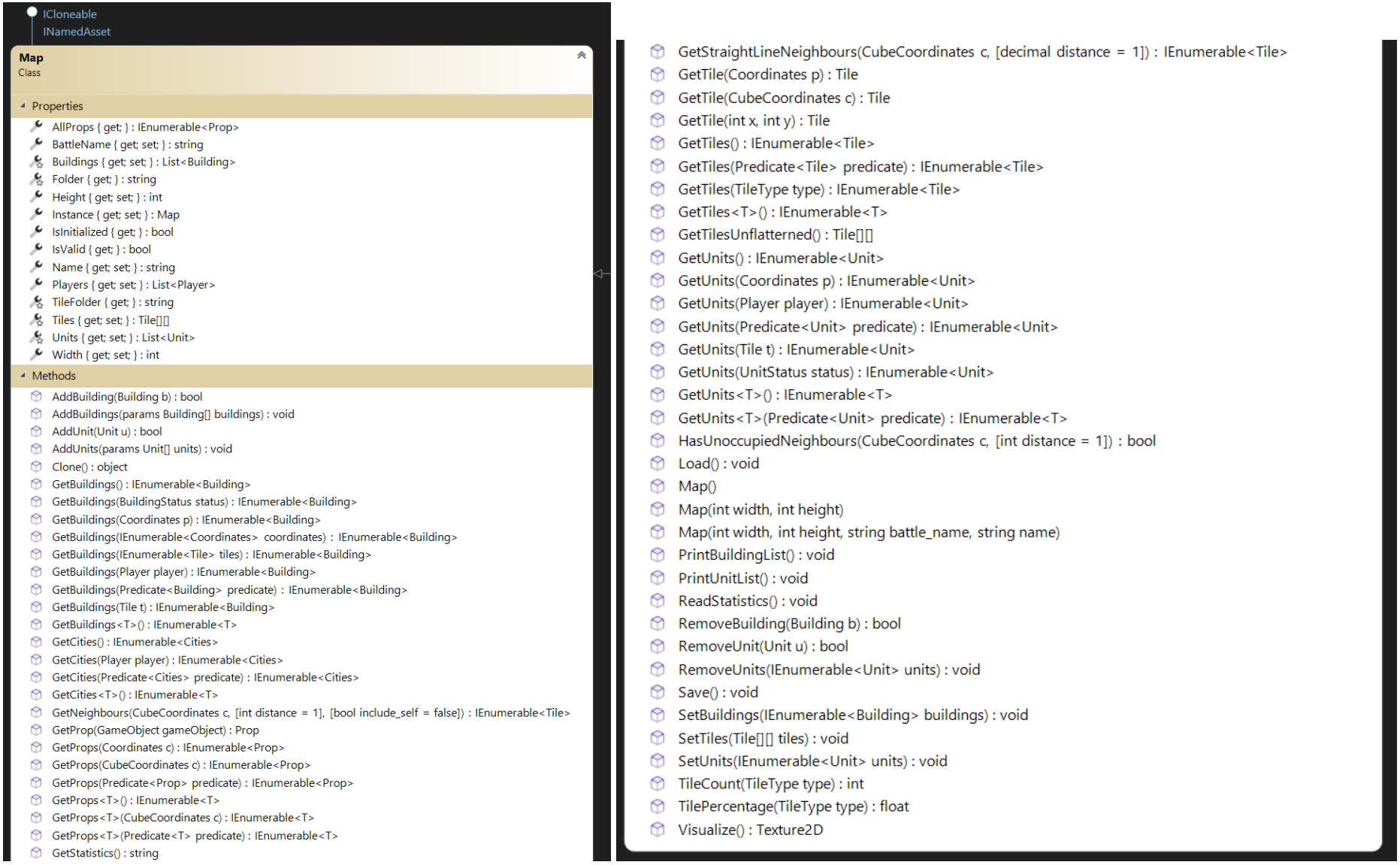
The Module class and all of its subclasses

10.4.2.3. Shell Classes

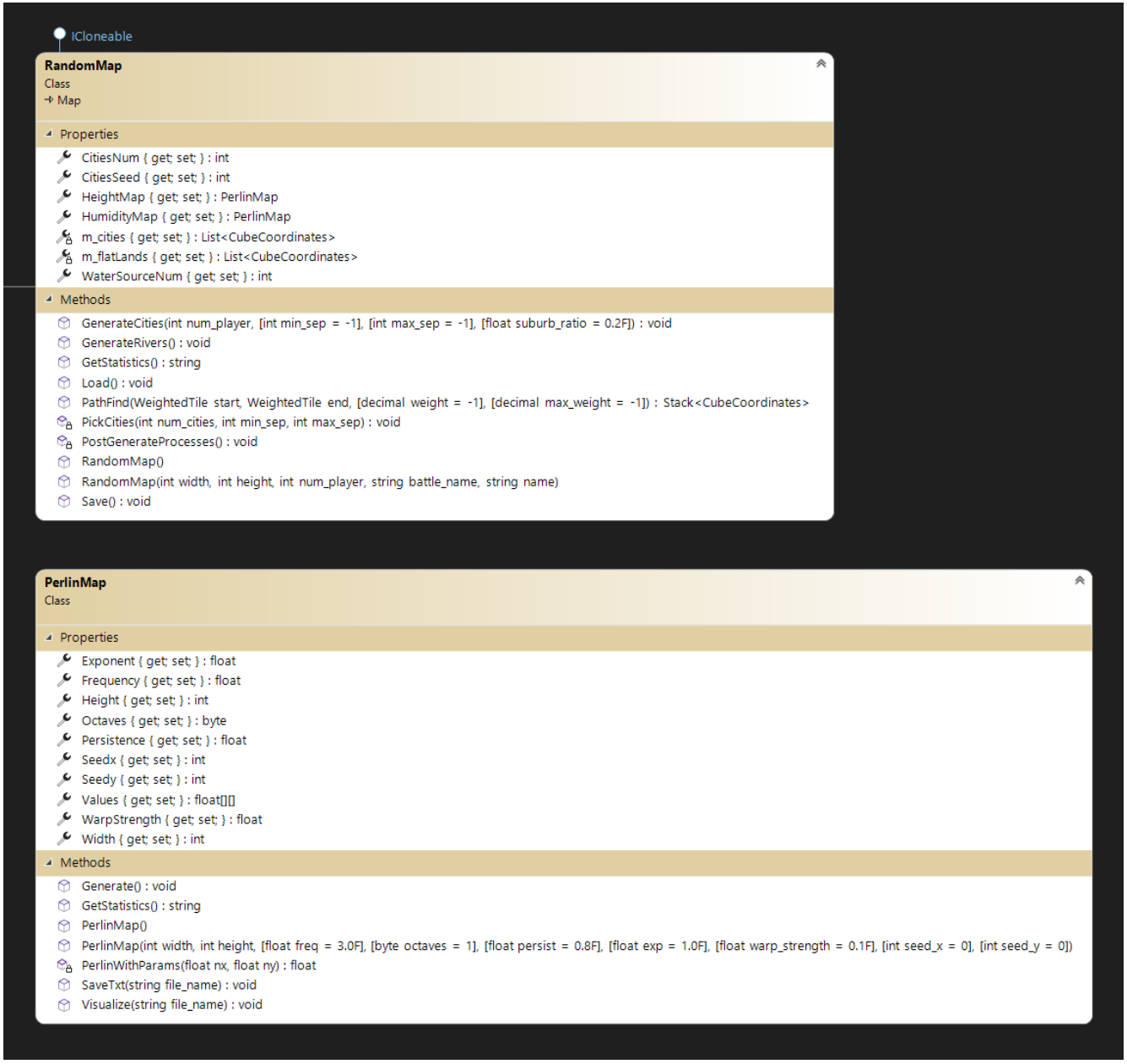


The Shell class and all of its subclasses

10.5. Map Classes

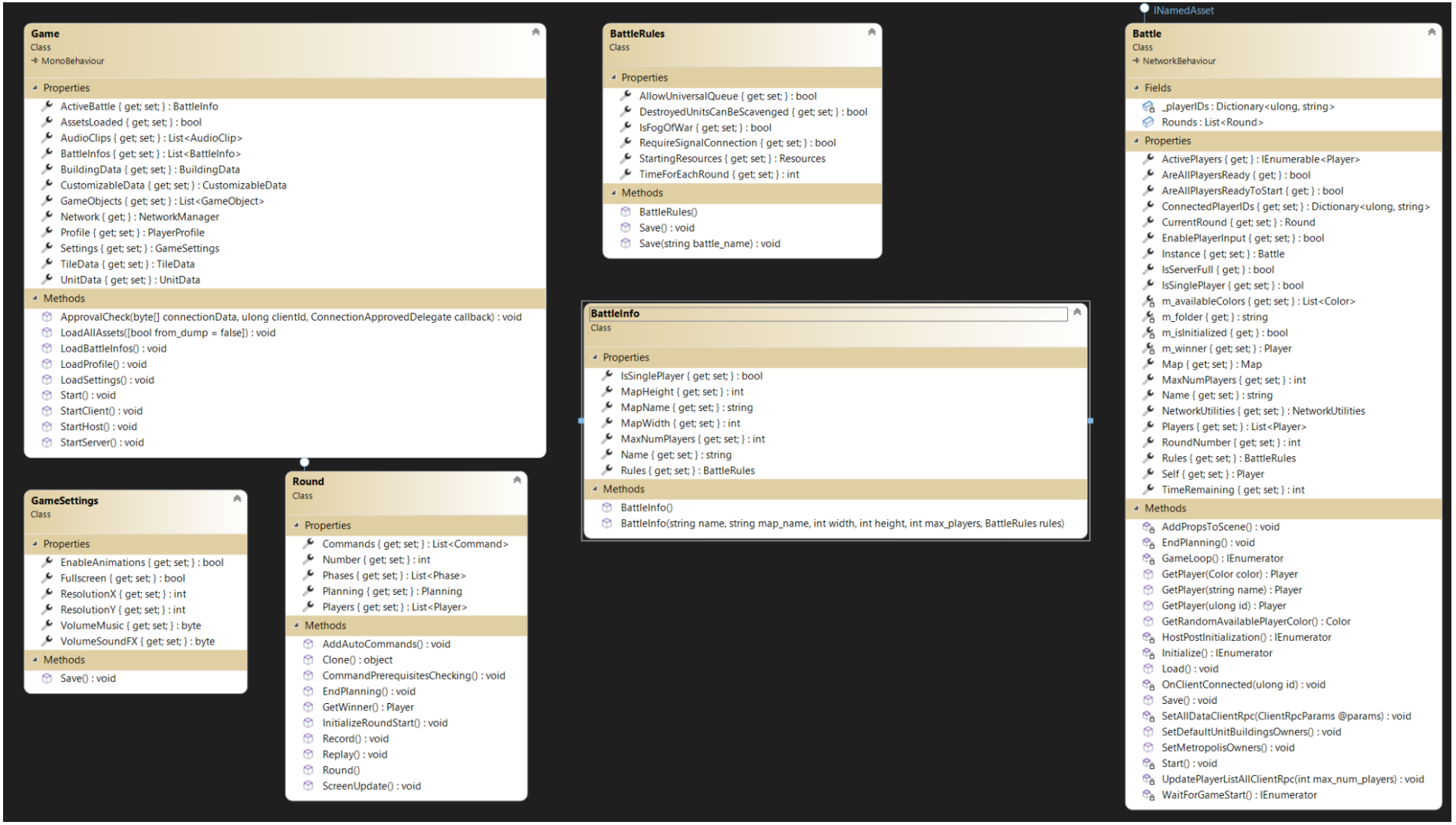


The Map class.



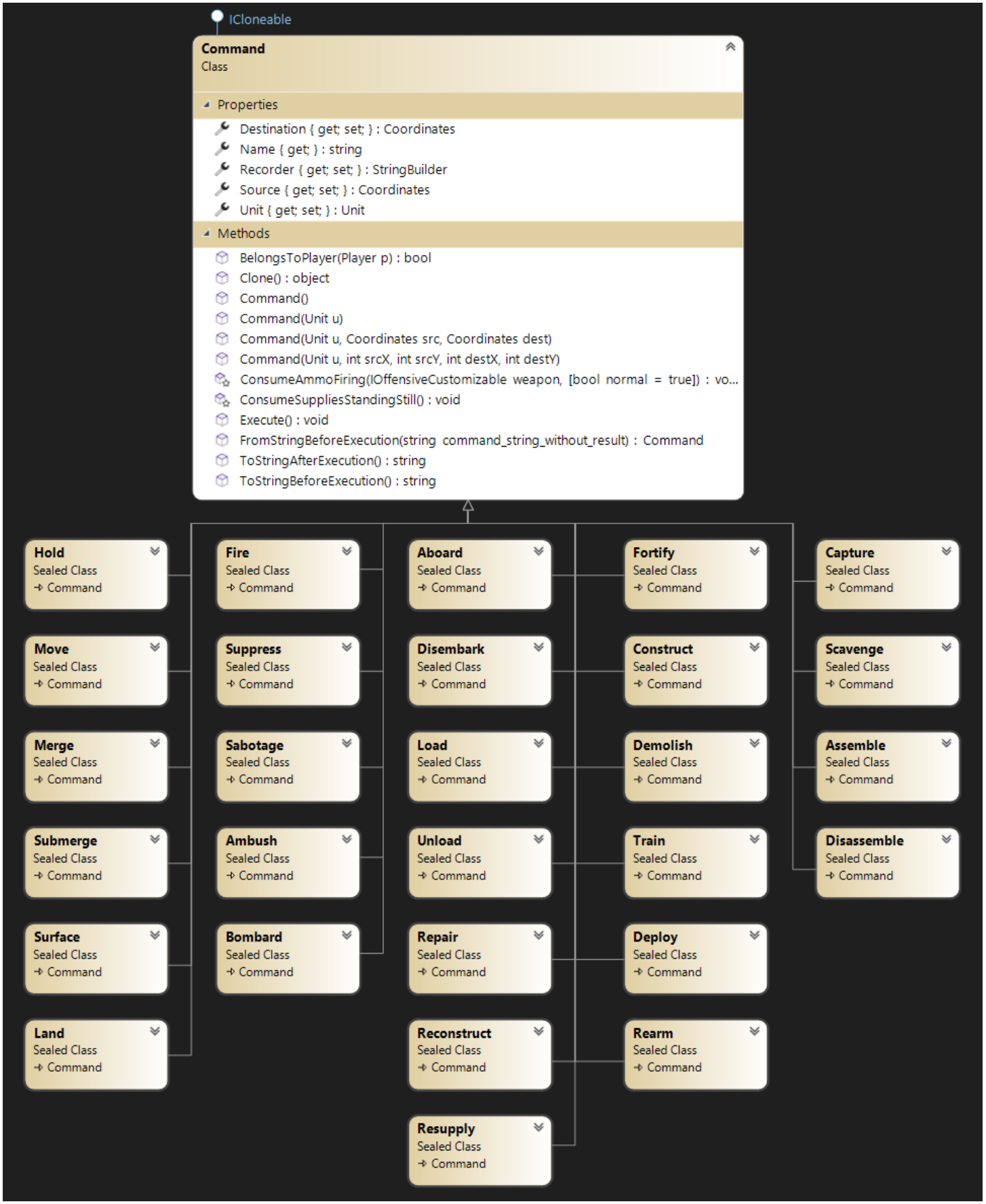
The RandomMap class (inherit from Map class) and the PerlinMap class.

10.6. Game Flow Classes



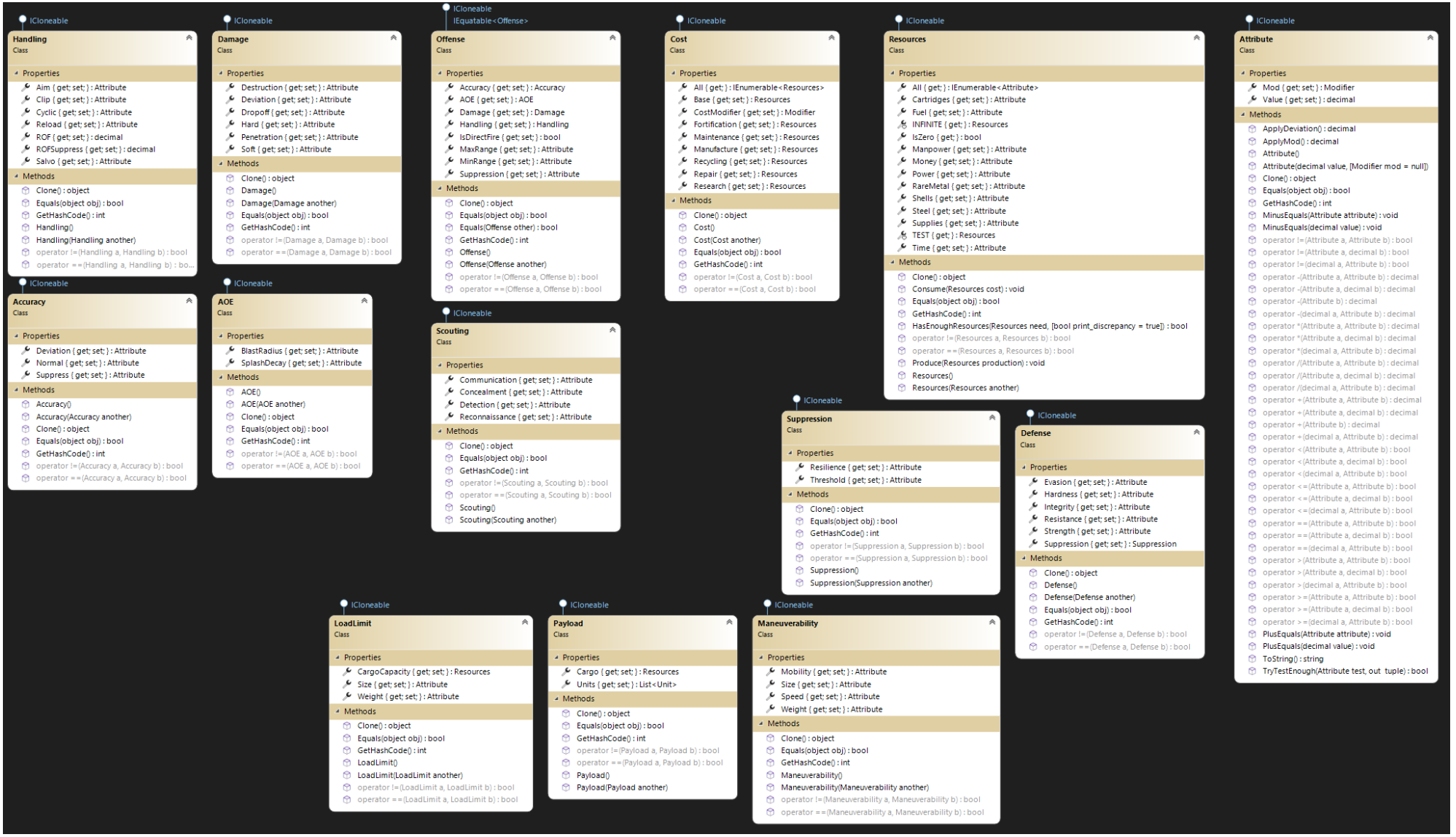
Some classes which are vital to the flow of the game.

The phase class and all its subclasses. Commands will be executed when the Execute() method is called. Server side only.

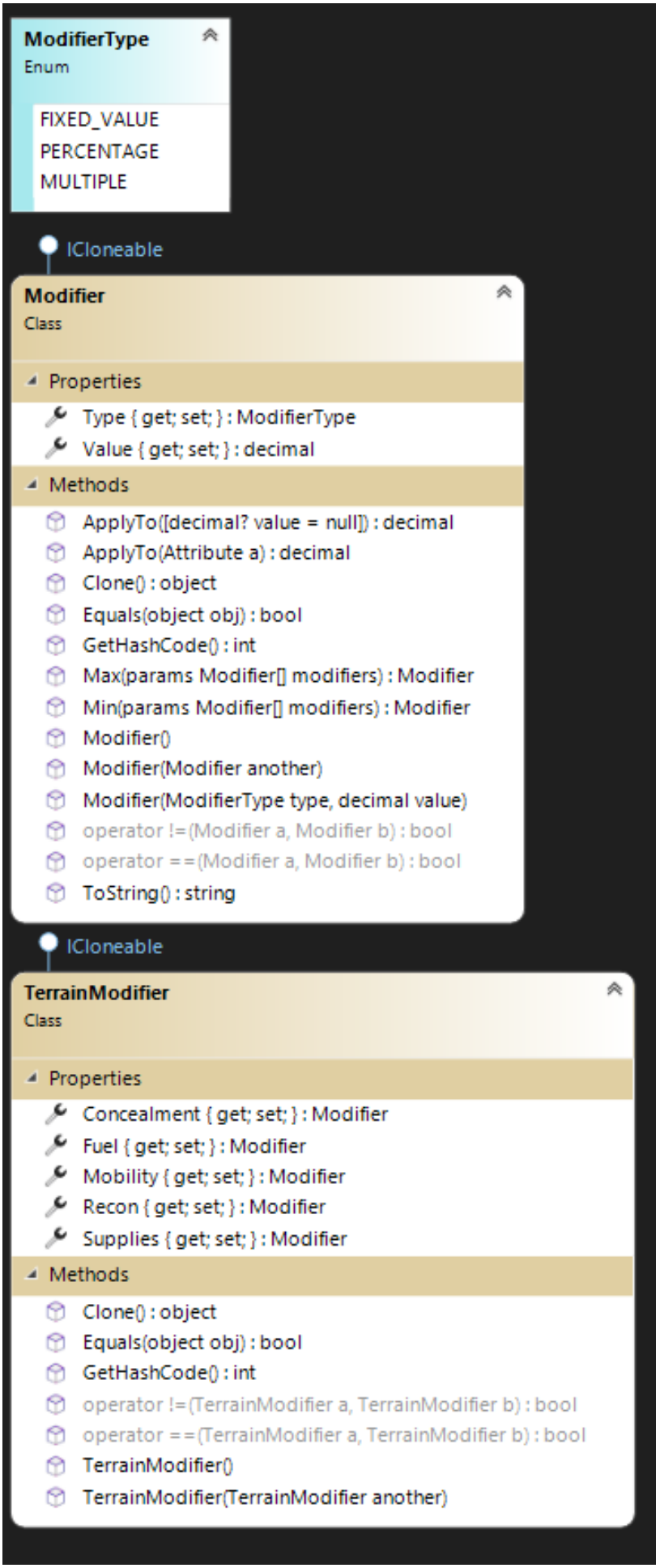


The command class and all of its subclasses. Strings, instead of the command objects, are passed via the network. It's vital to convert to string first before sending them through the network.

10.7. Attribute and Modifier Classes

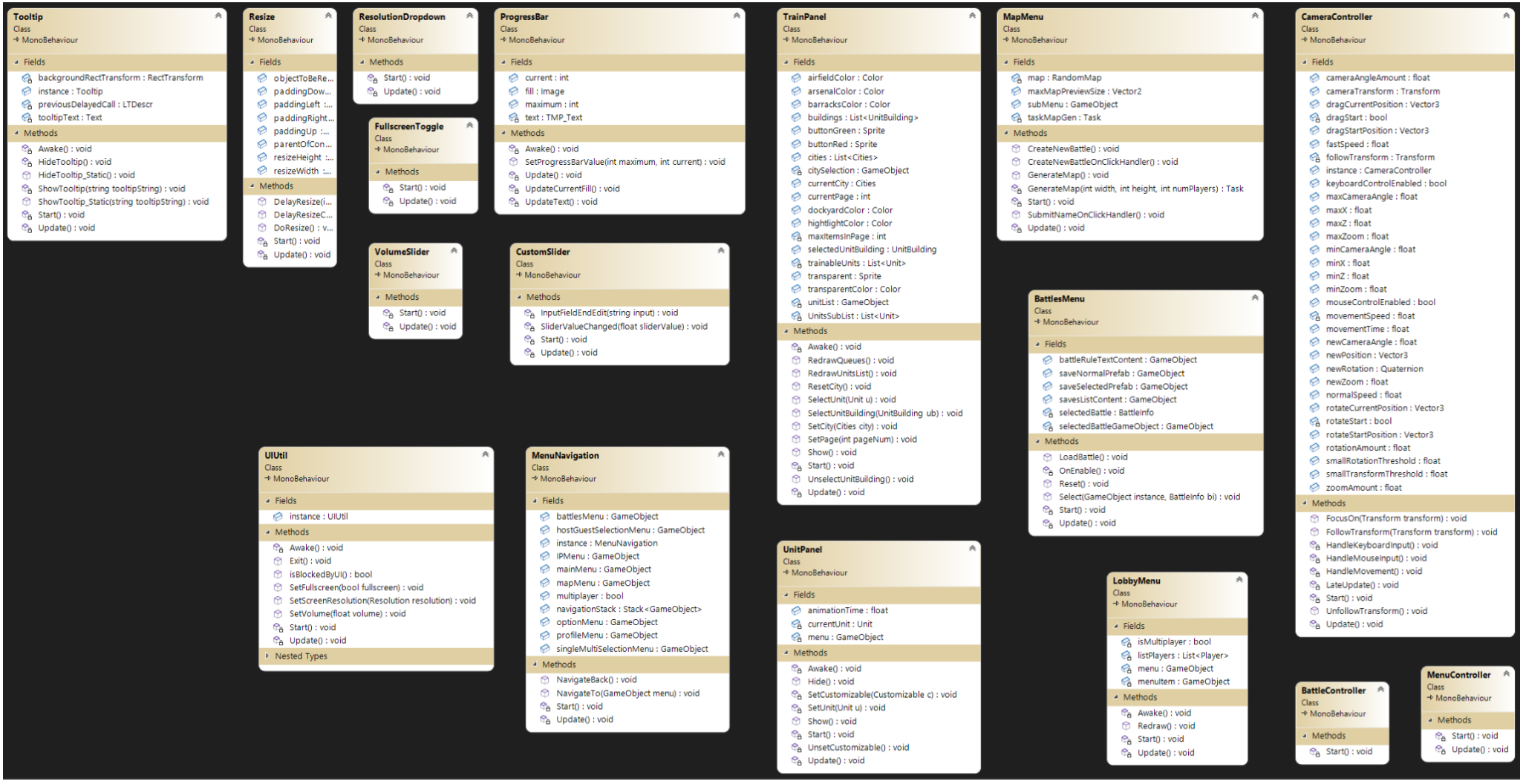


The Attribute class and all sub-set classes of attributes. Operators are overloaded for attributes for direct manipulations of the underlying values.



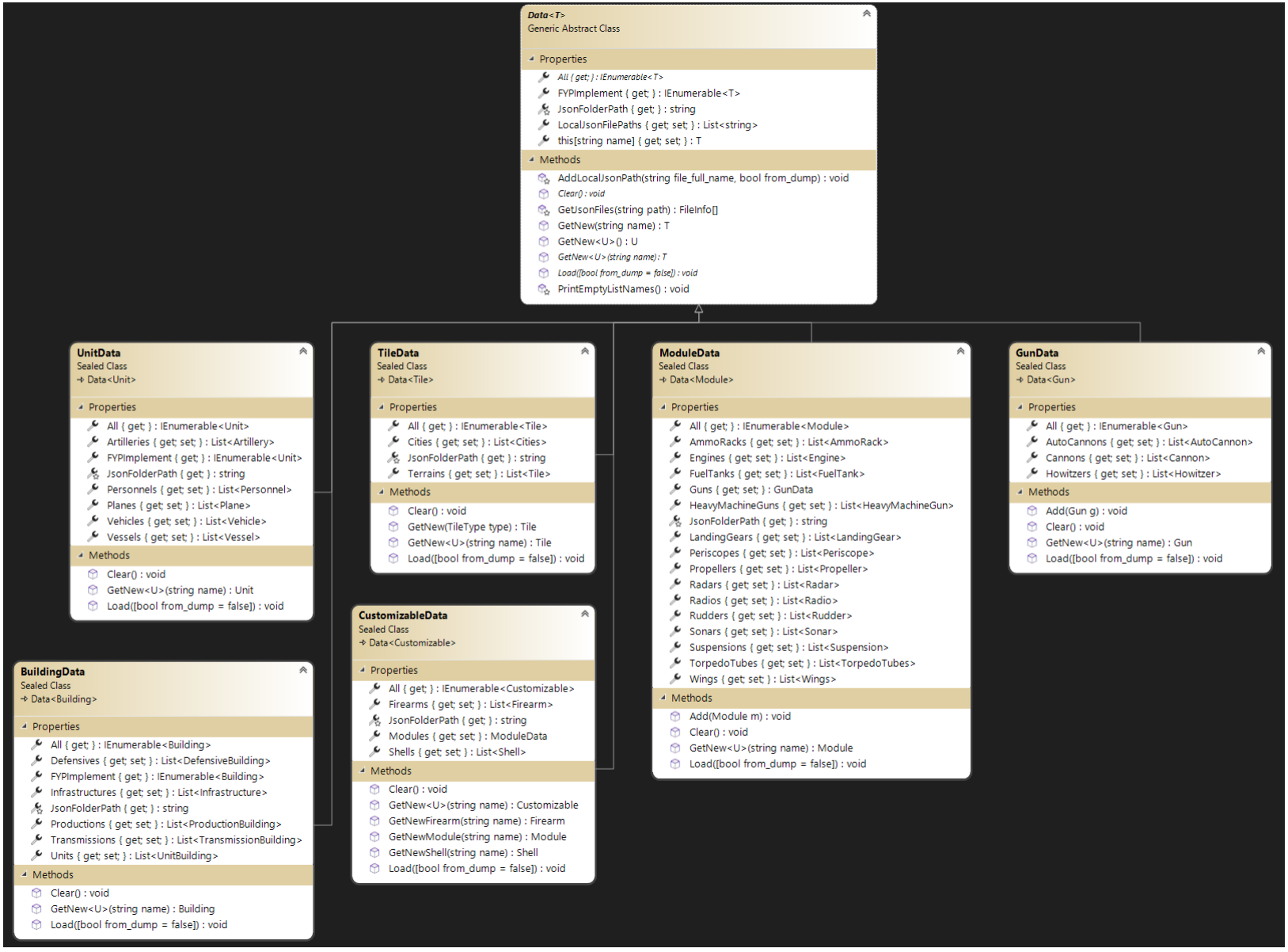
The Modifier class and the TerrainModifier class

10.8. UI Classes

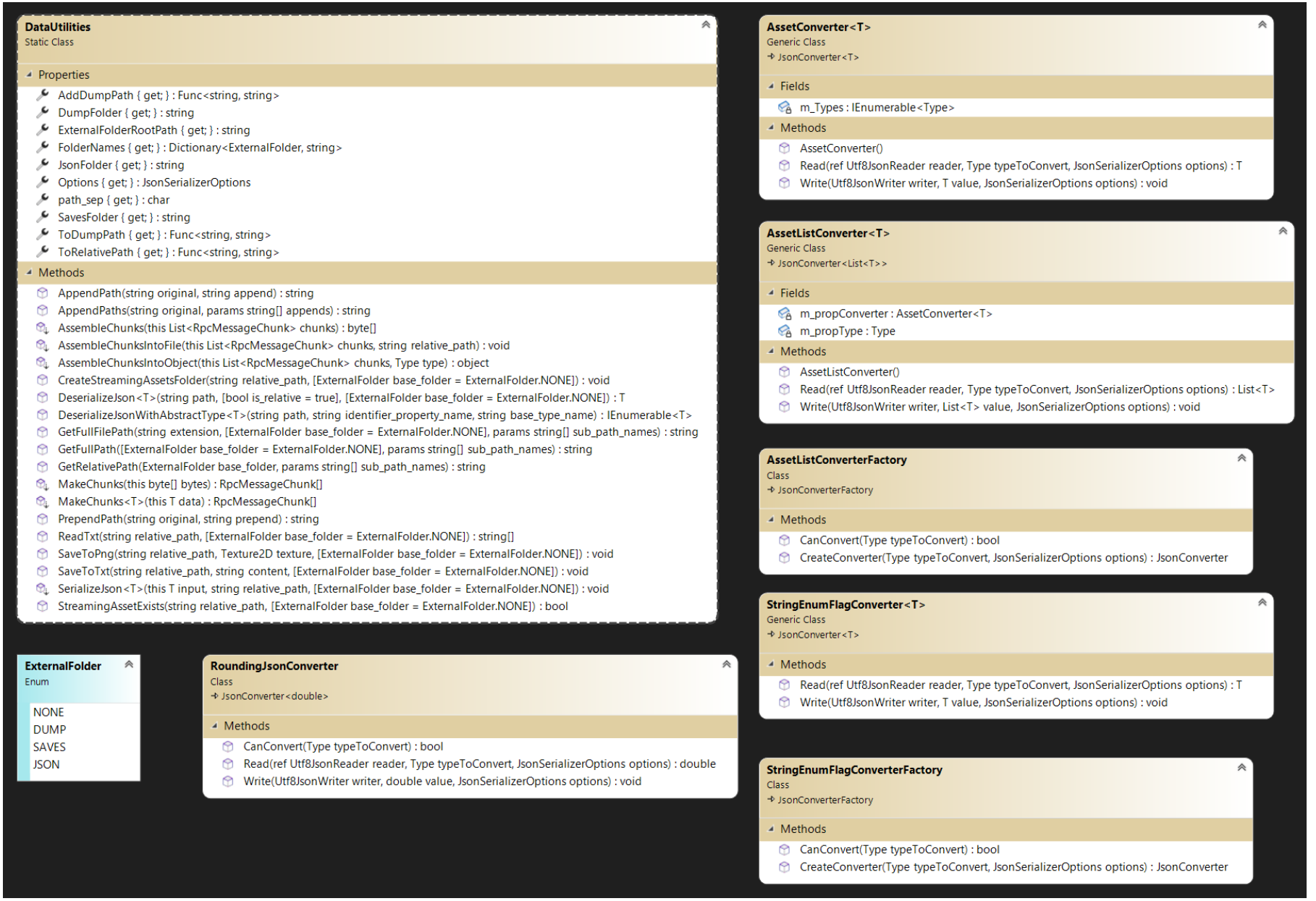


UI and controller classes. All of them inherit MonoBehaviour. Mainly used for manipulating game objects on scenes

10.9. Data Classes

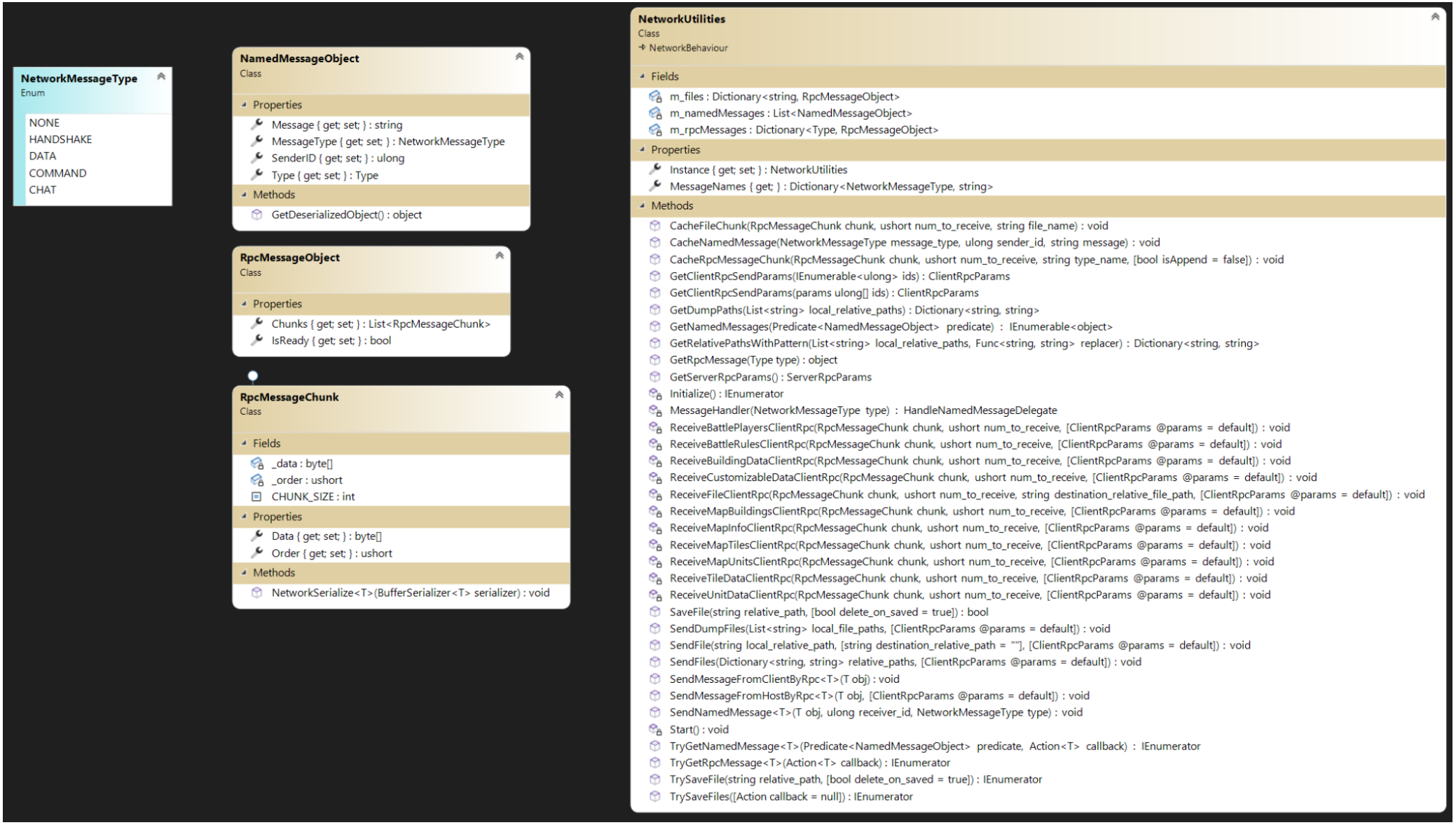


Generic abstract class Data and all of its subclasses



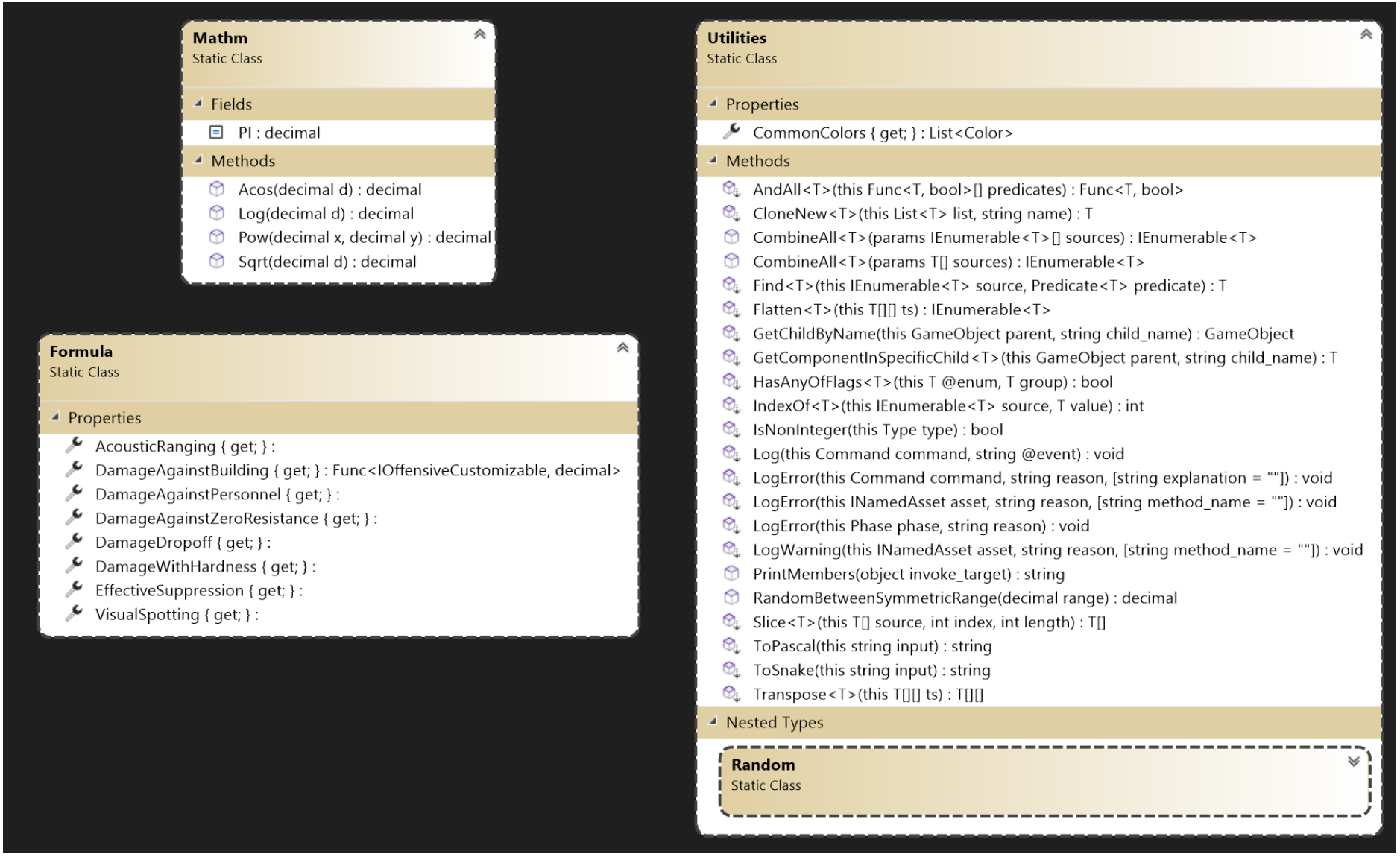
DataUtilities class and all custom Json Converters. It is necessary to implement such converters as deserialization of abstract classes is not available from the default Json Converter.

10.10. Networking Classes



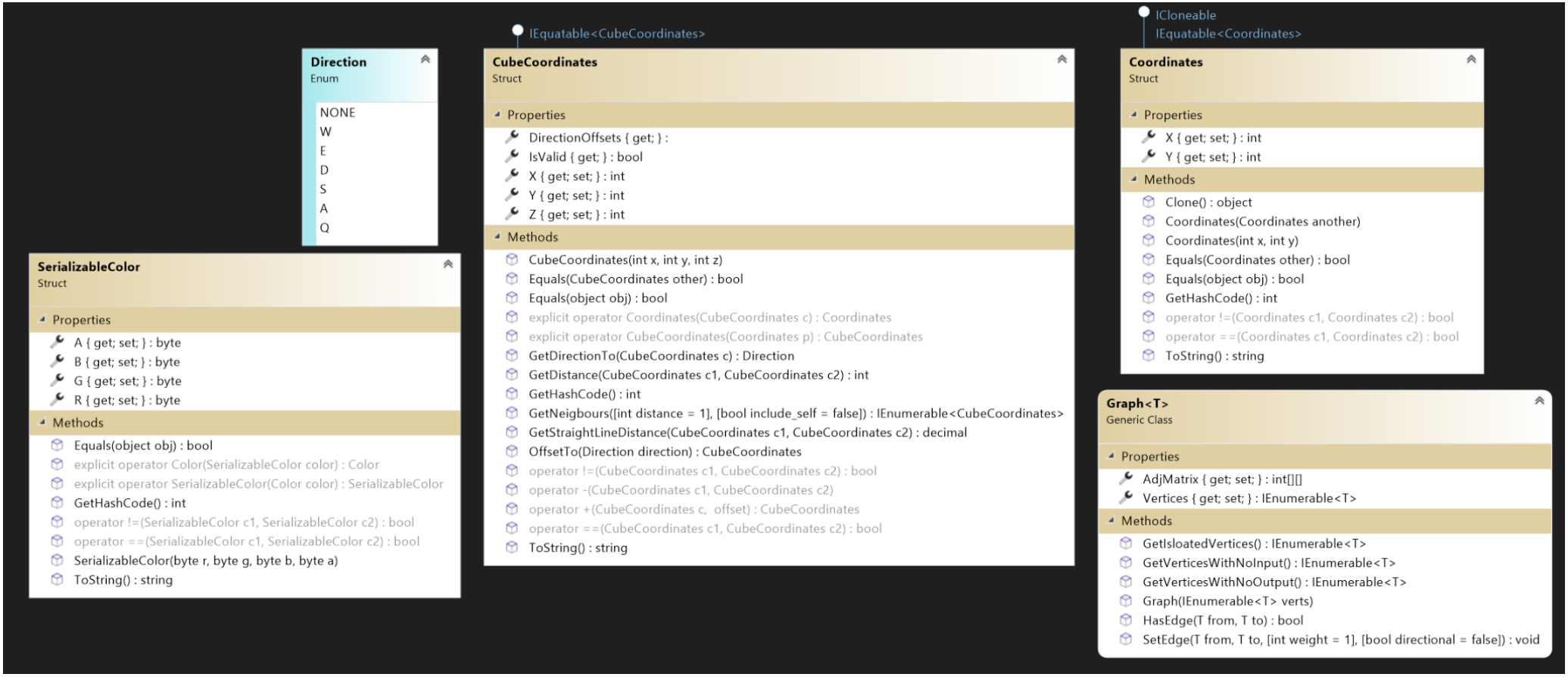
The classes served for communications between server and clients

10.11. Utility Classes



Utility classes implemented for handling error messages, list and array operations, and storing mathematical functions of formula used in the game.

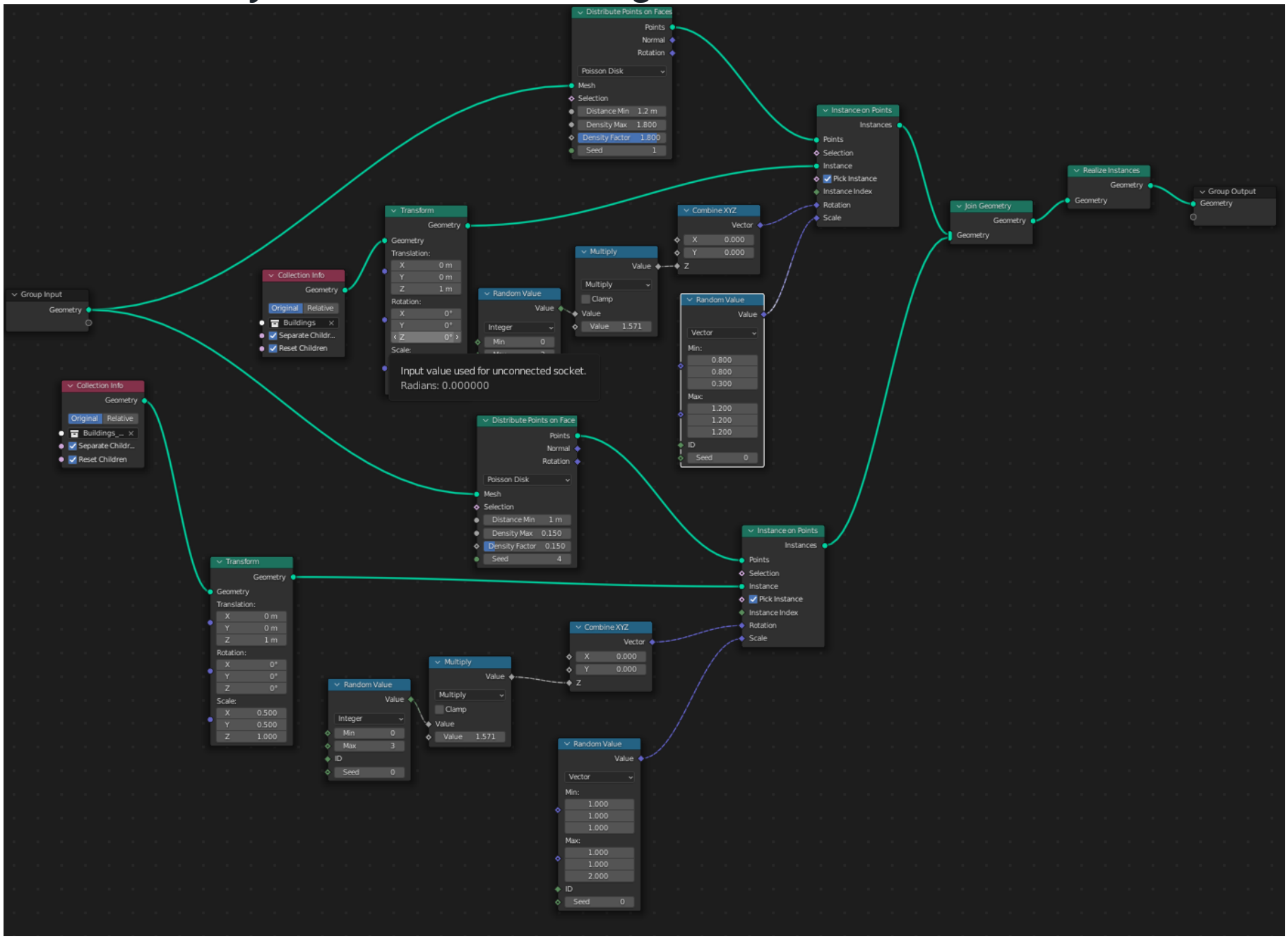
10.12. Custom Types



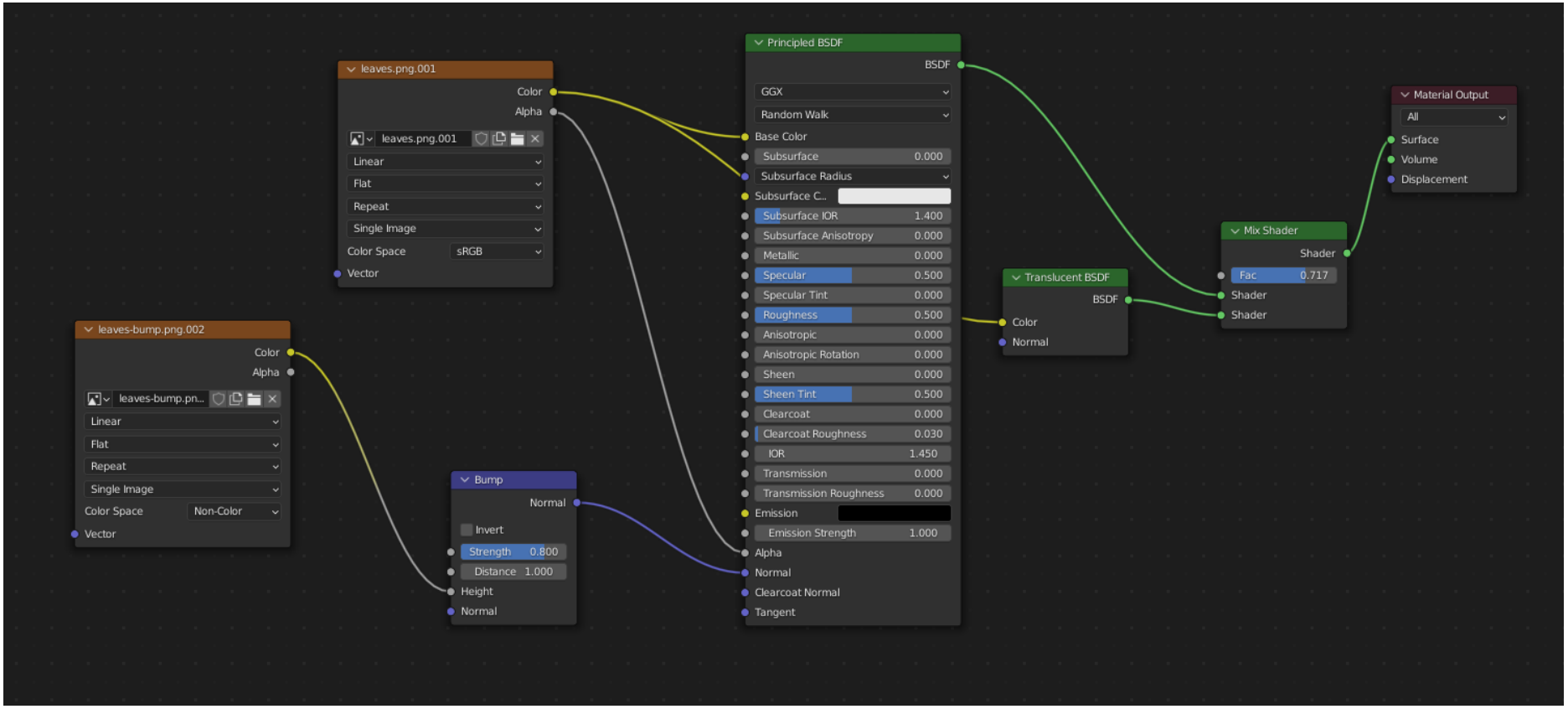
Custom types implemented for convenience of calculations and serialization. Unity's Color struct cannot be serialized and it's necessary to create a serializable struct for storing the color of the players.

11. Appendix D: Gallery

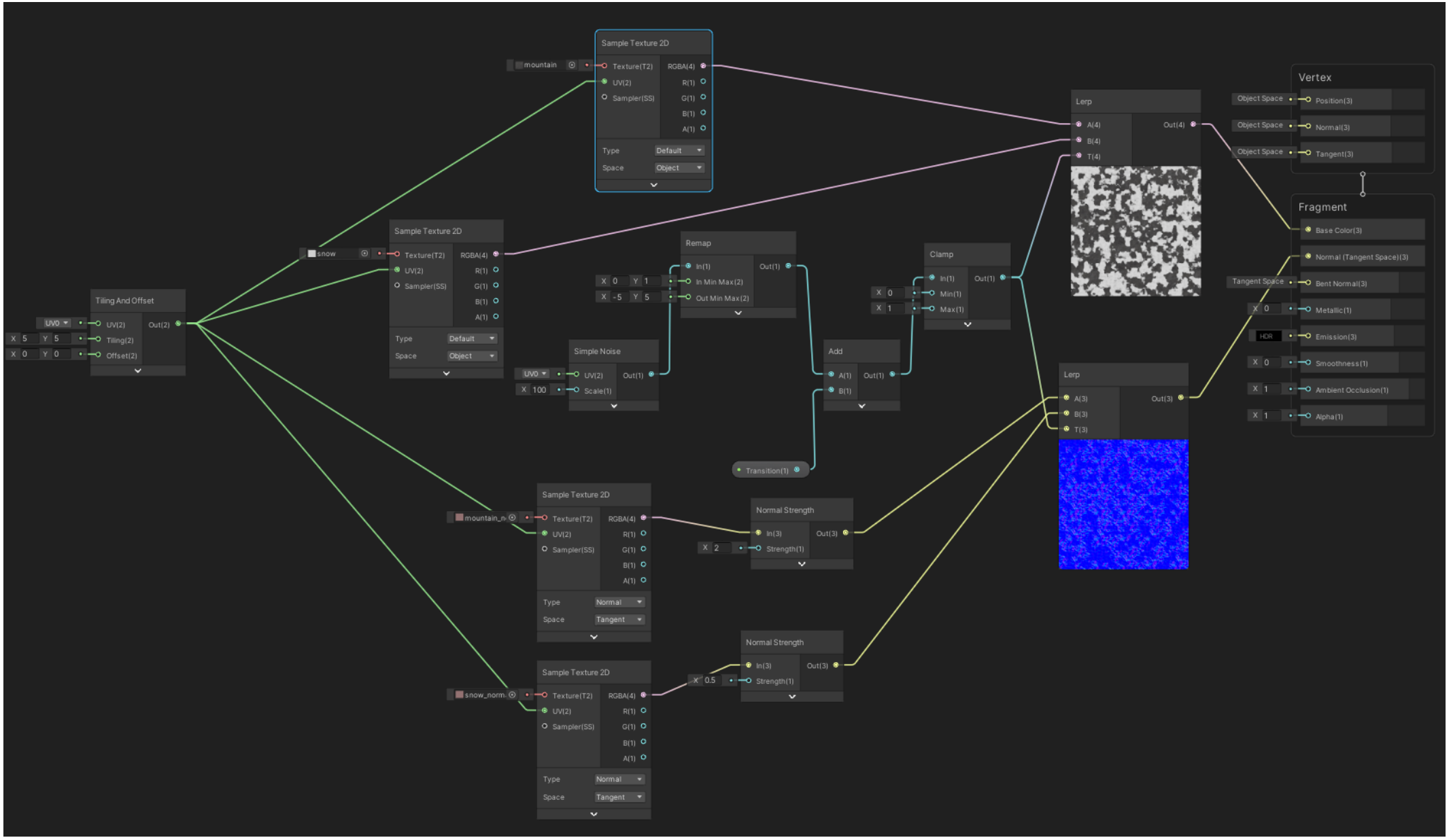
11.1. Geometry Nodes for cities models generation



11.2. Shader Nodes for leaves



11.3. Shader Graph for Mountains tile



11.4. Shader Graph for grasses