COMPUTER GAMES PROGRAMMING REPORT

TILE-BASED STRATEGY GAME

MARCEL OCHS
HOCHSCHULE-KAISERSLAUTERN
STANDORT ZWEIBRÜCKEN
13.07.2023

PROF. DR. TEA MARASOVIĆ - UNIVERSITY OF SPLIT - FESB

Table of Contents

Introduction

The purpose of this report is to document the development process and outcomes of my solo project, a tile-based strategy game. Inspired by popular turn-based games like Pokémon, my objective was to create a unique gaming experience by combining the strategic elements of tile-based gameplay with the mechanics of Pokémon games.

The tile-based part of the game is inspired by Heroes of Might and Magic and Battle Brothers. In those games, it's necessary to think about every move you make carefully to not lose the battle. These kinds of games engage the player through their strategic aspect and variety of possible moves you can do.

The goal of the game I developed is to provide players with a challenging and tactical adventure set in a dynamic world. Through the strategic placement and movement of units on a grid-based map, players must navigate through a richly detailed environment, encountering various factions, resources, and quests. The tile-based mechanics allow for careful planning and decision-making, as players strategically control their monsters.

In this report, I will discuss the different parts of the game's development process: designing, making, and testing. I will talk about the monster mechanics, which includes the various types of monsters, their special abilities, and how their interactions affect the strategy of the game. I will also cover the important aspects of the game, like the battles between the player and the computer, how the game looks and feels, and the AI system. Finally, I will evaluate how well the game achieved its goals and whether it was successful or not.

Now that I have established the purpose and objectives of this report, I will proceed to explain the details of the planning, implementation, and testing phases.

Planning

Game concept

The game takes place in an exciting world where players start in a small town and receive their first monster companion. This world is filled with different kinds of monsters, each having their own unique strengths and abilities. Players train their monsters to battle against other people, aiming to become famous monster trainers.

As players get their first monster, they also face their first rival, which is an important challenge in their journey. Throughout the game, players can challenge other people in the town who also have monsters. Winning battles helps players gain experience, unlock new monsters, and earn rewards. Successful battles unlock access to new cities, expanding the adventure.

The game focuses on the bond between the player and their monster, strategic battles, and discovering new cities. Players aim to become skilled monster trainers, exploring the world and achieving greatness.

The game made for the report should be a prototype and feature most of the core features. The user should already be able to get his first monster, fight his rival, walk in the first city and fight at least one other opponent. The fight should cover a few different monster and more then 2 attacks to choose from and different movement patterns.

The city

The starting city is a small town with only one exit and a few houses. It's the first scene the player can interact with and discover the game. In that town should be a few NPCs (non-playable characters) standing or walking around. The player needs a script for him to move around and a model that's different to the NPC-models. The NPCs need to be drag and droppable into the scene and easy to configure to make it fast and easy for having unique battle experiences. A fightable NPC needs some kind of collision detector to detect the player and interact to start a battle scene. The city should be visually pleasing for the player.

Turn-Based Game-Loop

When interacting with an enemy, a new scene should be opened. The first thing that happens is a grid that randomly generates different kinds of maps. The map is made out of square tiles, which are interactable for the player and the AI. An example of a tile map is a chessboard where each piece is correctly placed at the beginning. The tiles should have different appearances and also have some effects on the monster. The width and length of a map should be adjustable, and different types of gameboards should be easily interchangeable. For example, some should have more grass tiles or some more water tiles, depending on the location of the player before he goes into combat.

After spawning the map and putting the monster on the tiles the next step is to order the characters in the right way. Because the game is in a loop until one side won each round should start with this part of the loop. After every round monsters can be defeated and need to get deleted out of the list and their initiative can also be different then in other rounds. That’s why the order of monster can always change between each round.

The next stage is to arrange the monsters in the right order. Because the game is in a loop until one side wins, each round should start with this part of the loop. After every round, monsters can be defeated and need to be deleted from the list, and their initiative can also be different than in other rounds. Because of this, the monsters' order can always vary between rounds.

Afterwards, the combat stage begins. In the combat stage, every monster has so many turns until they have no condition anymore and need a break, or the player tells them to take a break. In combat, monsters can rest, attack, or move. If the condition is empty, the next monster in the queue takes a turn. For attacking, every monster has different kinds of moves that can be single or multi-target and do damage to every creature they hit. If the health goes to zero, the monster gets deleted from the battle until one side has no characters left. This then leads to the end of the fight, and you return to the start map. If the player who interacts is the NPC, an AI chooses the next move; otherwise, the player should have an UI to interact with his monster.

After every turn of the monster, the status effects of the monster will be checked. For example, they can have the status burn and get additional damage, or they can have the status focused and gain extra conditions and initiative for the next round. After that, the loop goes either to the end of the game or back to the ordering of the monster.

If the player is the winner of the combat, each monster regains their health and gets some experience to level up. The scene should change back to the city scene in front of the enemy, which is not able to attack the player anymore.
If the enemy wins the player should either lose all the progress or spawn at a different location. Another idea for a more difficult mode that every monster that dies

it should stay dead and not be useable anymore. This feature would make players more careful and keep the monsters they don’t want to lose in the backline.

Monster, Movements and Attacks

The monsters in the game have various attributes like strength, initiative, and health points, making them different from one another in combat. Similar to the rock-paper-scissors principle, some monsters are stronger than certain types but weaker against others. For example, one monster may be strong against another but lose to a different kind of monster. This principle encourages players to think strategically about their actions and the choice of monsters.

To achieve this, the monsters in the game also have different nature types and usability types. Usability types include tanks, which have high health points but lower strength, and assassins, which have less health but deal high damage.

Regarding movement, the game considers the starting location of a monster, the destination, and any conditions required for certain movements to be possible.

The attacks in the game also have various attributes. These include the attack's range, the type of damage it inflicts, the amount of damage dealt, and the conditions required for the monster to use it. Additionally, attacks may provide buffs or debuffs, affecting the performance of a monster for a period of time. The attacks may hit multiple monsters, and there is a chance of successfully hitting the target.

By incorporating these attributes, the game encourages players to consider the strengths and weaknesses of their monsters, choose their actions carefully, and strategize their gameplay to achieve success.

Game Design

Engine

For the development of this project, I used the Unity engine as the main tool. It was used in the lecture, and I'm familiar with using it because of previous work that helped during the course and the project. Unity provided a user-friendly and versatile platform for creating and managing different aspects of the game, such as mechanics and the user interface. Additionally, I incorporated various assets from the Unity Asset Store, following the approach used in previous classes, to enhance the overall quality and streamline the development process. These assets contributed to improving the efficiency and visual appeal of the game.
For the project I used the Unity version 2021.3.14f1.

Prefabs

In the project, I made use of prefabs. Prefabs are pre-made objects or templates that can be easily reused in the game. They allowed me to create and configure objects like characters, items, and environmental elements, and then use them multiple times throughout the game. By using prefabs, I saved time and effort because I didn't have to create each object from scratch. Prefabs helped me maintain consistency and efficiency in the development process.

Enemy and Player

The first prefabs I created were for the enemy and the player. They both got a capsule as bodies and needed the "Rigidbody" script to apply physics to them. Because of the changes in height of the map I needed to check the freeze rotation buttons so the player and enemy didn't start rotating when going down a hill. In addition, I gave the player the tag "Player". Furthermore, I added a box in front of the enemy prefab to detect when you get in sight of the NPC. The box collider is set to "Is Trigger" so the player can walk through it. You can see the Prefab in figure 1. The enemy and player can get edited like that super easily and changed in the end to give them a better visual appearance.

Monster & Tiles

For the Monster to appear in the tiles later, I made prefab objects out of the FBX data. The tile objects also got different colors to tell apart which is water and which is grass. For that I made a simple green and blue material. In addition the tile got a 2D plane above them. The 2D plane is a picture of a square with a hole, so the player can later see where he is pointing the mouse. If the mouse is hovering above the tile, the GameObject plane will be turned on; otherwise, it will be turned off. This feedback should help the player know where he is going to press.

Manager

To ensure efficient management of important components in my game, such as the GameManager, UIManager, and MonsterManager, I implemented singleton patterns. I created empty GameObjects that contained a script responsible for keeping track of these managers. This approach helped me organize and access these critical components throughout the game, ensuring smooth functionality and streamlined control. Further details regarding the implementation will be discussed in the relevant section of the report.

Test level

In the Sample Scene in figure 2, I used one direction light, the main camera, the first enemy prefab, and my player prefab, which I created before. I also put a plane underneath the character because they used gravity and needed something to stand on. To differentiate the plate from the prefabs, I gave it the green material that I already used for the grass tile. I used this scene to test the player controls, enemy behavior, and scene change.

The other scene in figure 3 is for the main combat. In there, you can find, by opening it, a camera, a direction light, a canvas for all the UI elements, the Unity EventSystem, and my managers, which I created before. The GridManager, which instantiates all the tiles in the Scene, the GameManager, which instantiates all the monsters, and an UIManager. This scene is used for all the combat game loops, which are explained in Turn-Based Game-Loop. Most of the gameplay happens here.

Implementation

All the scripts in the following chapter can be find in the project folder under Assets/Scripts.

Scriptable objects

Scriptable Objects were utilized for the monster, attacks, and movement templates to provide a flexible and efficient way of creating and managing different variations of these elements in the game. By using Scriptable Objects, I could create reusable templates that separate the data from the code, allowing for easy modification and customization without modifying the underlying codebase. This approach enabled me to create and modify various monster types, attacks, and movement patterns quickly, enhancing the scalability and modifiability of the game. In figure 4 you can see how to add a new monster.

Monster base

To be able to create a scriptable object the class needs to inherit from the ScriptableObject class. In addition the CreateAssetMenu attribute is used to create a custom asset creation menu in the Unity Editor. By adding this attribute to a script, you can right-click and navigate to the specified menu path.

The MonsterBase class has a lot of attributes like maxHP, monsterName, initiative to make all the monster unique. It also has a list of AttackBase and MovementBase variable to allow the monster interacting during the fight.

Besides all the setter and getter there is only one important method. The GetDamage() Method receives a number and subtracts it from the current health points of the monster. If the health points reach 0 the monster gets deleted from the battle.

Player movement

For the movement of the player character in my game, I created two scripts. The first script, CharacterMovement, handles the actual movement and rotation of the character. It receives input from the second script, InputVector, which captures the player's input from the horizontal and vertical axes. The CharacterMovement script uses the input values to calculate the direction of movement and rotation for the character. By separating the movement logic into these two scripts, I was able to have a clear and organized system for controlling the player's movements in the game.

Character Movement

The CharacterMovement script handles the movement and rotation of the character in the game. It receives input from the InputVector script to determine the direction of movement. The script contains references the player model GameObject, moveSpeed, and rotateSpeed variables. All the variables are private but with the [SerializeField] they can be adjusted in the Unity Editor.
In the Awake() method, the script retrieves the InputVector component attached to the same GameObject.
In the Update() method, which is called once per frame, the script calculates the targetVector based on the input received. It then calls the MoveTowardVector() and RotateTowardVector() methods to move and rotate the character accordingly.
The RotateTowardVector() method rotates the player model to face the targetVector, while the MoveTowardVector() method translates the character's position based on the targetVector and the moveSpeed variable.
The getMoveSpeed() method allows external scripts to access the moveSpeed value.

Input Vector

The InputVector script handles input from the player, specifically the horizontal and vertical axes (e.g., arrow keys or WASD). In the Update() method, it retrieves the input values using the Input.GetAxis() function and assigns them to the _inputVector variable, which represents the direction the character should move. This _inputVector can be accessed by other scripts to determine the player's input for movement calculations.

Enemy behaviour

For the enemy movement I only needed one script called MovementBehaviour. It controls how enemies move around in the game, making their actions dynamic and engaging. To determine which monster the player and non-player characters have, I used a script called MonsterPossession. This script influenced their abilities and how they interacted with each other. Additionally, I created a script that detected collisions between the player and enemies, triggering the combat sequences.

Movement Behaviour

The MovementBehaviour script is responsible for controlling the movement behavior of game objects in the game. The script includes variables for speed, distance, movement type, and flags to track movement direction. In the inspector you can easily change the type of movement of the NPC, the duration of the movement and how far he is walking.

In the Update() method, the Move() function is called to handle the actual movement. For walking movement types, the object is translated horizontally based on the specified speed and time. The movement is restricted within the defined distance range, and the object's rotation is adjusted accordingly when reaching the limits. The script also includes a rotating movement type, which is not yet implemented because there is no need for the variety of movements. Because it’s an important feature for the future it’s already included. The MovementType enum defines three movement options: standing, walking, and rotating.

Monster Possession

The MonsterPossesion script manages the possession of monsters by the player or non-player characters. The script contains the list allMonsterTemplate which contains all the templates of monster and the level so if they get created they have the correct amount of health and damage for example. It also has a list of vector2 pos so that the monsters can get positioned in the right starting place. Another list, allMonster, is used to hold the actual instantiated monsters.

Whenever the start methode gets called the script will call the CreateMonsters() function. This function clears the allMonster list and then iterates through the allMonsterTemplate list. For each template monster, a new instance is created using Instantiate(), and its Initialize() method is called to set the level, experience type, and position based on the corresponding values from the lists. The instantiated monsters are added to the allMonster list.

By using the MonsterPossesion script, the game can create and manage multiple monsters, initializing them with specific attributes and positions. This script allows for dynamic and customizable monster possession within the game and also in the inspector for fast creation of different levels in the future.

Battle Interact

The BattleInteract script is a simple script attached to the collider children of NPCs to detect if they are in proximity to the player. When the collider triggers with another collider tagged as "Player," the OnTriggerEnter() method is called. If the player is in they are “in sight” of the enemy.

In the OnTriggerEnter() method, the script retrieves the lists of monsters owned by the NPC and the player. It accesses the MonsterPossesion component of the NPC's

parent object to get the enemyMonster list, and it accesses the MonsterPossesion component of the other collider's game object (which should be the player) to get the ownMonster list.
Next, the script uses the MonsterManager.MonsterInstance.setMonster() method to set the monsters for the upcoming battle. It passes the ownMonster list and the enemyMonster list to the setMonster() method. So that in the combat the right monsters will be spawned with the correct health. The next scene will be loaded with the SceneManager.

Scene change

For changing the Scene the MonsterManager Script is needed. With the DontDestroyOnLoad method it’s transferred to the combat scene. Another important Script is the GameManager which has the game loop logic inside.

Monster Manager

The Monster Manager script is a singleton. It has an instance of its own script, which gets called by creating the script. So there is always only one instance of the object. Because the instance is public and static, it can be called from every other script without a reference to the game object. All the managers walk this way to make them easily accessible and to deny the chance of multiple instances.

All the variables have easy-to-access getters and setters, so they can be read or rewritten without changing the variable directly.

For spawning the monster in the right position the script has two methods. When SpawnOwnMonster() or SpawnEnemyMonster() are called the monster of the team spawns onto the game grid. It iterates through the list of monster, sets their team, adds them to the allMonster list, and instantiates their corresponding monster prefab at a specific position on the grid.

The OrderMonsterList() method orders all monsters based on each initiative value, ensuring the monsters take turns in the correct order. The monsters get added to a queue in which they can be picked out in the right order.

Another important method, which is called chargeCondition(), is what the manager takes care of when charging the condition back up. This method needs to be called after every round to ensure that all the monsters are able to attack after the new round starts.

The AreMonstersRemaining() method checks if there are any monsters left for a specific team. By providing the team as a parameter, it returns a true or false value. If there are still monsters for the team, it returns true; if there are no monsters left, it returns false. This method helps determine if there are active monsters remaining for a particular team.

Game Manager

The GameManager script handles the combat's overall state and progression. It includes various methods and a GameState enum that determine the current state of the game. By incorporating different methods and functionalities, the GameManager script ensures that the game progresses smoothly and follows a logical sequence of events. It orchestrates actions such as generating the game grid, spawning own and enemy monsters, ordering monster turns, facilitating player interactions, determining game over conditions, and more.

The ChangeState() method in the GameManager script is responsible for facilitating transitions between different game states. When invoked with a new state parameter, it updates the GameState variable accordingly and triggers the corresponding actions. For instance, when the game enters the GenerateGrid state, the method calls the GenerateGrid() function from the GridManager script to generate the game grid. This ensures a seamless and organized progression throughout the game by executing the necessary functions and actions associated with each state.

The MonsterTurn() method is called during the MonsterTurn state. It first checks if the game is in the GameOver state or if there are no more monsters in the MonsterQueue. If either condition is true, it transitions to the OrderMonster state to proceed with the next phase. However, if there is a current monster in the MonsterQueue, it checks the condition of that monster. If the condition is positive, it enables player interaction if it's the player's turn or triggers AI behavior if it's the enemy's turn. On the other hand, if the condition is zero or negative, indicating that the monster cannot take any action, it removes the monster from the MonsterQueue and calls the MonsterTurn() method again to continue the sequence with the next monster in line. This process ensures a smooth and efficient flow of monster turns within the game.

Overall, the GameManager script controls the flow of the game by managing the game state and executing the appropriate actions based on the current state.

The game grid

The grid system plays a fundamental role in the game. It serves as a visual representation of the game world, consisting of interconnected tiles that facilitate movement, positioning, and interaction. The grid system enables strategic planning, pathfinding, and object placement, laying the foundation for engaging gameplay mechanics and dynamic encounters. The grid system consists of two parts, one is the gridmanager and the other is the tile script.

Tile

The Tile script consists of two important methods, one of them is OnMouseDown(). This method is activated when the player clicks on a tile. It examines the tile's current state to determine if it can be moved to or attacked. If the tile is movable, the method triggers the moveTo() function, relocating the current monster from the MonsterQueue to the clicked tile. This action modifies the grid by updating the occupied and unoccupied tiles, decreases the monster's condition based on the cost of movement, and refreshes the game status user interface. In the case of an attackable tile, the method generates an AttackBehaviour instance and performs the DoDmg() action, causing damage to the target based on the attack type and tile position. Subsequently, it updates the grid to indicate that the tile is no longer attackable. If the game state is not GameOver, the method transitions to the MonsterTurn state, allowing the next monster to commence its turn.

The other important method which was called before, the moveTo() function moves the current monster to the tile's position. It first obtains the current monster's prefab and the grid instance. It updates the grid by setting the clicked tile as occupied and the previously occupied tile as unoccupied. It then sets the current monster's position to the tile's position, ensuring it visually aligns with the clicked tile. Additionally, it reduces the monster's condition based on the move cost defined by the moveType. Finally, it updates the game status UI to reflect the changes.

Grid Manager

The GenerateGrid() method in the GridManager script is responsible for creating the game grid. It initializes the tiles and occupiedTiles dictionaries to store the tile objects and occupied game objects, respectively. Using nested for loops, it iterates through the desired width and height of the grid. For each grid position, it randomly selects a tile type (waterTile or grassTile) to instantiate at the corresponding position in the 3D space. The spawned tile is assigned a unique name based on its coordinates. The method then adds the spawned tile to the tiles dictionary, using its position as the key. After generating the grid, the camera's position is adjusted to center it over the grid. Finally, the game state is changed to SpawnOwnMonster, allowing the spawning of the player's monsters to commence.

In the GridManager script, both the instantiated tiles and the spawned monsters are stored in dictionaries using Vector2 positions as keys. These dictionaries allow for efficient retrieval and modification of tile and monster data based on their positions. By utilizing the Set and Get methods associated with the dictionaries, specific tiles or monsters can be accessed and modified as needed. Additionally, the dictionaries provide a Boolean function that allows for quick checks to determine if a specific position is occupied or not. This data structure ensures convenient and organized management of the tiles and monsters within the game grid.

Combat AI

The MonsterAI script is a vital component responsible for guiding the behavior of enemy monsters during the game. It employs several key methods that contribute to their strategic decision-making. For example, the CheckCondition() method assesses the monster's current condition, determining if it has enough resources to engage in an attack or movement. This ensures that monsters are capable of executing actions when necessary.
During their turn, the PerformTurn() method evaluates the available options based on the monster's condition. It considers factors such as proximity to the nearest enemy and available attack moves or movements. The enemy monster may either initiate an attack through the AttackEnemy() method or move closer to the enemy using the MoveToEnemy() method. The FindNearestEnemy() method facilitates the identification of the closest enemy monster, assisting the AI in making informed decisions.
Overall, the MonsterAI script empowers enemy monsters with decision-making abilities. It’s just a simple way to make the enemy move without lots of effort. The enemy AI can be made stronger in future of the project with different level of difficulty.

UI Manager

The UIManager script is responsible for managing the functionality of UI buttons in the game. It controls the activation and deactivation of buttons as needed during different stages of the fight. The script also handles the implementation of button actions, such as attacking an enemy, ensuring that the correct actions are triggered when the corresponding buttons are pressed. This centralized approach simplifies the management and control of button behaviors, which should lead to a smooth and intuitive user interface experience.

Level Design

To achieve a nice look I first wanted to make Blender models for every assets myself, but because of the limited time I ended up using following Unity assets from the asset store.

Assets used:

Low Poly Game Ready History Village, Low Poly Megapack – Lite, Free Stylized Skybox, Environment Pack: Free Forest Sample, Toony Tiny People Demo, Free Low Poly Nature Forest

Design

In order to enhance the player's immersion and create an engaging game world, I designed a starting city within the game. This city features two houses, a well, and a museum, which provide points of interest and opportunities for exploration. The city is nestled amidst picturesque mountains, creating a sense of isolation and setting the stage for exciting adventures to come. The surrounding mountains not only serve as a visually appealing backdrop but also serve a practical purpose by enclosing the city and limiting the player's movement. This design choice encourages players to explore the city and interact with its inhabitants and establishing a starting point for their adventures.

I made some changes to the game's visuals to improve the player's experience. The player character was transformed from a simple capsule to a friendly-looking little boy, giving them a more relatable appearance. On the other hand, the enemy's mesh was updated to make them appear more menacing by equipping them with a weapon. These visual modifications aim to enhance the overall gameplay and create a more engaging and immersive experience for players.

The finished world can be seen in figure 5. On the left screen is the whole map and on the right screen the current gameplay.

Conclusion

The process of developing the prototype for the game was largely successful, although it encountered some bugs along the way. One area that required extensive testing and refinement was the implementation of the game loop process within the grid. This process took longer than anticipated, causing some objectives to remain unachieved within the given timeframe. However, despite these challenges, valuable insights were gained through continuous testing, and the majority of the prototype's goals were accomplished.

Due to time constraints, several elements that contribute to a good user experience were omitted in the current prototype. As a result, it is challenging for players to discern the actions of the enemy, as this aspect was not fully implemented. While this limitation may impact the overall clarity of gameplay interactions, it is important to note that the focus of the prototype was primarily on core mechanics and functionality. Future iterations can address these issues and prioritize enhancing the user experience to provide clearer feedback and more engaging gameplay.

I am pleased with the outcome and will persist in developing the game even after the course concludes, aiming to eventually release it when it is fully polished. The prototype has revealed that the development process may demand more time than initially anticipated. However, I am confident that with dedication and diligence, the goals can be achieved.

List of figures

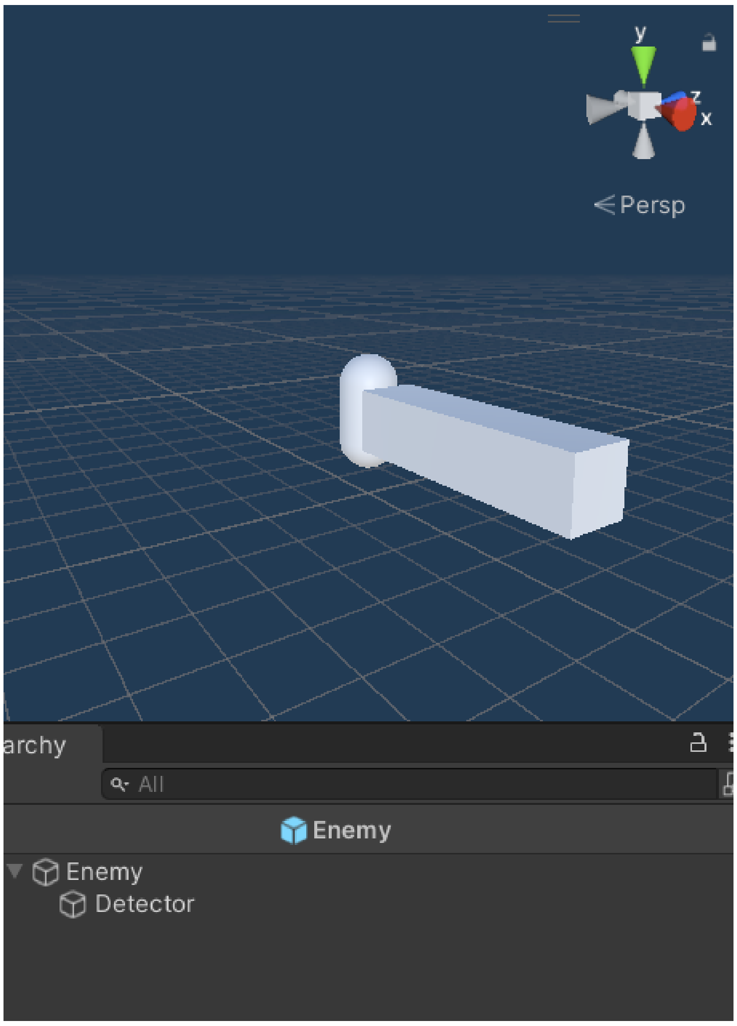

Figure 1: Enemy prefab

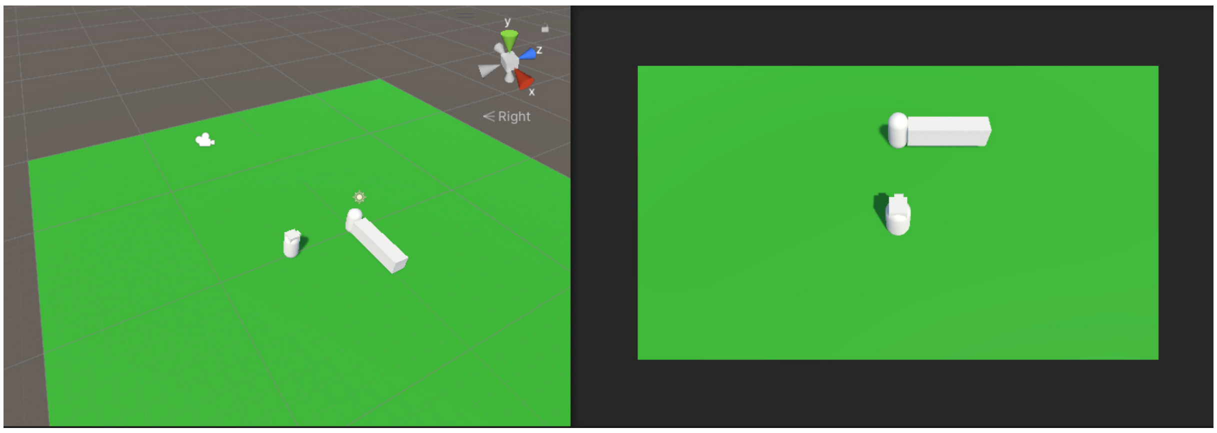

figure 2: Sample Scene

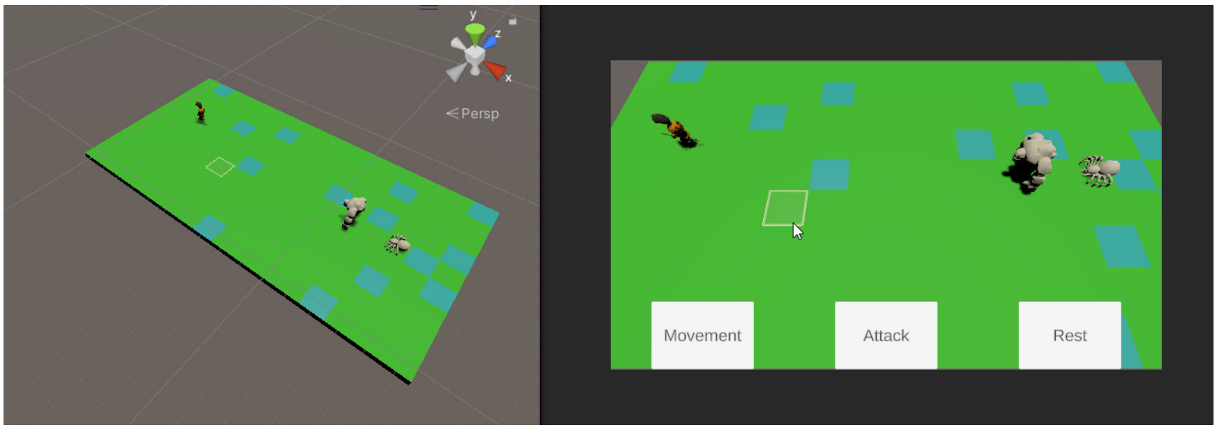

Figure 3: Combat Scene

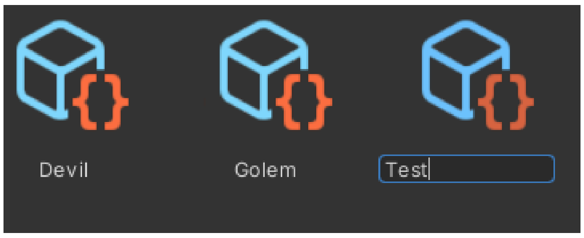

Figure 4: Configurate new monster

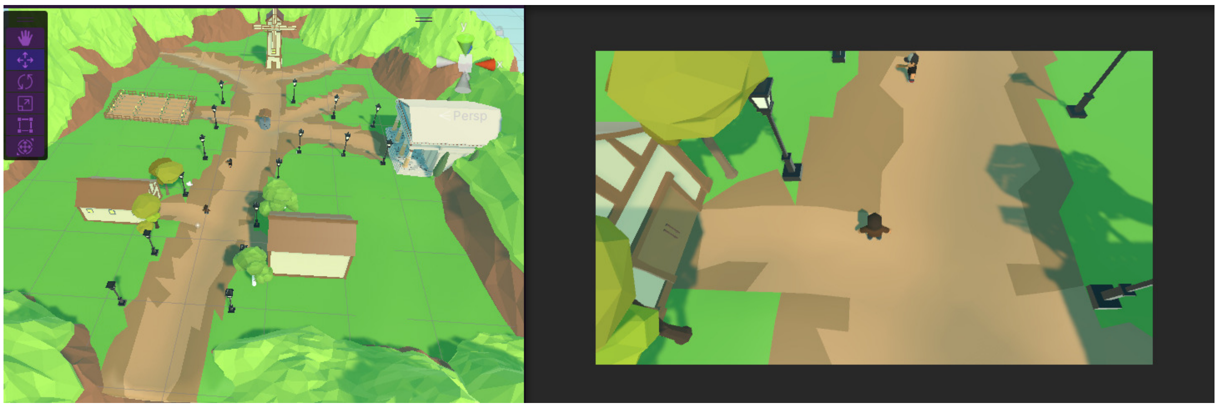

Figure 5: the finished map