POLISH-JAPANESE ACADEMY
OF INFORMATION TECHNOLOGY

Creating the Haardon: a fast hybrid of a first person shooter and a real time strategy.

Kamil Majchrzak S19568
Marcin Piotrkowicz S22222
Wiktor Orzołek S22437
Batuhan Seyhan S20343
Mateusz Sasor-Adamczyk S22454

eXtended Reality and Immersive Systems Department
Polish-Japanese Academy of Information Technology
Thesis prepared under the supervision of:
Supervisor:
Rafał Masłyk, MScEng
Reviewer:
Barbara Karpowicz, MScEng

Warsaw, January 27, 2024

POLSKO-JAPOŃSKA AKADEMIA
TECHNIK KOMPUTEROWYCH

Tworzenie Haardon: szybkiej hybrydy strzelanek pierwszoosobowych ze strategią czasu rzeczywistego.

Kamil Majchrzak S19568
Marcin Piotrkowicz S22222
Wiktor Orzołek S22437
Batuhan Seyhan S20343
Mateusz Sasor-Adamczyk S22454

eXtended Reality i Systemy Immersyjne
Polsko-Japońska Akademia Technik Informatycznych
Praca przygotowana pod nadzorem:
Promotor:
Rafał Masłyk, MScEng
Recenzent:
Barbara Karpowicz, MScEng

Warszawa, 27 stycznia 2024

Abstract. First person shooters are a very popular type of computer games. The goal was to create a hybrid mix of game that consists of an first person shooter combined with Real Time strategy elements. The computer game "Haardon" was in Unity cross-platform game engine. The terrain of "Haardon" is a procedurally generated world [17] with randomly generated objective that will show up later in the game and will be determined randomly every game, on a random locations. There are two factions: Humanity and Aliens. Player controls mankind and most of the gameplay focus on expanding, defending locations and gathering resources scattered across the map. The player spends the resources on almost everything - crafting items, training units, upgrading units and on building structures. The game flow contains two phases: building phase, focused on the RTS part and the combat phase (fight in FPS, ordering units in RTS with possibility to switch mode whenever Player want to) Building phase ends after some time, fighting phase ends when the player repels the enemy.
Assets such as 3D models for characters, environment and textures are mostly free-to-use game assets from websites like Open Game Art and some of them have been purchased in Unity assets store by the team members.

Keywords: FPS-RTS Hybrid · Procedural and Random Terrain Generation · C# · Science-Fiction · Game · Unity

Streszczenie Gry typu FPS należą do jednego z najpopularniejszych gatunków gier komputerowych. Celem projektu jest utworzenie hybrydy gry FPS i strategii czasu rzeczywistego. Gra „Haardon" powstała na międzyplatformowym silniku Unity. Teren gry jest generowany proceduralnie [17] z losowo wygenerowanymi celami, które zostaną wylosowane na początku każdego rozpoczęcia nowej gry. Gra zawiera w sobie dwie frakcje, ludzi i kosmitów. Gracz prowadzi siły ludzkości w misji, której głównymi elementami jest ekspansja, obrona lokacji i zbieranie zasobów rozrzuconych po mapie. Wszystkie akcje, takie jak: szkolenie jednostek, ulepszanie i budowanie budynków będą kosztowały gracza zbierane zasoby. Gra przebiega w dwóch fazach, faza budowania skupiona na elementach RTS i faza walki(w której to gracz będzie miał możliwość zmiany pomiędzy zarządzaniem jednostkami z lotu ptaka, a przełączeniem na tryb FPS). Faza budowania jest ograniczona czasowo, a faza walki kończy się z pokonaniem ostatniego przeciwnika. Zasoby, takie jak modele 3D postaci, otoczenia czy tekstury są głównie darmowymi zasobami pobranymi ze stron takich jak Open Game Art, natomiast część z nich została zakupiona na Unity Asset Store przez zespół.

Keywords: FPS-RTS Hybrid · Proceduralna Generacja Świata · C# · Science-Fiction · Game · Unity

Table of Contents

1 Introduction

1.1 Context

"Haardon" is set in a distant future where humanity has expanded into the stars and is engaged in a struggle for dominance across the galaxy. The player takes on the role of a commander who must lead their faction to victory by conquering planets, managing resources, and engaging in tactical combat against other factions. The game features both first-person shooter and real-time strategy gameplay elements, providing a unique and immersive experience for players.

1.2 Goals of the work

The primary goal of "Haardon" was to provide players with an engaging and challenging experience that combined elements of both first-person shooter and real-time strategy games. The game aimed to be accessible to players of all skill levels while also providing depth and complexity for more experienced players. The game also sought to provide a compelling storyline that immersed players in the game's universe and characters.

1.3 Motivation

The motivation behind the creation of "Haardon" was to provide players with a unique and exciting gameplay experience that combined elements of two popular genres. There was a gap in the market for a game that successfully merged first-person shooter and real-time strategy gameplay, and the game aimed to fill that gap. The game also sought to provide a rich and immersive universe for players to explore, with a compelling story and memorable characters that would keep players engaged for hours on end.

2 Game Design

2.1 General idea

The general idea of the game was to create an immersive first-person shooter (FPS) and real-time strategy (RTS) hybrid game, set in a post-apocalyptic world where players had to gather resources, build bases, and command armies to survive and defeat enemy factions. The game featured a single-player mode, with players able to customize their characters, weapons, and bases to suit their playstyle.

2.2 Target Audience

The target audience for this game was primarily gamers who enjoyed FPS and RTS games, as well as those interested in post-apocalyptic settings and base-building mechanics. The game was targeted towards players aged 18 and above due to its mature themes, violence, and language.

2.3 Genre

The game is a hybrid of FPS and RTS genres, combining elements of both gameplay styles. The FPS aspect of the game will focus on combat and exploration, while the RTS aspect will involve base-building, resource management, and strategic planning.

2.3.1 Warhammer Dawn of War

Warhammer Dawn of War is a series of Real-Time Strategy games set in the world of Warhammer 40.000, a grim and dystopian version of humanity's future set in the 41st milenium. The series allow the player to take controll of Space Marines, genetically modified super-soldiers in powerful armour. The player can build his base and train units and send them to complete objectives, with special characters being the centre of the player's forces.

2.3.2 Warcraft III

Warcraft III Reign of Chaos is a high fantasy real-time strategy game released in 2002. It's main game loop consisted of building base which in turn allowed for training units, buying upgrades and gathering resources. More importantly each faction had a limited access to a special unique unit. This, so called hero unit was powerful, had many abilities and could respawn after death. In story missions important characters were those hero units. Haardon contains many mechanics from Warcraft III which are emblematic to the genre as a hole however the hero units were inspiration for a main character the player controls in FPS mode. Haardon takes principle behind hero units and puts it in the main stage, resulting in a mixture of RTS and FPS.

2.3.3 Halo Series

The magnetic accelerator cannon also known as MAC, is type of heavy weapon system that exists in Halo Series. They are designed to fire ferric-tungsten round at supersonic velocities from the bow of a capital ship, orbital defense platform or mounted driver emplacement. This coilgun use one or more coils used as electromagnets in the configuration of a linear motor that accelerate a projectile to high velocity. During the first project meetup the team discussed different abilities for a player that would help overpower him and Haardon have similar ability for the player called Orbitral Airstrike aka Airstrike. Whenever player does use Airstrike ability he waits about a second and there is the projectile comming from the Sky to the place pointed by the Player. Projectile is really strong and it can also crush terrain.

2.4 Mechanics

In "Haardon", players are tasked with leading their faction to victory by conquering planets, managing resources , and engaging in tactical combat against enemy factions in a post-apocalyptic world.The game combines both first-person shooter and real-time strategy gameplay elements, providing players with a truly unique and immersive experience. In order to survive, players must gather resources such as food, water, and materials to build their bases and create their armies. They can also customize their weapons, and bases to suit their individual playstyle, giving them a greater sense of ownership and control over their gameplay experience. The real-time strategy gameplay element requires players to manage their resources and build their bases strategically, as well as command their armies during battle. Meanwhile, the first-person shooter gameplay element offers player an up-close and personal experience of the combat and action.

3 Tools and methods

3.1 Game Engine (e.g., Unity)

For the development of the game, Unity game engine was used. Unity is a widely used game engine that supports the development of games across various platforms, including Windows, macOS, Android, iOS, and web-based platforms. Unity provides a wide range of features, such as 2D and 3D graphics rendering, physics simulation, audio, scripting, and user interface development.

3.1.1 Animator The Animator tool, provided by Unity was used for animation in the game. The Animator tool was used for creating complex animations and managing animation transitions in the game. It provided a visual editor for creating and modifying animation states, blending animations, and controlling the animation flow.

3.2 Save System

Save System [3] allows the player to save the progress of the game between sessions or to go back to a specific moment of the playthrough. It has a modular design, which greatly simplifies its implementation into classes that require it.

Save System consists of:

- **SavingLoadingSystem** - a class attached to the game object which is responsible for collecting data to save, managing the save path and passing the data to the appropriate classes when loading the save
- **ISaveable** - an interface that allows to easily manage the save classes and provides the impelmentation of the methods required to do so
- **SaveableEntity** - an intermediate class that is a module attached to the objects in the game that are to be saved
- **SaveData** - an intermediate struct [15] for saving and loading data, individually implemented by classes implementing the ISavable interface.

SavingLoadingSystem

This system, at the time of saving, searches for all objects in the game that have the "SaveableEntity" modulus, then using the "CaptureState" method from this modulus saves the data contained in the lists received from the "SaveableEntity" to a dictionary in which the key is the Id of the given "SaveableEntity" and the data are the dictionaries from the individual objects. In this way, an organized collection of all objects is created to be saved with data assigned to them accordingly.

```
private void CaptureState(Dictionary<string, object> state)
{
    foreach (var saveable in FindObjectsOfType<SaveableEntity
    >())
    {
         state[saveable.Id] = saveable.CaptureState();
    }
}
```

The class then proceeds to serialize [9] the aforementioned dictionary into a binary form which is saved to the save file selected by the player or creates a new one if necessary.

```
private void SaveFile(object state)
{
    using (var stream = File.Open(_savePath, FileMode.Create)
    )
    {
        var formatter = new BinaryFormatter();
        formatter.Serialize(stream, state);
    }
}
```

When loading a save, the situation is analogous but reversed. When the player selects a save, the path to the file is written to a variable. The file is then opened and its contents are read by the "BinaryFormatter", deserialized [9] and treated as a dictionary with Id as the key and object as the value.

```
private Dictionary<string, object> LoadFile()
{
    if (!File.Exists(_savePath))
    {
        return new Dictionary<string, object>();
    }

    using (FileStream stream = File.Open(_savePath, FileMode.
    Open))
    {
        var formatter = new BinaryFormatter();
        return (Dictionary<string, object>) formatter.
    Deserialize(stream);
    }
}
```

The "SavingLoadingSystem" class then finds for each "SaveableEntity" in the dictionary the data assigned to it based on the Id and passes it through the "RestoreState" method.

```
private void RestoreState(Dictionary<string, object> state)
{
    foreach (var saveable in FindObjectsOfType<SaveableEntity
    >())
     {
        if (state.TryGetValue(saveable.Id, out object value))
          {
              saveable.RestoreState(value);
         }
    }
}
```

In addition, the class "SavingLoadingSystem" is responsible for notifying other classes when the currently selected save is changed or when the list of all saves is modified. It does this by passing the modified variables via an event [12]. This is especially important for UI elements responsible for displaying and selecting saves.

```
public static event Action<List<string>> OnSavesListChange;
public static event Action<string> OnCurrentSaveChange;
```

SaveData

The implementation of these struct's [15] varies depending on the class of which data it stores. Since it is a serializable [9] struct [15] every time, it never contains reference variables. As an example, consider its implementation for the "GameFazeSystem" class responsible for managing game phases. Between sessions, such data as the current game phase, time left to change, increment rate, phase duration and maximum phase duration should be stored for this class. In the "GameFazeSystem" class, the "gameFaze" variable is stored as an enum, but as "SaveData" does not accept reference types it is stored as an int.

```
[Serializable]
private struct SaveData
{
    public int gameFaze;
    public float fazeTimeLeft;
    public float growthRate;
    public float fazeDuration;
    public float maxFazeDuration;
}
```

SaveableEntity

A module attached to objects in the game that have components implementing the ISavable interface. It has an Id field generated using the "NewGuid()" [2] method from Unity with the help of which its instances receive corresponding data from the "SavingLoadingSystem" class when loading a save.

```
[SerializeField] private string id = string.Empty;
public string Id => id;

[ContextMenu("Generate Id")]
private void GenerateId() => id = Guid.NewGuid().ToString();
```

When saving, the module finds all components of the object to which it belongs that implement the ISavable interface, and then saves them to a dictionary in which the key is the type of the class and the value is "SaveData" belonging to it. Then the dictionary prepared this way is passed as an object to the "SavingLoadingSystem" class.

```
public object CaptureState()
{
    var state = new Dictionary<string, object>();

    foreach (var saveable in GetComponents<ISaveable>())
    {
        state[saveable.GetType().ToString()] = saveable.
    CaptureState();
    }

    return state;
}
```

Loading a save starts by receiving a data object from the "SavingLoadingSystem" class, which is casted into a dictionary form that allows the data in it to be decoded. "SaveableEntity" then finds the components of the object to which it belongs that implement the ISavable interface and, using the "RestoreState" method, passes them the corresponding "SaveData" from the dictionary.

```
public void RestoreState(object state)
{
    var stateDictionary = (Dictionary<string, object>) state;

    foreach (var saveable in GetComponents<ISaveable>())
    {
        string type = saveable.GetType().ToString();

        if (stateDictionary.TryGetValue(type, out object
    value))
        {
            saveable.RestoreState(value);
        }
    }
}
```

ISaveable

An interface implemented by components/modules whose data should be stored between gameplay sessions. It forces the classes implementing it to have two methods "CaptureState" and "RestoreState" responsible for saving and loading class variables, respectively.

```
public interface ISaveable
{
    object CaptureState();
    void RestoreState(object state);
}
```

Its implementation varies drastically depending on the needs of the class. For example, in the "GameFazySystem" class, the "CaptureState" method is very simple because in addition to casting the "CurrentFaze" enum to an int, it simply returns an instance of the "SaveData" struct [15] with the values of its variables.

```
public object CaptureState()
{
    // ...
    // Debug stuff
    // ...

    return new SaveData
    {
        gameFaze = (int) CurrentFaze,
        fazeTimeLeft = _fazeTimeLeft,
        growthRate = _growthRate,
        fazeDuration = _fazeDuration,
        maxFazeDuration = _maxFazeDuration
       };
}
```

Loading data with the "RestoreState" method in the "GameFazySystem" class is slightly more complicated. Since the operation of other objects in the game depends on some variables of the "GameFazySystem" class, it has events [12] that are fired at the moment of changing the phase of the game and changing the time remaining until the phase switch. Therefore, when loading data in order to maintain the correct course of the game, these events [12] must be fired by passing the appropriate variables.

```
public void RestoreState(object state)
{
    // Change state type to SaveData
    SaveData data = (SaveData) state;

    // Set variables
    CurrentFaze = (GameFaze) data.gameFaze;
    _fazeTimeLeft = data.fazeTimeLeft;
    _growthRate = data.growthRate;
    _fazeDuration = data.fazeDuration;
    _maxFazeDuration = data.maxFazeDuration;

    // Fire up events
    OnFazeChange?.Invoke(CurrentFaze);
    OnTimeLeftChange?.Invoke(_fazeTimeLeft);

    // ...
    // Debug stuff
    // ...
}
```

As mentioned above, the implementation of this interface varies drastically from class to class. For example, for the class "GridBuildingSystem" the implementation of the method "RestoreState" is much more complicated. Since it deals with the management of structures placed by the player on the map, it must first remove all the buildings currently on the map and inform the "GridXZ" class about the newly freed space which depends on the type of building being removed, before clearing the data from the save. It also stores a list of buildings on the map, but since all of them have been deleted it must be cleared. Then it can load the data, which consists of three lists storing ranked data on the types of buildings and their coordinates on the "GridXZ". Only having this data it can proceed to instantiate buildings at the appropriate locations.

```
public void RestoreState(object state)
{
    // Destroy current buildings
    foreach (IBuilding building in _buildings)
     {
            // unreserve grid space
        _grid.GetXZ(building.GetGameObject().transform.position, out int x, out int z);
        BuildingsSizes.GetBuildingSize(_building.GetBuildingType(), out int width, out int height);
        List<int[]> space = GetBuildingGridSpace(x, z, width, height);
        foreach (int[] coordinates in space)
            _grid.GetGridXYObject(coordinates[0], coordinates[1]).ClearTransform();

        // Destroy game object
        Destroy(building.GetGameObject());
    }

    // Clear buildings list
    _buildings.Clear();

    // Change state type to SaveData
    SaveData data = (SaveData) state;

    // Set variables
    List<int> bTypes = data.buildingsTypes;
    List<int> bX = data.buildingsX;
    List<int> bZ = data.buildingsZ;

    // Construct loaded buildings
    for (int i = 0; i < bTypes.Count(); i++)
    {
        switch ((BuildingTypes) bTypes[i])
        {
            case BuildingTypes.EnergyGenerator:
                _building = energyGenerator;
                Construct(bX[i], bZ[i]);
                break;

            case BuildingTypes.MetalCollector:
                _building = metalCollector;
                Construct(bX[i], bZ[i]);
                break;

            case BuildingTypes.Barracks:
                _building = barracks;
                Construct(bX[i], bZ[i]);
```

```
46                  break;
47
48              default:
49                  Debug.Log($"!!! ERROR in loading buildings: {
    bTypes[i]}/{(BuildingTypes) bTypes[i]} at {bX[i]}, {bZ[i
    ]}");
50                  break;
51          }
52      }
53
54      // ...
55      // Debug stuff
56      // ...
57  }
```

3.3 Job System

As shown in the screenshot below, without any optimization or a job system [6] for 50 enemies spawned as a wave, the user war averaging around 5.5 FPS, and generating every frame was taking around 188 ms. Of course, such a frame rate is unacceptable for comfortable gameplay, so something had to be done to massively speed up frame generation. Identifying the problem, which was pathfinding, showed multiple options of fixing it.

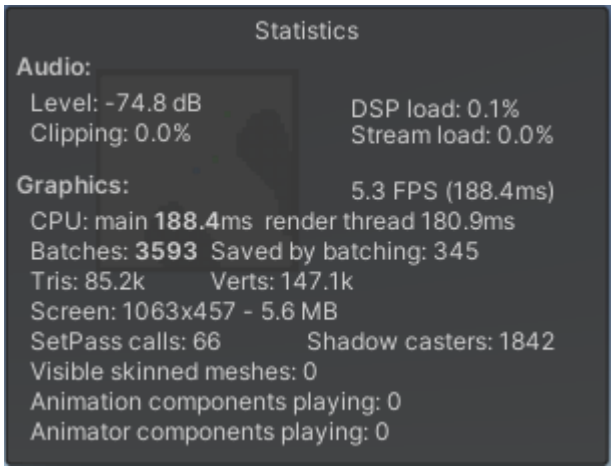

Fig. 1: Before job

In order to achieve better results, pathfinding was implemented for both enemy and ally units using Unity's Job System [6], which is Unity's way of multithreading. The task wasn't easy because not only was it a new system for the authors that they had to learn, but it was also quite complicated with its native types [10]. Native types [10] in Unity's Job System [6] are variable types that have a static address in computer memory. This way, they can be shared through many different threads without any errors. Although the task wasn't easy, the authors managed to implement it, and it was definitely worth the time they had to spend on it.

After a very complicated implementation of both Unity's Job System [6] and the Burst compiler [1], which is covered below, along with some classic code optimization, pathfinding algorithm execution went from 160 ms to less than 6 ms for all threads and to 0.2 ms for a single task. By speeding up algorithm execution, FPS increased from 5.5 to over 60.

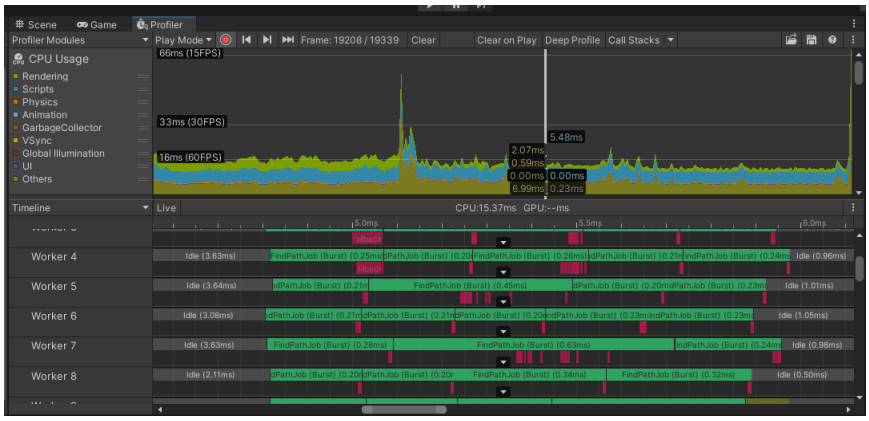

Fig. 2: After job

3.4 Pathfinding

The Pathfinding system consists of four modules attached to units, an interface, two main classes and two support classes.

Modules attached to units:

- UnitPathData() - module that stores the path for a given unit
- PathfidingMove() - module responsible for moving a unit based on data from UnitPathData() module
- PathfidingEnemyMove() - module attached to enemy units, which allows to determine the range at which enemies continue to follow the target (aggro range)
- ShowPath() - module that visualizes the path taken by the unit needed for development

Interface:

- IUnit - a simple interface that provides the ability to set the calculated path to the unit

Support classes:

- PathNode - an abstract class storing variables relevant to Pathfinding from which the GridObject() class inherits
- PathNodeStruct - storing analogous data PathNode class needed for NativeLists [10]

Main classes:

- UnitsManager() - class that manages and passes unit path data
- FindPathJob() - a class using the A* algorithm [13] that is implemented in multithreading through the Unity Job System [6] together with the BurstCompiler [1] system

UnitsManager

The main component of the Pathfinding system is the UnitsManager() class. After the grid is initialized and the map is generated, and each time a change is made to the map, it creates and stores a copy of the grid in the form of a NativeArray [10] which is compatible with the Unity Job System [6]. Units that have the PathfindingMove() modulus send requests to this class to determine the path to a specific target represented as coordinates on the grid, after which it creates a set of variables needed by the FindPathJob() class. These variables are stored in the form of dictionaries where the key is the unit to which the associated native list [10] belongs.

```
// ...

private static Dictionary<IUnit, JobHandle> _units;
private static Dictionary<IUnit, NativeList<PathNodeStruct>>
    _openLists;
private static Dictionary<IUnit, NativeList<PathNodeStruct>>
    _closedLists;
private static Dictionary<IUnit, NativeArray<PathNodeStruct>>
     _neighbors;
private static Dictionary<IUnit, NativeList<PathNodeStruct>>
    _pathsReversed;
private static Dictionary<IUnit, NativeList<PathNodeStruct>>
    _paths;
private static Dictionary<IUnit, NativeArray<PathNodeStruct>>
     _nativeGrids;

private static GridXZ<GridObject> _gridXZ;
private static NativeArray<PathNodeStruct> _nativeGrid;
private static int _gridWidth;
private static int _gridHeight;

// ...
```

Listing 1: UnitsManager variables

The prepared variables are passed to the FindPathJob() class, which, on a separate processor thread, starts calculating the path for a given unit by returning the JobHandle variable stored by the UnitsManager() class to control the entire process. Then in the LateUpdate() method and so at the end of the frame generation, the UnitsManager class receives all calculated paths and passes them to the corresponding unit through the IUnit interface and then releases temporary NativeLists [10] to avoid memory leaks.

FindPathJob

The second no less important main element is the FindPathJob() class. It is in this class that the actual calculation of the path for an unit takes place using a slightly modified A* algorithm [13]. Thanks to the fact that the class is implemented to work with Unity Job System [6], all calculations are carried out on a separate processor thread, which significantly reduces the frame generation time. In addition, the entire class uses Unity BurstCompiler [1] and optimized Unity math from the Mathf library [7] to further reduce calculation time.

3.5 Perlin Noise & Procedural Generation

A system of procedural terrain generation [17] was implemented in this game, which, based on the parameters of frequency, density and grain, generates the world on which the player will play. The "Mathf" library [7] available in Unity, was used. It contains many useful mathematics methods, including the "Mathf.PerlinNoise(float,float)" method which returns the value of Perlin noise [14] for the given coordinates. As an argument it receives the product of the coordinate located on the "GridSystem" and the frequency and the result is summed with the seed given by the player. The value thus obtained is then compared with the density factor and if it exceeds it and the given area is not marked as a safe zone (the zone of the player's initial base) then the given cell in the grid is marked as a mountain if there is nothing on it yet.

```
// ...

for (int x = 0; x < width; x++)
{
    for (int z = 0; z < height; z++)
    {
        if (x >= safeZoneStartX && x <= safeZoneEndX && z >= safeZoneStartZ && z <= safeZoneEndZ)
        {
            res[x, z] = 0f;

            if (DebugSettings.SHOW_SAFE_ZONE)
            {
                Vector3 position = _gridXZ.GetWorldPosition(x, z);
                position.y += 0.25f;
                position.x += 0.5f;
                position.z += 0.5f;

                Instantiate(
                    indicator,
                    position,
                    Quaternion.identity
                    );
            }
        }
        else
        {
            res[x, z] = Mathf.PerlinNoise(x*frequency+seed, z*frequency+seed);
```

```
28          }
29
30          if (DebugSettings.WRITE_PERLIN_TO_CONSOLE)
31          {
32              Debug.Log("[ " + x + ", " + z + "] " + res[x, z])
    ;
33          }
34      }
35  }
36
37  // ...
```

Listing 2: PerlinNoise usage

These designations are stored as a two-dimensional array of floating-point numbers which is then passed to the appropriate methods involved in instantiating the corresponding structures. After completing the instantiation of terrain structures into the game, all mountain objects are divided into two categories, the inner mountain and the generated one. The first one is characterized by the fact that it is surrounded on all sides by other mountains. Generated mountain, as it is not surrounded on all sides, will be partially visible to the player during the game, so its surface must be adjusted accordingly. This is handled by the "MountainGenerator" class, which is a component of each generated mountain and its methods are called by the "WorldGenerationSystem" in the appropriate order.

3.5.1 Mountain Generation The generation of mountain surfaces is handled by the "MountainGenerator" class whose methods are called by the "WorldGenerationSystem" in the appropriate order. This process is divided into three stages:

1. **Preparing the generated mountains to fold their surfaces** This step consists of four actions that prepare all the vertices of the mesh [8] for further transformations in subsequent steps. First, mountains that have no neighbors in any of the 4 main directions (up, down, right, left) are removed. By way of experimentation, the authors found this to be the best solution because otherwise the terrain would be similar to strange-looking cones sticking out of the ground. Then the vertices that are on the outer edges are moved to the height of either the ground or the inner mountain (an inner mountain is one that is adjacent to other mountains in all 8 directions) depending on what they are adjacent to. The next step is to detect and raise to the height of the inner mountains vertices located in the corners which are adjacent to three mountains of any kind, for example, if a vertex located in the lower left corner of W(0, y, 0) is considered, then in order for it to be raised the mountain to which it belongs must be adjacent to another mountain at the bottom, left and diagonally down and left. The last step in this stage is to detect edges which two ends, i.e. the corners of the generated mountain's mesh, are raised and raise the entire edge.

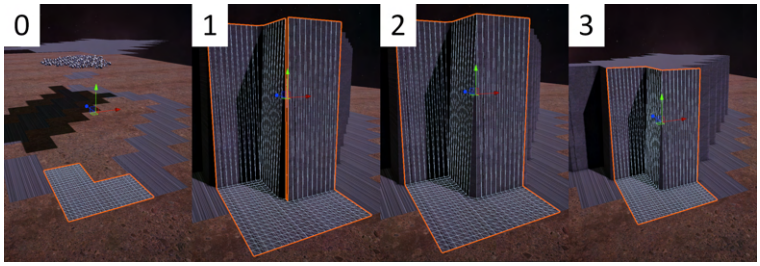

Fig. 3: Prepare for folding surface

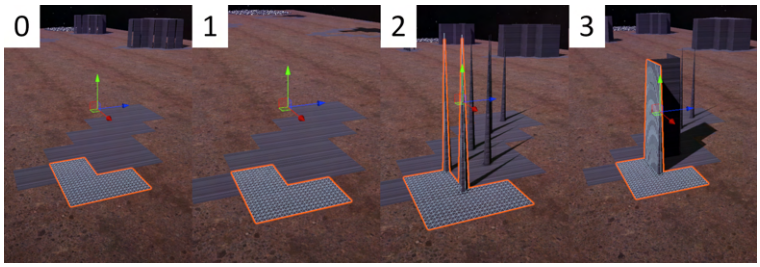

Fig. 4: Prepare for folding surface

2. **Adjusting the mountains that are connectors and their neighbors (mountains that have only two neighbors top-bottom or right-left)**
   This stage is quite simple, but because the steps contained in the previous stage must be performed for all instances of generated mountains for this one to run correctly it has been separated as a distinct stage. Two methods are performed here, the first of which detects and adjusts "connector mountains". These are mountains that have only two neighboring mountains in all eight directions and these neighbors are on opposite sides in two of the four main directions (up, down, right, left). If the method detects such a mountain, it raises one or two columns/rows of vertices located in the middle depending on whether the neighboring mountains are located on top and bottom or right and left, and whether the number of vertices on the axis is even or odd. Then a method is executed that detects all mountains that have not been modified up to this point and then removes them. Such mountains, due to their unusual position on the map, would look strange and unnatural if they were modified in the end.

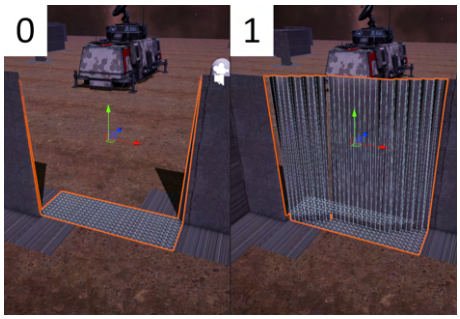

Fig. 5: Adjust connector mountains

3. **Surface folding using Perlin noise [14] from the "Mathf" library [7]**

   The final step in the algorithm for generating the side surface of mountains is surface folding. The first step adjusts the internal mesh [8] vertices. In the previous steps, the script detected all the vertices of each "generated mountain" that should be raised to the maximum (fixed) height, so now the straight lines between each raised vertex and all the vertices on the edges can be determined. The coefficients of these straight lines are then used to determine which vertices should be modified and to what height they should be raised. In order to avoid the computationally complex Cartesian distance calculation, only two vertices adjacent to each other on the argument axis should be compared. In addition, the new height of a vertex is assigned only if it is greater than the existing height. This allows the algorithm to avoid a situation where the just calculated straight line removes the changes made by the previous one.

```
// ...

// modify vertices if new height is greater then old one
for (int modX = startX; modX < endX; modX++)
{
    // calculate straight variables
    float t = (modX - startX) / vector[0];
    float height = startY + t * vector[1];

    // choose z to modify
    float zRes = startZ + t * vector[2];
```

```
12      int modZ = Mathf.Abs((int)zRes - zRes) < Mathf.Abs((
        int)zRes + 1 - zRes)
13          ? (int)zRes
14          : (int)zRes + 1;
15
16      // modify y if new one is greater
17      if (_vertices[GetIndex(modX, modZ)].y < height)
18          _vertices[GetIndex(modX, modZ)].y = height;
19  }
20
21  // ...
```

Listing 3: Part of AdjustInnerVertices() function

Now that the general shape of each "generated mountain" is properly generated, the Perlin noise [14] function can be applies to all vertices. Depending on the position of the vertex, not all of its variables can be modified. For example, if the vertex is on the right edge its x-coordinate cannot be modified, because otherwise there could be a gap between the meshes of neighboring mountains through which the player could look under the textures. Of course, this is an undesirable situation so the script takes into account the position of the vertex when applying noise. For all coordinates for which such changes can be made for a given vertex, the arguments passed to Perlin's noise [14] function are calculated. To avoid repetition between closely located mountains, but to keep the generation procedural [17], the following algorithm for calculating the arguments of the function is used.

```
1  // ...
2
3  _vertices[GetIndex(x, z)].x +=
4      -1 * baseOffsetX
5      + Mathf.PerlinNoise(CalculateX(x) + 10,  CalculateZ(z)
       + 10) * 2 * baseOffsetX;
6
7  // ...
8
9  float CalculateX(int x) => x * _gridPosX * 0.1f + _seed;
10 float CalculateZ(int z) => z * _gridPosZ * 0.1f + _seed;
11
12 // ...
```

Listing 4: Part of ApplyPerlinNoise() function

Offset is a constant not exceeding 0.5f defined for each of the three axes. If the offset value were greater than 0.5f, a situation could arise in which an earlier vertex would be beyond the next which would cause graphical defects. The values calculated in this way are then summed up with the current ones, with the exception of the y-coordinate, which is additionally checked to make sure it does not exceed the maximum height of the mountain.

The final stage of mountain generation is to close the small holes between the generated mountains that were created when Perlin noise [14] was applied. The algorithm compares the coordinates of the vertices on the left and bottom edges of the mesh [8] to the coordinates of the vertices of the right or top edge to the left or bottom edge, respectively, of a neighboring generated mountain if one exists. These edges are compared because the WorldGenerationSystem() class calls the individual generation stages for each generated mountain in sequence starting from the left bottom edge of the Grid so selecting those edges ensures that the neighboring mountain has already completed all generation stages.

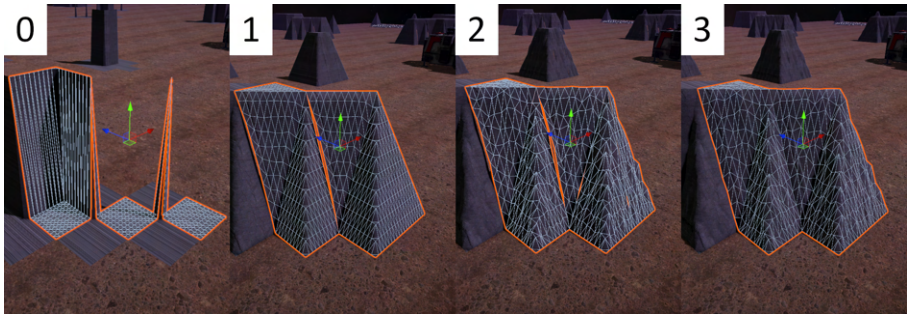

Fig. 6: Fold surface

3.6 Singleton

Singleton design pattern [4] was used for Game Event system. Singleton is a design pattern used to easily access vital data all across the code, while ensuring only one instance of Event System exists across the game.

3.7 Your proposed solution (e.g. A*, Design pattern)

The A* (A-Star) algorithm [13] was proposed for pathfinding in the game. The A* algorithm [13] is widely used in game development for pathfinding because it provides an optimal path to the target with minimal computation. The algorithm uses heuristics to prioritize nodes that are closer to the target, making it efficient in finding the shortest path.

3.8 Your proposed testing protocol

To ensure the quality and stability of the game, it was proposed to use both manual and automated testing methods. Manual testing was conducted throughout the development process to identify and fix issues in the game. Additionally, Unity's testing framework was used to create and run automated tests that checked the game's functionality and performance. Profiling tools, such as Unity Profiler and Visual Studio Profiler, were used to identify and optimize performance bottlenecks in the game.

4 Results

4.1 Feature/System 1 (Your own implementation, e.g. Enemies/Combat System)

After games starts player is being spawned on proceduraly generated [17] map and random mission is being assigned to him. He has a few minutes to build a simple base with his starting units and after that combat phase begins. During the combat phase EnemiesSystem starts to spawn enemy troops who begin to attack the player and his base. If player manages to defeat them he gets the rewards and another cycle begins. With each consecutive fighting phase the number of enemies is being increased so that player doesn't have infinite time for accomplishing the mission.

4.2 RTS System

While in RTS mode, the camera switches to a bird's-eye view, and main character controls are disabled. Instead, the player is able to control units and buildings, construct new ones, and cast abilities with HUD buttons. In addition, the player has access to battle reports, resource counters and objectives.

4.2.1 Buildings There are three types of buildings to construct: barracks, energy generators, and metal collectors. Energy generators passively generate energy resources, while metal collectors automatically send drone units to the nearest metal reserve. Both of them can have their efficiency upgraded. Barracks allow for training and queueing up to five units at a time. They also have the option to cancel the queue.

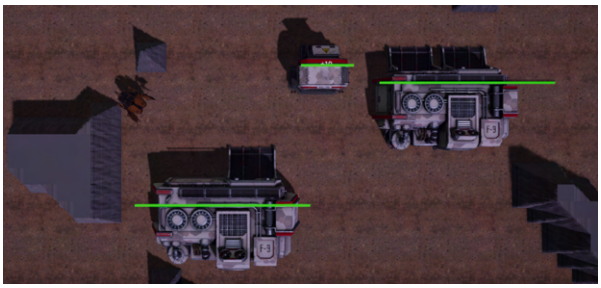

Fig. 7: Buildings

4.2.2 Resources Typically in RTS games, resources are constraints put on the player to create a need for strategic decision-making. There are four main constraints put on the player. The first two are metal and energy. They are gained by building metal collectors and energy generators, respectively. Both are needed to perform any RTS tasks, such as training units. Another constraint is the maximum number of units. To improve both balance and performance, a strict unit limit is enforced. Finally, there is a time constraint. Constructing buildings is disabled in the fighting phase; therefore, the player must take that into consideration. Resources are managed by the Game Event System and are generated within specific building scripts.

```
1  private IEnumerator GenerateEnergy()
2      {
3          while (true)
4          {
5              yield return new WaitForSeconds(_speed);
6              if (_work)
7              {
8                  ResourceSystem.AddEnergy(_efficiency);
9                  _feedback.ResetPos();
10             }
11         }
12     }
```

Listing 5: Energy Generator Coroutine

4.2.3 Building Construction Process The building construction process starts when the player selects the building icon in the HUD and clicks on a suitable place on the game space. Then resources are spent, and a placeholder is created. The building health bar is also visible filling up, informing on the progress.

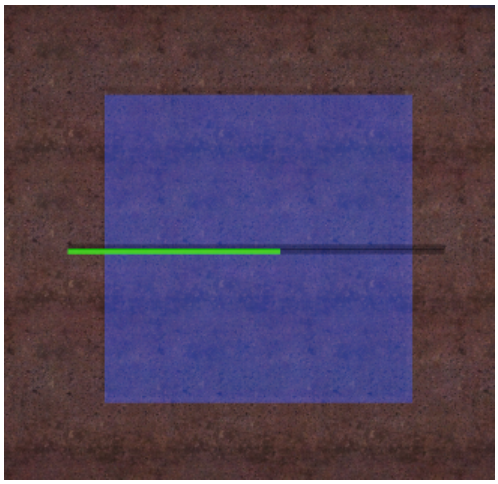

Fig. 8: Building Construction Process

After the bar is filled, an object falls from the sky, making the building visible and usable, while the object and the placeholder are destroyed.

```
private void OnCollisionEnter(Collision collision)
{
    if(!_contacted)
    {
        _contacted = true;
        OnContact?.Invoke();
        Destroy(frame.gameObject);
        building.SetActive(true);
        Destroy(gameObject);
    }
}
```

Listing 6: OnCollision function enabling building

4.2.4 Units Units can be trained in barracks. Barracks hold a queue of up to five units. Before enqueuing a unit, they check if all conditions are met. In that case, resources are taken, and units go into a queue. Only one unit can be trained at a time, and each unit has its own separate train time progress, which is displayed directly above the building's health bar.

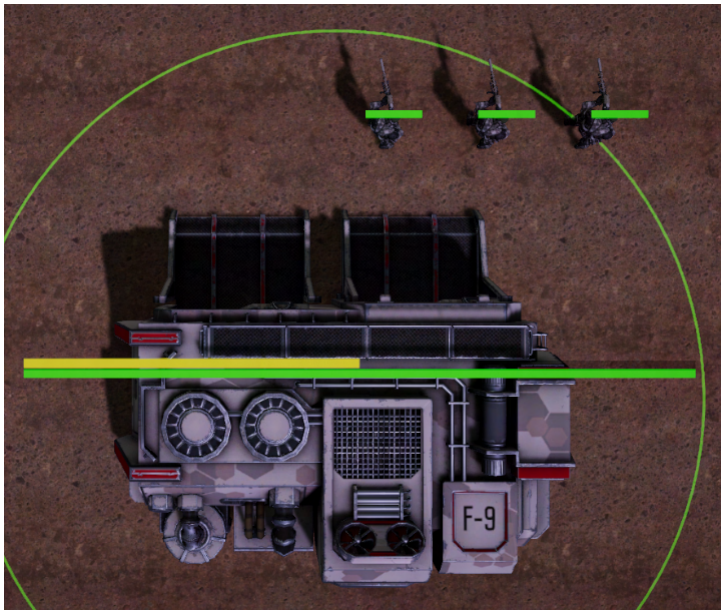

Fig. 9: Unit Training

```
1     public void StartTraining(int i)
2     {
3         if (ResourceSystem.GetMetal() < _unitMetalCost[i]||
    ResourceSystem.GetEnergy() < _unitEnergyCost[i])
4         {
5             GameEventSystem.SendMessage(MessageFlag.
    LackResources);
6             return;
7         }
8
9         if (_unitsToTrain.Count >= 5)
10        {
11            GameEventSystem.SendMessage("Queue Full");
12            return;
13        }
```

```
14          if (GameEventSystem.Instance.unitCount >=
    GameEventSystem.Instance.unitLimit )
15          {
16              GameEventSystem.SendMessage("Unit Limit Reached")
    ;
17              return;
18          }
19          ResourceSystem.ConsumeEnergy(_unitEnergyCost[i]);
20          ResourceSystem.ConsumeMetal(_unitMetalCost[i]);
21          GameEventSystem.Instance.unitCount++;
22          GameEventSystem.TrainUnit(1);
23
24          _unitsToTrain.Enqueue(i);
25      }
```

Listing 7: Enqueuing script

```
1  IEnumerator TrainUnits()
2      {
3
4          while (true)
5          {
6              bool broken=false;
7              if (_unitsToTrain.Count > 0)
8              {
9                  {
10
11                 }
12                 for (int i = 0; i < 10; i++)
13                 {
14                     _trainProgress.fillAmount = (float)i /
    10;
15                     yield return new WaitForSeconds(
    _unitTimers[_unitsToTrain.Peek()] / 10f);
16                     if (_scheduleForClear)
17                     {
18                         _scheduleForClear = false;
19                         ClearQueue();
20                         _trainProgress.fillAmount = 0;
21                         broken = true;
22                         break;
23                     }
24                 }
25
26                 if (broken)
27                 {
28                     continue;
29                 }
30                 Instantiate(_units[_unitsToTrain.Dequeue()],
```

```
31                      spawnPoint.position + (Vector3.right * (2
    * _unitsToTrain.Count)),
32                      Quaternion.identity,
33                      _alliesHolder
34                  );
35                  _trainProgress.fillAmount =0;
36              }
37              else
38              {
39                  yield return null;
40              }
41          }
42      }
```

Listing 8: training coroutine

4.3 Models

The game models were selected based on their compatibility with the game's theme. All chosen models were rigged, enabling the use of humanoid animations from Mixamo.com. such as idle, shoot, walk, run, death, and specialized actions like shooting while kicking. One of the most challenging aspects was learning Inverse Kinematic[**?**]. To ensure the weapon aligned correctly with different animations, Unity's Animation Rigging package was extensively used. Particularly, the Two Bone IK constraint[5] was employed to accurately position the hands for each animation. Scripts were also used to adjust the body and head offset as needed for specific parts of the animation. When it came to animating the guns, a similar approach was taken. However, in some instances, manual animation within Unity was required for more detailed actions, such as dropping the magazine. This combination of automated and manual techniques provided a better appearance. Additionally, the player animator includes four distinct layers: base layer, upper body layer, head layer, and rifle layer, each with a unique mask. On the base layer, a simple blend tree was implemented for movement, providing a fluid transition between different states of motion. For gun animations, the rifle layer is primarily used. Through scripting, each animation's layer weight is modified to enhance the visual appeal. Furthermore, animation keyframing is employed in certain cases for the Two Bone IK constraint target, adding further precision and detail to the weapon handling animations.

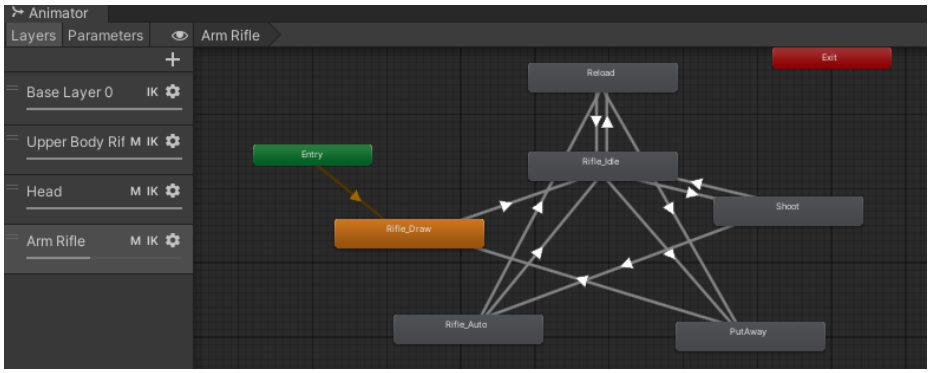

Fig. 10: Player animator

4.3.1 Ranged enemy The Lu model from the Adam Character Pack made by Unity Technologies was used as the ranged ally character.

Fig. 11: Enemy model

4.3.2 Player Model The Guard model from the Adam Character Pack made by Unity Technologies was used as the main character.

Fig. 12: Player Model

4.3.3 Weapon Models For the weapons in the game, the 'Sci-fi Weapons Arsenal' by Chiako3D (also known as CGi) from the Unity Asset Store was utilized.

Fig. 13: Weapon models

4.3.4 Ranged ally The Sci-Fi Character 08 Renegade model created by Maksim Bugrimov from the Unity Asset Store was used as the ranged ally character.

Fig. 14: Ally model

4.4 Camera System

The game features a dynamic camera system that adapts to the current game mode FPS or hud using four scripts: MovePlayerCamera , CameraController , WeaponCamera ,RtsMovmentCamera.

This setup involves three distinct cameras:

Main Camera: This camera is designed to adapt to gameplay styles seamlessly switching between first person shooter (FPS) and real time strategy (RTS) perspectives.

Weapon Camera: This camera is active only during first-person gameplay, ensuring that the weapons look clear and visually striking. Its main role is to enhance the visual quality of the firearms, contributing to a more immersive FPS experience.

Minimap Camera: This camera is configured to only render the minimap layer. It gives players a top-down view of the area around them.

MovePlayerCamera: This script switches the camera position between the FPS and RTS modes. It uses two Transform objects, cameraPositionFPS and cameraPositionRts, to define the camera's location in each mode. When the game mode changes, the camera's position is updated to the respective mode's position.

CameraController: This script manages the camera's rotation, allowing the player to look around in FPS mode. It captures mouse movement and applies it to the camera's rotation, with sensitivity settings for X and Y axes. In RTS mode, it sets a fixed camera rotation. Additionally, in the Late Update method, the script aligns the spine's rotation with the camera in FPS mode, ensuring the player model's upper body orientation matches the camera's direction.

```
1        private void UpdateRotation()
2        {
3            if (GameEventSystem.GameMode == GameMode.FPS_MODE
    )
4            {
5                // get mouse input
6                float mouseX = Input.GetAxisRaw(
    PlayerConstants.MOUSEX) * Time.deltaTime * sensX;
7                float mouseY = Input.GetAxisRaw(
    PlayerConstants.MOUSEY) * Time.deltaTime * sensY;
8
9                _yRotation += mouseX;
10               _xRotation -= mouseY;
11
12               // lock camera rotation up to 90
13               _xRotation = Mathf.Clamp(_xRotation, -90f, 90
    f);
14
15                _xRotation = Mathf.Clamp(_xRotation, -90f,
    60f);
16               orientationFPS.rotation = Quaternion.Euler(0,
     _yRotation, 0);
17               cameraPos.localRotation = Quaternion.Euler(
    _xRotation, 0, 0);;
18           }
19           else
20           {
21               // rotation of camera in RTS mode
22               transform.rotation = Quaternion.Euler(90f, 0,
     0);
23           }
24       }
```

Listing 9: Update Rotation function

CameraPositionController(RTS Camera): This script controls the RTS camera, enabling strategic overhead views. It allows panning and zooming and dynamic clamping based on the camera's height. As the camera zooms in or out, the script dynamically adjusts the panning boundaries using a linear regression method.

The boundaries were calculated by analyzing data points representing different heights and their corresponding maximum allowable pan distances. This linear regression provides a formula that adjusts the pan limits as the camera height changes.

```
//Panning the camera based on player input and mouse position
if (Input.GetKey(PlayerConstants.MOVEUP) || Input.
    mousePosition.y >= Screen.height - panBorderThickness)
{
    pos.z += panSpeed * Time.deltaTime;
}
// Similar logic applies for moving down, left, and right.

float scroll = Input.GetAxis(PlayerConstants.SCROLL);
pos.y -= scroll * scrollSpeed * 100f * Time.deltaTime;

// dynamic pan limits based on height
float dynamicPanLimitX = -0.95f * pos.y + 47.17f;
float dynamicPanLimitZ = -0.55f * pos.y + 48.83f;

pos.x = Mathf.Clamp(pos.x, -dynamicPanLimitX,
    dynamicPanLimitX);
pos.y = Mathf.Clamp(pos.y, minY, maxY);
pos.z = Mathf.Clamp(pos.z, -dynamicPanLimitZ,
    dynamicPanLimitZ);

transform.position = pos;
```

WeaponCamera: This script is specifically for rendering weapons in the FPS mode. It ensures that weapons are visually detailed and prominently displayed when in the first-person view. Another reason for using a separate weapon camera is to prevent weapons from clipping through objects, which can break the realistic look of the game.

4.5 FPS System

4.5.1 Movement Player movement in the game is handled with a basic movement script. The MovePlayer() method in the PlayerMovement script captures player inputs for horizontal and vertical movements, calculates the direction based on the player's orientation, and applies force for movement. This ensures smooth transitions between walking, running, and standing still. The Jump() method allows the player to jump, adding a vertical dimension to the movement. This action is only possible when the player is grounded, which adds a layer of realism and strategic movement to the gameplay.

4.5.2 Weapons The Shoot() function is responsible for shooting mechanism for the equipped gun. When the player attempts to shoot the function first checks if the gun is reloading or out of ammo. If the gun is ready to fire then performs a raycast starting from the camera perspective. This raycast helps determine where the player is aiming and identifies where the bullet will hit upon colliding with objects in the game world.

Once a target position is determined the function calculates how the bullet will travel. This calculation includes an element of spread, which adds a realistic variance to the bullet's path.The script then instantiates a bullet projectile at the gun's barrel and applies force to propel it towards its target.

Aside, from controlling the behavior of bullets this function also enhances players shooting experience through auditory feedback. It triggers a muzzle flash effect when firing.The method also controls how the gun fires, by keeping track of the number of shots fired.

Lastly when the gun runs out of ammo during this process the Shoot() function triggers a reloading sequence. This sequence not replenishes the guns ammunition. Also includes a visual animation of the reloading action enhancing the authenticity of the game and prompting players to tactfully manage their ammo usage.

```
1
2 public void Shoot(){
3
4 // Check if the gun is currently reloading
5 // Check if there is available ammo to fire
6 // Perform raycasting to determine the target point (ignoring
      unnecessary layers)
7     // Determine the target point
8     Vector3 directionWithoutSpread = targetPoint -
    attackPoint.position;
9
10    // Apply bullet spread
11    float x = Random.Range(-spread, spread);
12    float y = Random.Range(-spread, spread);
13    float z = Random.Range(-spread, spread);
14
15    // Calculate the direction with spread
16    Vector3 directionWithSpread = directionWithoutSpread +
    new Vector3(x, y, z);
17
18    // Create and shoot the bullet
19    GameObject currentBullet = Instantiate(bullet,
    attackPoint.position, Quaternion.identity);
20    currentBullet.transform.forward = directionWithSpread.
    normalized;
21
22    // Apply forces to simulate shooting
23    currentBullet.GetComponent<Rigidbody>()
24        .AddForce(directionWithSpread.normalized * shootForce
    , ForceMode.Impulse);
25    currentBullet.GetComponent<Rigidbody>().AddForce(fpsCam.
    transform.up * upwardForce, ForceMode.Impulse);
26
27    // Activate bullet behavior if applicable
28    if (currentBullet.GetComponent<CustomProjectiles>())
29        currentBullet.GetComponent<CustomProjectiles>().
    activated = true;
30
31    // Display the muzzle flash effect
32    Instantiate(muzzleFlash, attackPoint.position, Quaternion
    .identity);
33
34    // Update the fire counter to manage the firing rate
35
36    // Check if the player is out of ammo and initiate the
    reload sequence
37 }
```

Listing 10: Shoot function

In addition to this primary shooting function, several other methods contribute to the comprehensive weapons system:

SwitchGun(): This method allows player to switch between their weapons. It deactivates the currently active gun and activates the selected one from the allGuns list. It ensures a transition during combat by taking care of all updates to the weapons. This includes stopping any reload animations and updating the ammunition display, on the UI.

```
1
2  public void SwitchGun()
3  {
4      if (activeGun != null && activeGun.isReloading)
5      {
6          activeGun.StopCoroutine(activeGun.Reload());
7          activeGun.isReloading = false;
8      }
9
10     if (currentGun >= 0 && currentGun < allGuns.Count)
11     {
12         activeGun = allGuns[currentGun];
13         foreach (Gun gun in allGuns)
14         {
15             gun.gameObject.SetActive(false);
16         }
17         activeGun.gameObject.SetActive(true);
18     }
19     // Additional code for updating UI...
20 }
```

Reload(): This method manages the reloading of guns when ammo depletes. It includes animations and sound effects for realism, temporarily disabling the gun's functionality until the reload is complete.

```
1
2  public IEnumerator Reload()
3  {
4      isReloading = true;
5      // Play reload animation and sound
6      yield return new WaitForSeconds(reloadTime);
7      currentAmmo = maxAmmo;
8      isReloading = false;
9      // Update UI elements...
10 }
```

AddGun(): This method is used to introduce new weapons into the game. It moves guns from the unlockableGuns list to the allGuns list once players meet certain criteria, expanding their combat options.

```
1
2  public void AddGun(string gunName)
3  {
4      Gun gunToUnlock = unlockableGuns.Find(gun => gun.gunName
       == gunName);
5      if (gunToUnlock != null)
6      {
7          allGuns.Add(gunToUnlock);
8          unlockableGuns.Remove(gunToUnlock);
9          // Possibly trigger some notification or UI update...
10     }
11     // Additional logic for switching to the new gun if
       needed...
12 }
```

4.5.3 Custom projectile The CustomProjectiles.cs script is designed to control different types of behaviors for projectiles in the game. It utilizes Unitys built in features to ensure that the projectiles behave realistically. This includes how they bounce, whether they fall with gravity, and how they interact with other things like enemies and walls. To determine how a projectile bounces upon collision or slides on surfaces the script relies on Unitys physics materials. It also incorporates timers to introduce delays or trigger actions based on conditions like waiting before a projectile explodes. By utilizing Unitys transform component the script enables movement and rotation of the projectiles allowing them to soar through the air or even change size.

Explode() method is called when a projectile is set to explode, dealing damage within a specified range. This involves checking for nearby enemies using Physics.OverlapSphere and applying damage and forces accordingly. Additionally instantiate explosion effects and ensures that each projectile only explodes once. Upon explosion or at certain intervals, the projectile can spawn other objects. This is used for creating complex effects like a bomb that releases shrapnel.

Grow(), StickToObject(), DropDown(), Pearl() These methods are allows for special behaviors such as growing in size or damage

over time, sticking to surfaces upon impact, or having a delayed drop-down effect. Each of these behaviors is managed using conditional checks and timers within the Update method and is applied using Unity's physics and transform systems. Some of the more unique features include the ability for the projectile to teleport objects or generate smoke effects upon certain triggers.

```
1
2  public void Explode() {
3      if (alreadyExploded) return;
4      alreadyExploded = true;
5      if (explosionEffect != null) {
6          Instantiate(explosionEffect, transform.position,
       Quaternion.identity);
7      }
8
9      Collider[] enemies = Physics.OverlapSphere(transform.
       position, explosionRange, whatIsEnemies);
10         for (int i = 0; i < enemies.Length; i++)
11         {
12             //Damage enemies
13             if (enemies[i].GetComponentInParent<
       EnemyHealthController>())
14                 enemies[i].GetComponentInParent<
       EnemyHealthController>().TakeDamage(explosionDamage);
15
16             //Add explosion force to enemies
17             if (enemies[i].GetComponent<Rigidbody>())
18                 enemies[i].GetComponent<Rigidbody>()
19                 .AddExplosionForce(explosionForce, transform.
       position, explosionRange, 2f);
20         }
21     //Pearl(Teleport) to position
22     if (_objectToTp != null && tpOnExplosion) Pearl(transform
       .position);
23
24     //Spawn second bullets and add forces if second bullet
       attatched
25     for (int i = 0; i < sbAmount; i++)
26     {
27     float x = Random.Range(-1, 1f);
28     float y = Random.Range(-1, 1f);
29     float z = Random.Range(-1, 1f);
30     Vector3 random = new Vector3(x, y, z);
31     Rigidbody sbRigidbody = Instantiate(secondBullet,
       transform.position + random, Quaternion.identity)
32     .GetComponent<Rigidbody>();
33
```

```
34      if (sbForwardForce > 0)
35          sbRigidbody.AddForce(transform.forward *
    sbForwardForce, ForceMode.Impulse);
36      if (sbUpwardForce > 0)
37          sbRigidbody.AddForce(sbRigidbody.transform.up *
    sbUpwardForce, ForceMode.Impulse);
38      if (sbRandomForce > 0)
39          sbRigidbody.AddForce(random * sbRandomForce,
    ForceMode.Impulse);
40
41      sbAmount--;
42  }
```

Listing 11: Explode Function

```
1
2       private void Pearl(Vector3 teleportPosition)
3       {
4           if (switchPlaces) transform.position = _objectToTp.
    position;
5           if (stickToObjects)
6           {
7               _stickingPos = _objectToTp.position;
8               _sticking = true;
9
10              if (_turnCheckForEnemyOnAgain) Invoke("
    TurnCheckForEnemiesOnAgain", 0.1f);
11          }
12
13          _objectToTp.position = teleportPosition;
14      }
```

Listing 12: Pearl Function

4.5.4 Enemies Each enemy includes two colliders to handle their interaction with players or allies effectively. They managed by distinct scripts named EnemieAgroCollider.cs and EnemieAttackCollider.cs respectively. The larger Aggro Collider is focused on early detection and engagement initiation. It triggers when a potential target, such as a player or an ally, enters its range, signaling the enemy to focus on or move towards the detected entity. Additionally, there's a smaller Attack Collider nested within the Aggro Collider. This collider is specifically responsible for determining when attacks should start, continue, or stop using OnTriggerEnter(), OnTriggerExit(), and OnTriggerStay() respectively.

Enemies get spawned next to the border of the map. The amount of enemies spawning is predetermined byt the difficulty of the game at that moment. The spawning system finds an empty spot next to the border. After finding a spot, it checks if it's already empty. Then it checks if an enemy spawned there would be able to find a path to the character - for example, if there is a small area surrounded by rocks, the enemy wouldn't be able to pathfind the the character from that place. In that case, the spawner stops. If the enemy would be able to find a path, it spawns an enemy there. After that, if it tries to spawn more than one enemy in this area, the system checks which places next to the main one are free - if there's nothing there, if an enemy spawned there can pathfind from there and so on. It repeats until it can't find a good spot or until it spawns all the enemies it tries to.

```
1
2 private void OnTriggerEnter(Collider other) {
3     if (other.tag.Equals(TagConstants.ALLY) || other.tag.
    Equals(TagConstants.PLAYER)) {
4         _papa.SetChaseTarget(other.gameObject);
5     }
6 }
```

Listing 13: Aggro Collider

```
1
2 private void OnTriggerEnter(Collider other) {
3     if (other.tag.Equals(TagConstants.ALLY) || other.tag.
    Equals(TagConstants.PLAYER)) {
4         if (!_isAttacking) {
5             _papa.Attack(other.gameObject);
6         }
7         _inRange.Add(other);
8         _isAttacking = true;
9     }
10     else if (other.tag.Equals(TagConstants.MAIN_BASE)) {
11         _papa.Stop();
12     }
13 }
14
15 private void OnTriggerStay(Collider other) {
16     if (other.CompareTag(TagConstants.ALLY) || other.
    CompareTag(TagConstants.PLAYER)) {
17         _papa.Attack(other.gameObject);
18     }
19 }
```

```
20
21 private void OnTriggerExit(Collider other) {
22     if (other.tag.Equals(TagConstants.ALLY) || other.tag.
       Equals(TagConstants.PLAYER)) {
23         _inRange.Remove(other);
24
25         if (_inRange.Count != 0) {
26             _isAttacking = true;
27             _papa.Attack(_inRange[0].gameObject);
28         } else {
29             _isAttacking = false;
30             _papa.StopAttack();
31         }
32     }
33 }
```

Listing 14: Attack Collider

The main 'Enemy.cs' script utilizes the information from the Aggro and Attack colliders to make informed decisions about the enemy's actions. Upon detecting a player or ally through the Aggro Collider, the script initiates movement towards the target, transitioning to a chase state. Once the target enters the Attack Collider's range, the script triggers the combat mechanisms, handling details like when and what type of attack to perform using methods such Attack(), StopAttack(), and decision making processes that rely on OnTrigger events [12]. The script continuously updates the enemy's state based on the current situation, whether chasing a target, attacking, or repositioning due to changes in the target's status or location. It includes functions for movement using pathfinding and speed adjustments.

```
1
2 public void Attack(GameObject target) {
3     IsAttacking = true;
4     _chase = false;
5     _pathfindingEnemyMove.SetSpeed(0f);
6     // Trigger animations for attack
7
8     enemyShooting.enabled = true; // Enable shooting
      component
9     enemyShooting.Shoot();
10 }
11
12 public void StopAttack() {
13     IsAttacking = false;
14     _chase = true;
```

```
15      // Reset animations from attack to movement
16      enemyShooting.enabled = false; // Disable shooting
        component
17  }
18
19  public void SetChaseTarget(GameObject target) {
20      _chaseTarget = target;
21      // Reset animations from attack to movement
22      _pathfindingEnemyMove.SetSpeed(EnemiesSettings.SPEED);
23  }
24
25  public void ResetChaseTarget() {
26      if (_chaseTarget != _mainBase) {
27          _chaseTarget = _mainBase; // Reassign the chase
        target to the main base
28      }
29  }
```

The games event system plays important role, in altering the behavior of enemies especially when an ally they were targeting dies or is no longer in the game. The HandleAllyDeath() method is invoked, which prompts the enemy to reevaluate its target. If other potential targets are in range, it may switch to the next available one. If no targets remain, the enemy might return to a default state, such as idle. Additionally, methods within EnemieAttackCollider.cs like AreAlliesInRange(), GetNextAllyInRange(), and RemoveAllyFromRange() dynamically manage the list of potential targets.

```
1
2   private void HandleAllyDeath(GameObject deadAlly) {
3       if (_chaseTarget == deadAlly) {
4           _enemyAttackCollider.RemoveAllyFromRange(deadAlly);
5           if (_enemyAttackCollider.AreAlliesInRange()) {
6
7               GameObject nextAlly = _enemyAttackCollider.
        GetNextAllyInRange();
8               if (nextAlly != null) {
9                   _chaseTarget = nextAlly;
10                  Attack(nextAlly);
11                  return;
12              }
13          }
14          Stop();
15          IsAttacking = false;
16      }
17  }
```

Listing 15: HandleAllyDeath function

For ranged enemies, EnemyShooting.cs script responsible the firing of projectiles at the player or allies. This script is activated when the enemy is in range and has been commanded to attack. It calculates the firing rate and incorporates an accuracy setting. Each shot accounts for possible deviation from the target based on the enemy's accuracy. The Shoot() method handles the instantiation of bullet projectiles, applying the necessary force and direction based on the enemy's current state and aiming accuracy.

```
1
2  public void Shoot() {
3      if (Time.time > fireCooldown) {
4          // Calculate deviation angle based on accuracy
5          float maxDeviation = (1 - accuracy) * 45;
6          float deviation = Random.Range(-maxDeviation,
       maxDeviation);
7
8          // Apply deviation to firepoint rotation
9          Quaternion deviationRotation = Quaternion.Euler(0,
       deviation, 0);
10         Quaternion bulletRotation = firePoint.rotation *
       deviationRotation;
11
12         Instantiate(bulletPrefab, firePoint.position,
       bulletRotation);
13
14         fireCooldown = Time.time + 1f / fireRate;
15     }
16 }
```

Listing 16: Shoot function

EnemyHealthController.cs script keeps track of how much health each enemy has. When enemies get hit by player attacks, the TakeDamage() function is used to reduce their health by the appropriate amount. If an enemy's health drops to zero, it means they're defeated. At that point, the Die() function is activated to remove the enemy from the game.

```
1
2  public void TakeDamage(int damage) {
3      health -= damage;
4      if (health <= 0) {
5          Destroy(gameObject);
6      }
7  }
```

Listing 17: Take Damage function

4.6 Abilities

There are 5 different abilities implemented in the game, each with their own special features to enhance the overall player experience. Some of these abilities can be active only in RTS mode, some only in FPS mode and some in both modes.

Power Up Ability

The power up ability implements a power-up ability that temporarily increases the player's damage output. It includes functionality for activating, deactivating, and managing the cooldown timer for the power-up. When the ActivatePowerUp() method is called,it increase the player's damage by increasing the damage multiplier, This increase affects all projectiles the player fires.It also sets the isActive flag to true, marking the start of the power-up phase. It initializes a timer timer to track the power-up's duration and resets cooldownTimer to ensure the ability goes through its cooldown phase once deactivated. After the set duration, or when the effect is supposed to end, the DeactivatePowerUp() method is automatically called. This method is responsible for returning the game to its normal state. It resets the damage multiplier, back to its original state. It also sets the isActive flag back to false, resets the timers, and updates the abilityImage to reflect that the ability is now in cooldown, waiting to be activated again.

Fig. 15: Power-up icon

Slow Time Ability

The slow time ability is responsible to slow down the pace of the game for a short period of time. It does this by applying a multiplier to the Time.timeScale property, causing all movement and actions to occur at half the normal speed. This includes the movement of enemies and bullets belong to enemy. When the ActivateSlowTime()

method is called, it triggers the slow time effect, affecting the movement of enemies and bullets. It accomplishes this by modifying the Time.timeScale property to slow down the game's overall pace. Additionally, it updates the movement speed of all enemies and bullets in the scene to match the slow time effect The DeactivateSlowTime() method is invoked when the slow time effect expires. It reverses the effects of the slow time effect, returning the game's pace and enemy/bullet velocities to normal.It sets the isActive flag to false, which indicates that the slow time effect is no longer in progress. It also resets the timer and cooldownTimer variables to 0.

```
private void ActivateSlowTime()
{
    GameObject[] enemies = GameObject.FindGameObjectsWithTag(
    TagConstants.ENEMY);
    foreach (GameObject enemy in enemies)
    {
        PathfindingEnemyMove enemyMove = enemy.
    GetComponentInParent<PathfindingEnemyMove>();
        if (enemyMove != null)
        {
            enemyMove.speed *= slowTimeMultiplier;
        }
    }
    Bullet.speedMultiplier = slowTimeMultiplier;
    Bullet.UpdateBulletSpeed();

    _isActive = true;
    _timer = 0f;
    _cooldownTimer = 0f;

    abilityImage.fillAmount = 1;
}
```

• Visual Feedback: Once the ability was used, the user can notice an indicator that fills up and rotates similar to a clock. This indicates the period that needs to pass before they can activate the ability again.

Teleport Ability

Teleport ability allows the player to instantly move to a specified location within a specified range. If the ability is not on cooldown and the player presses the specified key, it first calculates the target position based on the player's current pointer position using GetTeleportTargetPosition(). This method uses raycasting to find a valid

Fig. 16: Slow time icon

point within teleport range, only on surfaces on the ground. If a valid target location is found, it initiates a visual effect by changing the player's material to one of the teleportMaterials. This modification acts as a signal indicating that the teleportation process is currently underway. The actual teleport is delayed slightly using StartCoroutine(), which waits for a specified delay before moving the player to the target position. During this time, a particle effect [18] will be activated to visualize the teleportation process and a teleportEffectPrefab instance is created at the target position for additional visual feedback. After teleportation, the script resets the cooldown timer and begins counting up until it reaches the specified cooldown duration. During this time, the ability cannot be activated again. This is visually represented by gradually refilling the abilityImage over time.

Fig. 17: Teleport icon

Airstrike Ability This ability gives player the power to call down an airstrike on a target location. This feature is implemented across several scripts: Airstrike.cs, DebrisScript.cs, and LaserScript.cs, each handling different aspects of the ability. The script listens for the player's input to activate the airstrike. It checks if the game is in RTS mode and if the airstrike action flag is active. Upon activation and after ensuring the cooldown has elapsed, it calculates the airstrike position based on the current mouse position using GetMousePosition(). It then starts a coroutine StartLaser() to delay the

actual airstrike, giving a short anticipation period. Laser Deployment (StartLaser Coroutine): After a brief delay, it instantiates a laserPrefab at the calculated airstrike position. This prefab is responsible for the visual and damaging effects of the airstrike. LaserScript.cs script propels the laser downwards and handles collision with various objects. On collision, it checks the type of object hit and typically destroys it, simulating the destructive impact of the airstrike.Post-collision, it spawns multiple instances of debris using the debrisPrefab to enhance the visual destruction and chaos of the airstrike.

DebrisScript.cs script ensures that the debris gets cleaned up after a short period, removing them from the game to avoid unnecessary clutter and performance issues.

```
1  private void OnCollisionEnter(Collision collision)
2  {
3      if (collision.transform.CompareTag(TagConstants.ENEMY) ||
       collision.transform.CompareTag(TagConstants.PLAYER)||
       collision.transform.CompareTag(TagConstants.ENEMY))
4      {
5          Destroy(collision.gameObject);
6      }
7
8      var position = gameObject.transform.position;
9      position.y = 1;
10     for (int i = 0; i < 200; i++)
11     {
12         Instantiate(debrisPrefab, position, Quaternion.
       identity);
13     }
14
15     Destroy(gameObject);
16 }
```

Listing 18: Laser Collision and Debris Creation

Fig. 18: Airstrike icon

Healing Aura Ability Healing aura ability heals units in a given area for a set period of time. The core function, HealSelectedAlly() method is a part of the Healing Aura Ability's functionality, typically called when the player activates the ability. It iterates through all Selectable objects, checks if they are selected, and applies healing if they are alive. It also manages a visual healing effect that provides feedback to the player.

```
// ...
private void HealSelectedAlly()
{
    foreach (Selectable selectable in FindObjectsOfType<Selectable>())
    {
        if (selectable.IsSelected())
        {
            Killable allyHealth = selectable.GetComponent<Killable>();
            if (allyHealth != null && allyHealth.IsAlive)
            {
                allyHealth.Heal(healingFactor);
                GameObject healingEffect = allyHealth.transform.Find("Healing").gameObject;
                if (healingEffect != null)
                {
                    healingEffect.SetActive(true);
                    StartCoroutine(DisableHealingEffect(healingEffect, duration));
                }
            }
        }
    }
}
// ...
```

Listing 19: HealSelectedAlly function

The actual healing is implemented using a coroutine named HealOverTime, which is part of the Killable script attached to ally characters.

```

private IEnumerator HealOverTime(int totalHealAmount, float duration)
{
    float amountHealed = 0;
    float healPerSecond = totalHealAmount / duration;

    while (amountHealed < totalHealAmount)
    {
```

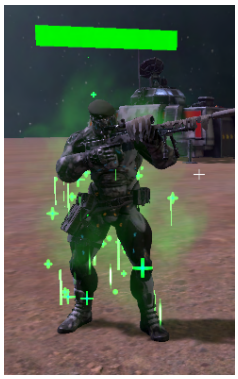

Fig. 19: Healing effect

```
9           int healThisFrame = Mathf.Min((int)healPerSecond,
    totalHealAmount - (int)amountHealed);
10          _hpCurrent = Mathf.Min(_hpCurrent + healThisFrame,
    hpMax);
11          amountHealed += healThisFrame;
12
13          UpdateHealthBar();
14          yield return new WaitForSeconds(1f);
15      }
16  }
```

Visual Feedback: The ability icon on the HUD is used to indicate the ability's cooldown, filling up or depleting based on the ability's usage. A particle system[18] or a healing effect is activated on the allies being healed, providing clear visual feedback of the healing process. This ensures players can visually track who is being healed and how long they have to wait before they can use the ability again

Fig. 20: Healing aura icon

4.7 UI/HUD

The in-game HUD is composed of four elements: an RTS HUD, an FPS HUD, a HUD swapper, and a minimap. Since the game is a blend of two different genres, it needs two distinct HUDs with similar art styles.

4.7.1 Minimap The minimap allows the player to see the position of objects on the scene in real time. The player can discern between allies, enemies, the main character, and different terrain elements. Since the player needs it to make informed and strategic decisions, it needs to be a reliable and easy-to-access source of information; therefore, the minimap is the only HUD element persistent between two HUDs, with its location unchanging. It works by being a target texture for a minimap camera. Each object has a minimap icon set to a specific 'Minimap' layer. It is only visible by the minimap camera.

4.7.2 FPS HUD FPS HUD is main character-focused. In the lower left corner, it contains a substantially larger health bar than an RTS HUD as well as icons for all abilities showing which ones are on or off cooldown. In addition, there is numerical information about the weapon magazine, which changes color when the weapon approaches zero bullets in the magazine to indicate a need for reloading. The whole screen is surrounded by a thin metallic frame to give the player a sense of being a heavily armored character.

4.7.3 RTS HUD RTS HUD takes the focus from the main character, who remains stationary in RTS mode and focuses on general map control. In RTS the HUD allows the player to place buildings, see materials and use RTS based abilities.

The upper bar shows general information about the game state, such as the game phase, the time remaining for the phase change, and the resources at the player's disposal. The lower bar shows reports about the events that are happening. It also provides feedback about user action. Each message appears for a limited time and has a cooldown that prevents the same message from being displayed

Fig. 21: FPS HUD

over and over. Any class can access the message via the Game Event System SendMessage() function.

```
1     public void SetMessage(string message)
2     {
3         if (CheckMessage(message))
4         {
5             StopCoroutine(Cooldown());
6             if(_tmp.text != null)
7             StopCoroutine(DisplayMessage());
8             _message = message;
9             _tmp.SetText(_message);
10            StartCoroutine(DisplayMessage());
11            StartCoroutine(Cooldown());
12        }
13    }
```

Listing 20: Message setting function

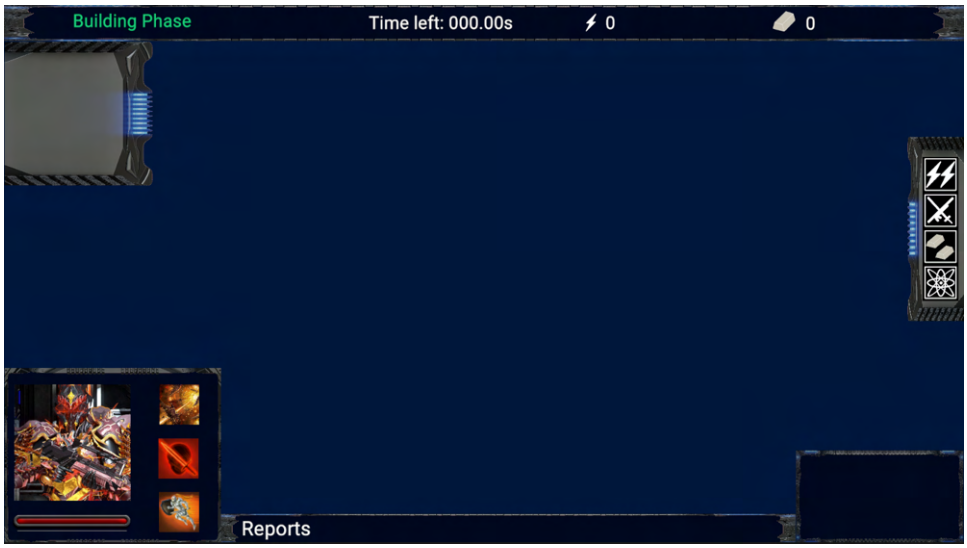

Fig. 22: RTS HUD

In the lower left corner, there is a hero tab. It provides information about the main character's health, although the health bar is noticeably smaller than in the FPS mode. There are also displays of hero abilities with cooldowns. Those displays, however, function as buttons for triggering abilities working in RTS mode. In addition, there is a portrait of a main character that, when clicked, centers a camera on the main character.

Fig. 23: Hero Tab

On the right, there is a bar for constructing buildings with buttons that allow the player to choose a new building construction site when clicked. Both building buttons and abilities change Action Flag, which changes the effect of a left mouse click on the map. The lower right corner bar contains a canvas of each building specific to the building that is currently clicked. It allows for training units and purchasing upgrades. It also has utility functions such as canceling training queues and destroying the building. The whole HUD utilizes semi-transparent black rectangles as well as the same bars as the FPS HUD.

4.7.4 HUD Swapper The HUD Swapper enables one HUD and disables another when changing the game mode. It calls a swapping function on the game mode change event [12] from the Game Event System.

```
1     void ChangeHUD(GameMode gameMode)
2     {
3         if (gameMode == GameMode.FPS_MODE)
4         {
5             fpsHUD.SetActive(true);
6             rtsHUD.SetActive(false);
7         }
8         else
9         {
10            fpsHUD.SetActive(false);
```

```
11              rtsHUD.SetActive(true);
12          }
```

Listing 21: Hud swapping function

4.7.5 Unit Bars Each unit and building has a health bar displayed above their head that is locked towards the player camera.

```
1   void FixedUpdate()
2   {
3       if (GameEventSystem.GameMode == GameMode.FPS_MODE)
4           transform.rotation = Quaternion.LookRotation(
    transform.position - _camera.transform.position);
5       else
6       {
7           Vector3 lookAtPosition = new Vector3(transform.
    position.x, _camera.position.y, transform.position.z);
8           transform.LookAt(lookAtPosition);
9           transform.rotation *= Quaternion.Euler(180, 0, 0)
    ;
10      }
11      //transform.localScale = new Vector3(1f, 1f, 1f);
12  }
```

Listing 22: Script for rotating Health Bars

The health bar color changes from red to yellow to green depending on percentage of health providing more visual feedback for player. In addition the health bar of building also serves as the progress bar of building construction. Moreover the building responsible for training units has a second bar responsible for a progress of current unit trained.

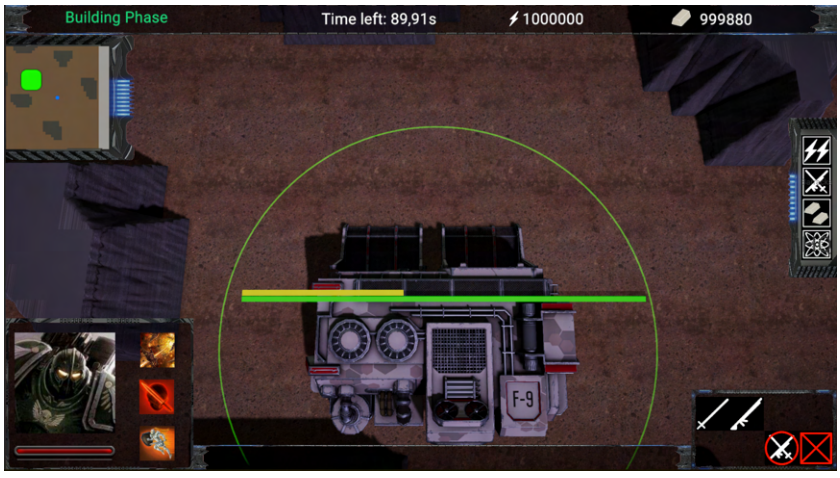

Fig. 24: Full health barracks in the middle of training units

4.7.6 Cursor The dynamic cursor implementation for an RTS game is involves two scripts: CursorManager.cs and CheckForEnemy.cs. These scripts changes the appearance of the cursor based on whether it's hovering over an enemy unit. This feature provides feedback and improves the games interactivity and responsiveness.

The CursorManager.cs script is responsible for changing the cursor's appearance. It uses two different textures: normalCrosshair for the default state and enemyCrosshair when hovering over an enemy. It defines defaultHotSpot and attackHotSpot to specify the precise point within the cursor image that will interact with elements on screen ensuring that cursor changes are intuitive and accurately centered. In OnEnable() and OnDisable(), the script subscribes and unsubscribes to the OnEnemyUnderCrosshair event. This event [12] is triggered whenever the mouse hovers over an enemy, indicating when to switch cursors. UpdateCrosshair(bool isEnemy) is the method called when the event is fired. It switches between the normal and enemy cursors based on the received boolean indicating enemy presence under the crosshair.

The CheckForEnemy script detects whether the player's cursor is hovering over an enemy unit. It works continuously to check the cursor's position and any underlying objects in the game world. Using the main camera, it casts a ray towards the point under the mouse cursor every frame. It then checks for collisions with objects on the enemyLayerMask, which includes all enemy units. When an enemy is detected under the cursor, the script triggers the OnEnemyUnderCrosshair event with a true value, and when no enemy is detected, it triggers the event [12] with a false value. This event [12] informs the CursorManager to update the cursor's appearance accordingly.

```
1     private void CheckForEnemyUnderCrosshair()
2     {
3         if (EventSystem.current != null && !EventSystem.
    current.IsPointerOverGameObject())
4         {
5             Ray ray = mainCamera.ScreenPointToRay(Input.
    mousePosition);
6             RaycastHit hit;
7             if (Physics.Raycast(ray, out hit, 100f,
    enemyLayerMask))
8             {
9                 OnEnemyUnderCrosshair?.Invoke(true);
```

```
10              }
11              else
12              {
13                  OnEnemyUnderCrosshair?.Invoke(false);
14              }
15          }
16      }
```

Listing 23: CheckForEnemyUnderCrosshair Function

4.7.7 Fps Counter The FPSCounter.cs script is designed to provide real-time updates on the frames per second. In FPS mode the script increments the frame count. It then checks if the current time has exceeded the lastInterval by more than the updateInterval. If so, it calculates the FPS as the number of frames divided by the total time elapsed since the last update. After calculation, it resets the frame count and updates the lastInterval to the current time. The calculated FPS is then formatted as a string and set as the text of the fpsText component, displaying it on the screen for the player to see.

4.8 Sounds

The game sounds are composed of three audio groups: music, effects and footsteps.

4.8.1 Music Composition The music is fully created Wiktor and his music teacher. The soundtrack consists of six music tracks. Two tracks are meant for calmer moments - the main menu, building phase, loading screen and so on. Two tracks are dynamic and quick, meant for combat scenarios. The last two tracks are meant to be played during the final animation. One of them is meant for a game that was lost and the second is meant as a victory theme. The music is mainly made using guitars, bass and drums, with some small mixing and electronic additions.

4.8.2 Music Implementation In the game, the tracks are attached to the main camera and are managed by a controller script responsible for changing music between phases as well as providing fade-ins and fade-outs.

```
    private IEnumerator PlayMusic()
    {
        while (true)
        {
            foreach (var song in _current)
            {

                _as.clip = song;
                _as.Play();
                for (int i = 0; i < 10; i++)
                {
                    _as.volume += 0.1f;
                    yield return new WaitForSeconds(0.3f);
                }
                yield return new WaitForSeconds(song.length
-6);
                for (int i = 0; i < 10; i++)
                {
                    _as.volume -= 0.1f;
                    yield return new WaitForSeconds(0.3f);
                }

            }
        }
    }
```

Listing 24: Coroutine handling playing music and transitions

4.8.3 Footsteps Player footsteps are selected randomly from a list of footsteps sound effects downloaded from the Unity Asset Store. Footsteps change pitch depending on whether the character is running or sprinting.

```
      public void PlayRandomFootstepSound()
       {
         _as.pitch = !Input.GetKey(PlayerConstants.RUN) ?
1.25f : 1.5f;
            if (_clips.Count > 0 && _clips != null && !
_as.isPlaying)
            {
                int randomIndex = Random.Range(0, _clips.
Count);
```

```
7                       AudioClip randomClip = _clips[randomIndex
    ];
8                       _as.PlayOneShot(randomClip);
9                   }
10              }
11
12
13          }
```

Listing 25: Script for playing random footstep

4.9 Lighting and Post Processing

Haardon is a science-fiction game and the lighting has to fit it's space setting.

4.9.1 Lights Game lighting consists of two directional lights: a light purple main light and a white secondary light. The light purple hue fits the space skybox downloaded from the Unity Asset Store, while the secondary light perpendicular to the main light softens the shadows cast on objects.

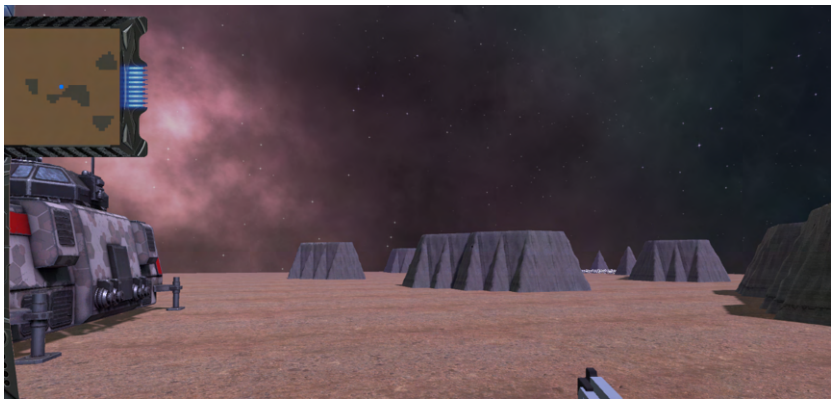

Fig. 25: Lighting with skybox with post-processing off

4.9.2 Post-Processing The game utilizes a variety of post-processing effects. Firstly, there is Color Grading with ACES tonemapping [11]. Secondly, the screen is surrounded by a vignette and thirdly, Sub-pixel Morphological Anti-aliasing is used. in addition, a Simple Space Ambient Occlusion from the Unity Asset Store is implemented.

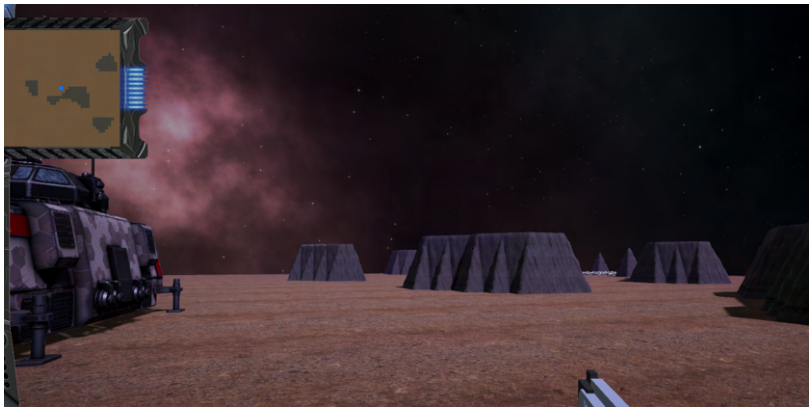

Fig. 26: A final result of lighting and post-processing

4.10 Outcomes from tests with users

5 Discussion

5.1 Challenges

During programming there were many challenges that warranted the use of new technologies. One of them was Unity's job system [6] which is basically Unity's implementation for multi-threading. It was used for path-finding in this game. The time required for this script was lowered from 50ms to 0.2ms. Most importantly, it was moved from main execution thread so it wouldn't stop rest of the game.

Another big challenge was around models and textures for game. The team spend around 30 hours speaking with different graphic artists in real life and using those found using world wide web. 3 different artists from PJATK were taken into consideration to help, but due to timing issues it wasn't possible. The second artist did creat this Mood Board, helping in creating visuals for the game:

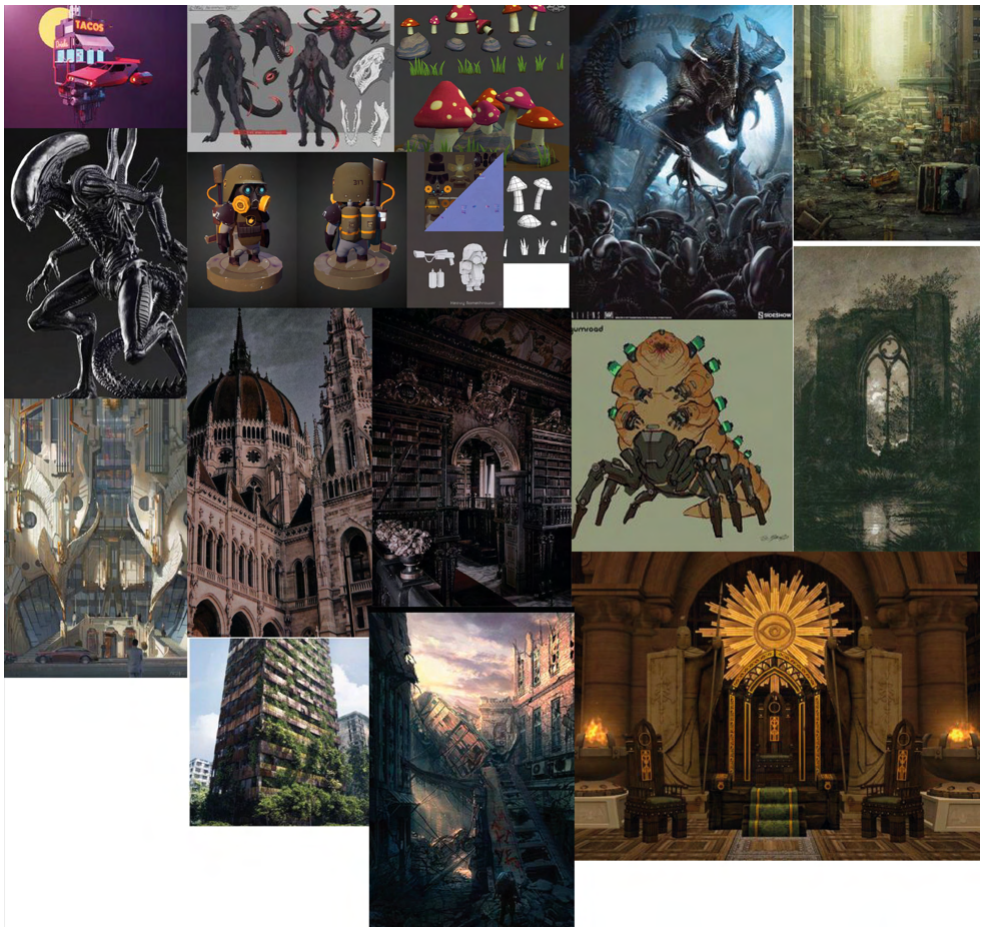

Fig. 27: Mood Board of the early gameconcept

5.2 Future work

5.2.1 Secret seed codes for world generation There's an idea of implementing a secret seed codes that will correspond to world generation. For example, a player can enter an e.g. "1337" code inside a seed input box, during creation of his world and he will see a really different world than a normal world generation system generates. Those are "easter eggs" for players that want to experience something other than usual.

5.2.2 Bullet holes In most FPS games there are bullet holes after the player shoots bullets from their weapon. That happens in real life too. In late version of game adding them is possible to make the game more immersive. [16] The player can have a lot of fun with this - they could make an inscription on the walls of buildings and mountains and then make a screenshot of it to share it someone.

5.3 Limitations

6 Conclusions

In conclusion, Haardon is a game that combines RTS and FPS elements of gameplay. This gameplay is a source of its uniqueness and significance. It shows a possibility for blending very different genres of video games to create a new type of experience for the player. It is yet to be seen if it paves a way for other games combining multiple modes of gameplay from different genres. The game mechanics and gameplay may inspire other developers in future to create something out of it or even a modification to the game that will change the whole gameplay.

References

1. About burst
   https://docs.unity3d.com/Packages/com.unity.burst@1.8/manual/index.html
2. Guid.newguid method
   https://learn.microsoft.com/en-us/dotnet/api/system.guid.newguid?view=net-8.0
3. Unity tutorials: How to save/load game using your own file (may 11 2020), https://weeklyhow.com/how-to-save-load-game-in-unity/
4. Singleton pattern (Apr 24 2022),
   https://blog.devgenius.io/the-singleton-pattern-in-unity-b7b3bc051a62
5. Two bone ik (Oct 18 2023),
   https://docs.unity3d.com/Packages/com.unity.animation.rigging@1.1/manual/constraints/TwoBoneIKConstraint.html
6. Job system overview (jun 13 2024),
   https://docs.unity3d.com/Manual/JobSystemOverview.html
7. Mathf (jun 13 2024),
   https://docs.unity3d.com/ScriptReference/Mathf.html
8. Mesh (jun 13 2024),
   https://docs.unity3d.com/ScriptReference/Mesh.html
9. Script serialization (jun 13 2024),
   https://docs.unity3d.com/Manual/script-Serialization.html
10. Thread safe types (jun 13 2024),
    https://docs.unity3d.com/Manual/JobSystemNativeContainer.html
11. Academy of Motion Picture Arts and Sciences: Academy color encoding system (aces) standard,
    https://www.oscars.org/science-technology/sci-tech-projects/aces, retrieved from the Oscars website
12. BillWagner, gewarren, nschonni, DCtheGeek, bradriske, jdmartinez36, mairaw, Youssef1313, nemrism, nxtn, TheSench, pkulikov, Mackiovello, mikeblome, mjhoffman65, guardrex, tompratt AQ: Events (c programming guide) (dec 04 2023), https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/events/
13. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics **4**(2), 100–107 (1968). https://doi.org/10.1109/TSSC.1968.300136
14. Perlin, K.: Making noise (dec 09 1999),
    https://web.archive.org/web/20071008162042/
    http://www.noisemachine.com/talk1/index.html
15. Thraka, MightyPen, tdykstra, Youssef1313, bravequickcleverfibreyarn, v kents, chessboards, bangseongbeom, GitHubPang, MichaelDeutschCoding, luizhlelis, DuncanSungWKim, pkulikov, BillWagner: Structure types (c# reference). Microsoft Documentation (Oct 2023),
    https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct
16. Valve Corporation: Half-life: 25th anniversary documentary (2023), https://www.youtube.com/watch?v=TbZ3HzvFEto
17. Zenva: What is procedural generation – complete guide (dec 23 2023), https://gamedevacademy.org/what-is-procedural-generation/
18. Zhang, B., Hu, W.: Particle system. In: 2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS). pp. 595–598. IEEE (2017)