Normative and Dynamic Virtual Worlds 20-21
The Lost Realm

Aglaia - Elli Galata, Michał Choiński, Jason Klimack

January 12, 2021

Contents

1 Introduction

1.1 Game Description

In this work, we create a 2D action-adventure game, comprised of a single quest on randomly generated maps. The inspiration for the game is 2-fold: 1) we wanted to create a *The Legends of Zelda* type game while 2) using randomly generated maps, such as those done in *Age of Empires*, to play through a single short scenario.

The main goal of the project is to use Procedural Content Generation (PCG) to generate a random map that provides the user with satisfying gameplay. In order for the map to be aesthetically pleasing, it must be able to simulate realistic natural formations, such as rivers and forests, as well as include the use of structures such as castles and bridges.

The game is initialized with a player wandering in a lost realm. The goal of the player is to reach a hidden castle, which is protected by a Boss. However, multiple hazards appear in this long road, and the player must use their cunning and strategy to prevail.

1.2 Scenario

Before the game begins, the player can choose the difficulty of the game from an initial menu. The available options are:

- Easy:adequate for familiarizing with the game rules,
- Normal: winning the game needs exploration of the map and collecting some items,
- Hard: also requires strategic thinking

Along the path to the castle of the realm, the player encounters enemy soldiers, who attack him when they realize his presence. To defend himself, the player needs to secretly explore the terrain and retrieve weapons that will fortify him against the enemies. Such weapons can increase his endurance in suffering hits or increase the damage that he enforces. Furthermore, the terrain has small hearts that increase the health of the player. The game ends in one of two final conditions.

1. Victory: The player kills the boss and enters the castle
2. Loss: The player gets killed.

1.3 AI Design

There are two main AI components in our game: PCG for map generation, and NPC behaviours. Since the terrain is procedurally generated at run-time, the player will encounter a different map in each new game independently of his chosen difficulty. Some of the advantages of procedural content generation (PCG) are listed below:

- Dynamic content creation
- Saving on memory usage
- Diminishing development time
- Offering a large variety of options
- Increasing replayability

The controlling of the enemies is performed through finite state machines. Furthermore, their navigation, either on the patrolling state or on the pursuing state, is based on the A* algorithm, which allows tracking of the player, and prevents collisions against non-walkable locations. [1]

1.4 Document Structure

This document is organized as follows: First, in section 2, the steps behind the procedural content generation will be discussed. Next, in section 3, the controlling of NPC characters will be introduced, and in section 4, the game's characteristics will be analyzed. Furthermore, in section 5, a deeper examination is performed on the game's experimental set-up, and in section 6 game's simulations are illustrated. Finally, in section 7 our conclusions over the report are stated.

2 Map Generation

2.1 Map Components

An important aspect of any game is the map design. Without good aesthetics and components of a map, the characters would appear to be walking in a void, which would result in a very boring game.

For 2D grid-based games, such as *Super Mario World*, *The Legend of Zelda: Link to the Past*, or our *The Lost World*, a common approach to creating a map is to use a tilemap. Tilemaps are a grid of small images called tiles, typically square in shape. The tiles set for creating a tilemap is called a tile-set and is usually created from a sprite sheet. Sprite sheets are simply an image with a series of textures, which are then sliced into cells to form the tile-set. Each tile in the tile-set represents a unique texture and can be placed adjacent to other tiles of the same or varying texture on the tilemap to obtain a larger scene. Figure 1 demonstrates the use of a sprite sheet to create a tilemap.

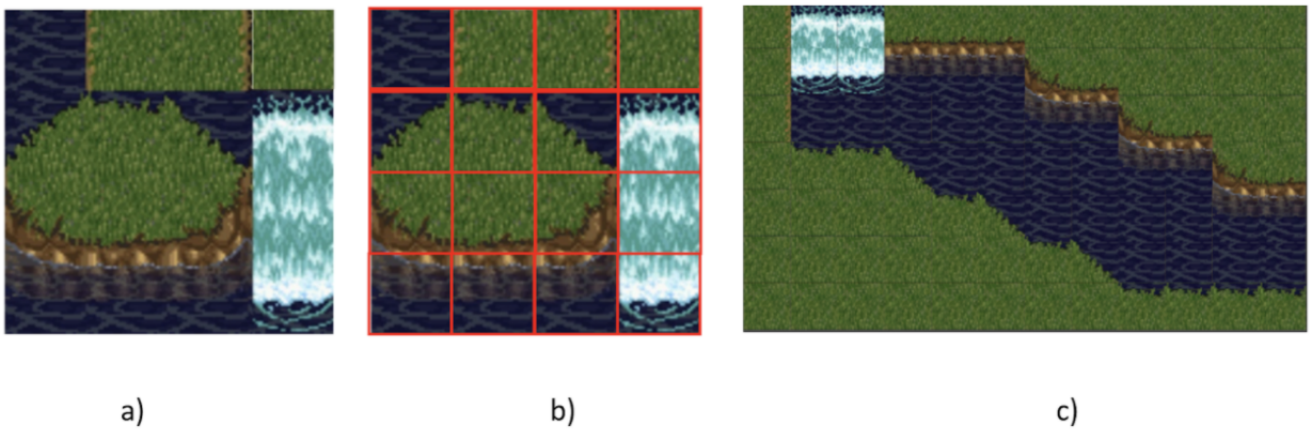

Figure 1: Example use of a sprite sheet to create a tilemap. a) the sprite sheet, b) tileset made by slicing the sprite sheet into square components, c) tilemap created by arranging tiles from the tileset in such a way that a scene is formed.

The Unity game engine has built-in Tilemap and Tile objects, as well as a slicer for obtaining the individual tiles from the sprite sheet. We used these default unity tools to create tilemaps for our game. Also included in Unity are the necessary tools for manually drawing a tilemap from a tile palette (which is the Unity version of the tile-set). However, while this tool is straightforward to use, our game's goal is to generate the maps dynamically using PCG. Thus, we used neither the tile palette nor the tile editor from Unity. Instead, we assigned our tiles to the tilemap programmatically at runtime. In order to prepare the maps to be created at runtime, three main steps need to occur:

- Create tile assets
- Define the tilemaps
- Load assets in the script and apply them to the tilemaps

In the first step, we retrieved, and appropriately sliced various sprite sheets [2, 3] into 32×32 pixel cells, using Unity's slicer. Each resulting tile is saved as a Unity asset. The sprite sheets that we selected are all of a top-down type, with

a similar artistic style. One of the main conditions incorporated in the selection was that the sprite sheets must include: grass, water, trees, and components to create a castle. The complete sprite sheet from [2] is shown in figure 2. The sprite sheet from [3] is too large to display, but it contains various grass types, trees, water, building structures, rocks and cliffs, and other miscellaneous items.

Figure 2: The complete sprite sheet created by [2]

The second step consists of creating three separate tilemap objects in Unity: a ground tilemap, a collision tilemap, and an aesthetic tilemap. Each of these three tilemaps is placed on a separate layer. There are two reasons for creating multiple layers in this game. The first reason is to allow multiple objects to occupy the same grid-cell. This is necessary where some of the sprite sheet objects have a transparent background and hence do not occupy the entire cell. Figure 3 shows an example of placing a plant on a grass background. If the plant object and grass objects were on the same layer, the result would have an unnatural square-shaped hole. Alternatively, using separate layers for the grass and the plant allows the scene to have a natural appearance. The second reason is to allow for the addition of collision detection between the player and certain objects. For example, if we want the player to walk on the grass but not be able to pass through the plant, we need to add a collision component to the plant. This is easily done in Unity by adding Collider components to a Tilemap, which then creates a boundary around all of the objects located on that tilemap simultaneously. Thus, we create a separate tilemap for objects that we do not want the player to pass through. Therefore, the ground layer is used for placing tiles such as grass that we allow the characters to walk on, the collision layer is used for the tiles that we do not want the characters to pass through, such as water, trees, and castle walls, and the aesthetic layer is used for placing additional objects on the map, such as the castle doorway or the plant from figure 3, where we do not want the player to collide, but instead, pass through the object.

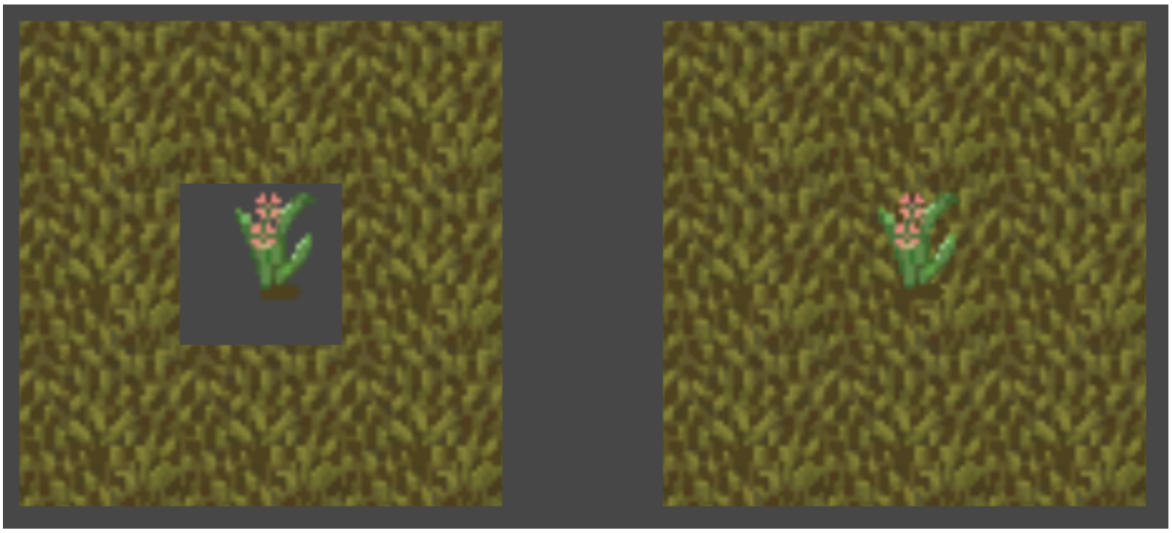

Figure 3: Demonstration of the necessity of multiple layers in a tile-map.

The third step is to use C# script to load the Tile assets, build a map, and apply the correct tile to each cell in the map on the appropriate layer. Each Tile asset that we use in our map is individually loaded and stored into an array of Tiles for convenience. Next, we use PCG to generate a 2D matrix of terrain types, representing the tile type at each location on the entire map. The main terrain types that we used for our map are:

- Water
- Grass
- Tree
- Cliff / Rock
- Bridge
- Castle
- House

While bridge, castle, and house are not naturally occurring terrain types, we include them in our definition of terrain types in order to simplify the tile allocation process defined in section 2.2.5, where tiles are assigned to the tilemap based on the output of the PCG generated map.

Another common practice for tilemaps is to use transition tiles between different terrain types. These transition tiles aim to increase the aesthetic appearance of the scene by providing a more natural flow between tiles. Figure 4 demonstrates the use of transition tiles between water and grass terrain types. We incorporate some transition tiles into our map using a set of rules. These transitions and rules are further described in section 2.2.5.

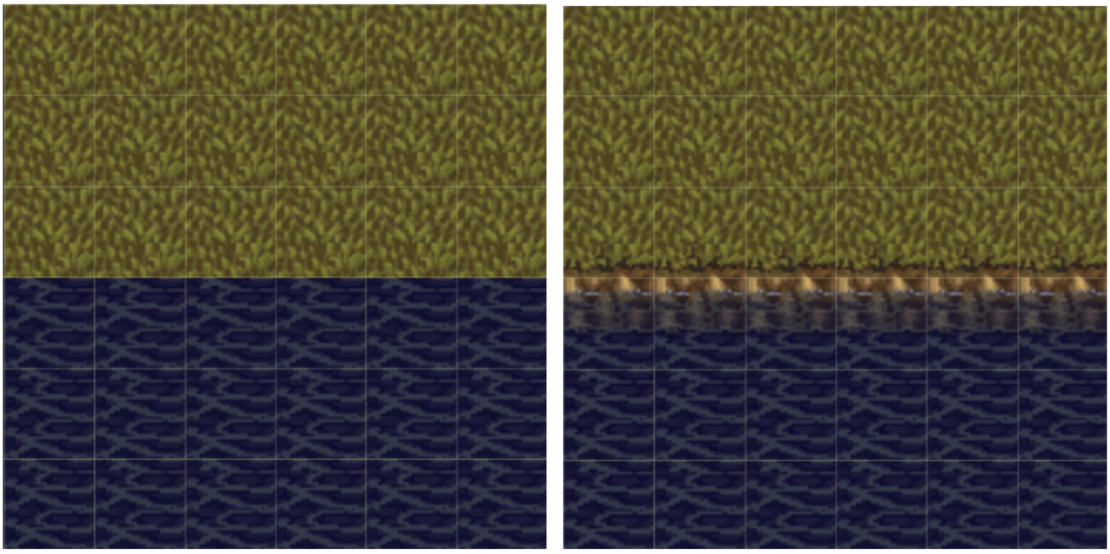

Figure 4: Example usage of transition tiles between water and grass: left: no transition tiles, and right: transition tiles.

2.2 Procedural Content Generation

In the scope of the map generation, different algorithms were employed. First, Perlin noise was applied independently to different terrain types such that the resulting map provides a natural appearing scene. Second, we create an algorithm for path connections to link all of the walkable areas in the map and do not isolate any characters or the player himself in a non-accessible location.

Finally, the placement of the castle, characters, and items is performed in a way that does not change the game's difficulty and preserves its characteristics. For instance, random placement of the player next to the castle would bias the game's difficulty since he would not have to explore and traverse through the map's hazards. The logic behind the Perlin noise, path connections, castle, character and item placement is analyzed in the next sections.

2.2.1 Perlin noise

Perlin noise developed by Ken Perlin is a procedural texture primitive; a type of gradient noise used by visual effects artists to increase the appearance of realism in computer graphics. [4]

Given a seed value, the Perlin noise algorithm will generate a grid of pseudorandom numbers in the range [0,1], with gradual transitions between values nearby on the grid. We can then utilize the algorithm by assigning a terrain type (grass, trees, etc.) to the different ranges of values [0,1]. This allows for the generation of a smooth, naturally appearing terrain.

Our initial implementation for creating the grid-map used a single instance of Perlin noise to generate the trees, water, and grass simultaneously (see figure 5). While the resulting map does indeed have nice smooth regions of different terrain types, the forests' geographical locations are not natural. A forest is not always completely bordered by a single terrain type (in this case, water). Alternatively, a forest could border a river on one side, while the other side is adjacent to grass fields. Alternatively, a single forest in the middle of a grass field can have a river run through its center. Given a single instance of Perlin noise, a natural separation of these three terrain types is not possible.

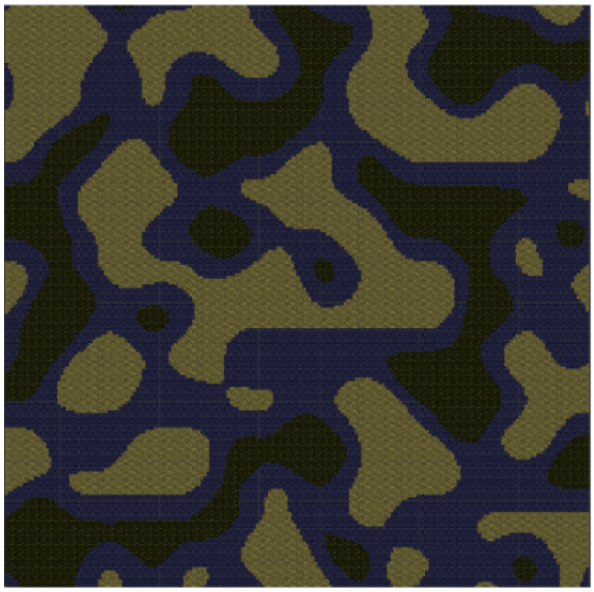

Figure 5: Generating trees (dark-green), grass (light-green), and water (blue) at once using a single instance of Perlin noise.

To solve the problem of unnaturally appearing forests and rivers, we apply Perlin noise at different levels for the different terrain types: rocks, grass, water, and trees. First, the map is initialized using only the forest and grass terrain types (see figure 6.a). The range covered by each terrain type in the Perlin map are manually adjusted such that the map has a natural balance of trees to grass. From the generated Perlin map, we assign terrain types by the following ranges:

1. $[0, 0.5] = grass$
2. $[0.5, 1.0] = tree$

Second, rocks are created similarly with the parameters set so that they are slightly more sparse than the trees (see figure 6.b). The rocks are created on a separate grid from the trees and grass. The ranges for the terrain types generated with the rocks are:

1. $[0, 0.2] = rock$
2. $[0.2, 1.0] = tree$ (as we see later, the value of this terrain does not matter).

Thirdly, to create rivers, we follow the same approach as the rocks, with the difference of assigning water and adjusting the parameters so that the rivers follow a narrow path rather than a singular region (shown in figure 6.c). This was done by setting three ranges for the Perlin noise:

1. $[0, 0.47] = grass$
2. $[0.47, 0.52] = water$
3. $[0.52, 1] = grass$

Finally, we use a mask to separate the rocks and rivers from their respective grids and apply them to the initial map with the trees and grass. In order to achieve uninterrupted rivers, the rocks are applied first, followed by the water. Figure 6.d shows the final map terrain after applying Perlin noise. The result of multiple instances of Perlin noise being applied to the terrain is that the map appears far more natural than a single instance.

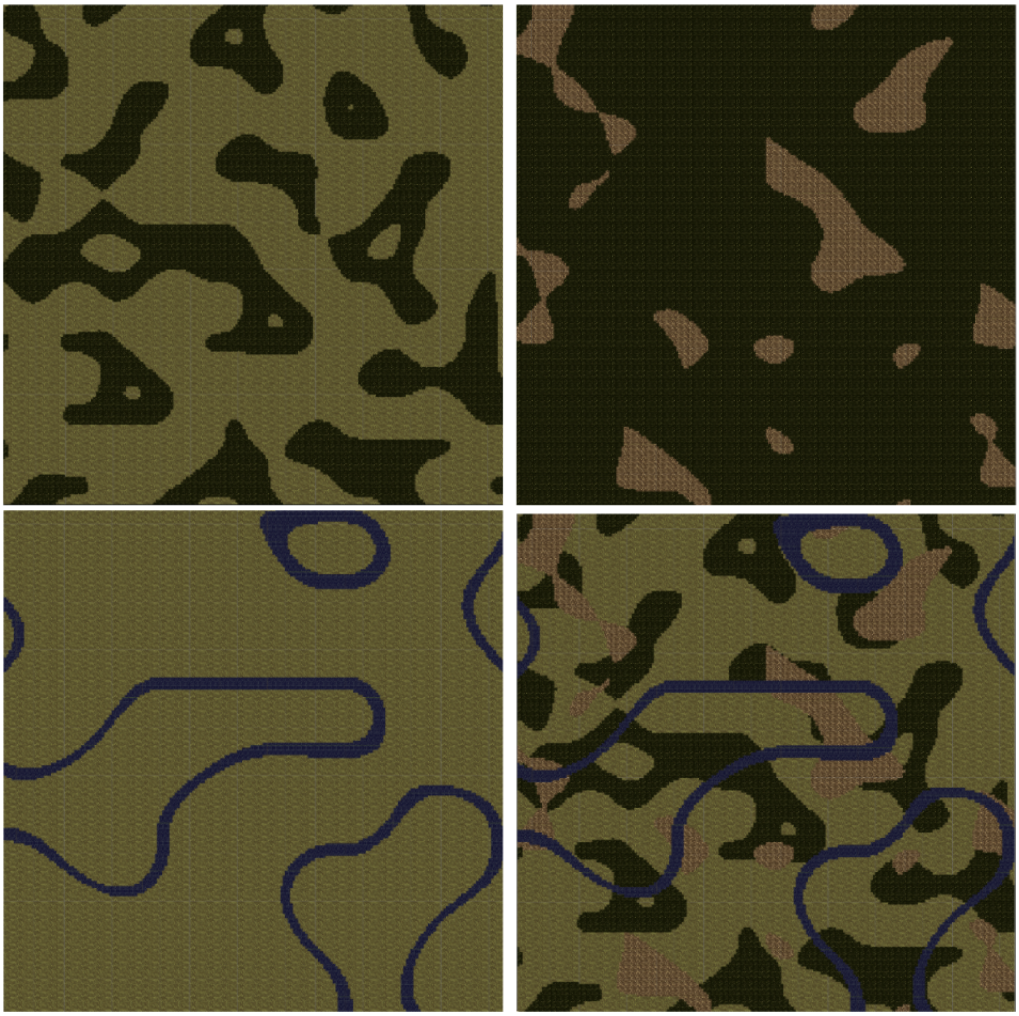

Figure 6: The steps taken to create the Perlin noise map in order are: a) Top-Left: Initial creation of trees (dark green) and grass (light green), b) Top-Right: mountains (brown), c) Bottom-Left: rivers (blue), d) Bottom-Right: final overlay of all components.

2.2.2 Map Boundaries

In order to hide the edge of the map from the player, as well as contain the player and NPCs within the map, a border was added around the entire exterior of the terrain. The border is 15 tiles thick with trees, which is wider than the scope of the game camera, so that the player is unable to see the edge of the map. The tile allocation process is simple: if a tile in the border region belongs to the grass type, change it to a tree. This way, the natural formations of rivers and rocks are not eliminated at the edge of the map. Instead, the rivers appear to flow directly outside the map. Examples of this are shown in figure 12.

2.2.3 Connecting all walkable regions

After the first phase of the map preparation, another step to ensure a higher quality of the generated content is to merge all the walkable areas so that they are accessible from any other location. To achieve a map meeting this condition, we implemented a scan-line region-growing algorithm that classifies each cell as walkable or not and assigns different numbers to each of the separate regions, finding the shortest paths between them, and finally creating the connections in between. In this paragraph we use walkable regions and clusters interchangeably.

After scanning and clustering is complete, the boundary points of each cluster are marked. This step is conducted in order to minimize the number of calculations performed while measuring distances between clusters. This distance is calculated between all pairs of boundary points between different areas. The minimum value is stored in a produced symmetric distance matrix for the row and column corresponding to the considered clusters. Cluster pairs are selected iteratively based on the minimum values retrieved from the matrix. A connection is then forced between the pair.

The assumption is that the resulting graph (with nodes being clusters and edges being the connections to be built) should be connected, which satisfies all walkable points' fundamental assumption to be accessible from anywhere. With this condition, the graph is iteratively built, adding other nodes closest to any already embodied vertices. This results in connections that are optimal and good-looking. After recognizing all the edges, the respective regions' closest points are used to calculate factors of the linear function that goes through them.

Finally, the computed edges are applied on the existing map, ensuring the player's minimum width to be able to walk on them. If the points where the new connection is going to be placed contain water, we apply the bridge terrain type, otherwise, the grass terrain type is used. The final results of this process are visible in the Figure 12.

2.2.4 Positioning the elements of the map

Castle:
The final point of the game is the castle. We implemented a procedure that constructs the building from the used tileset based on the input width. It is built by stacking the proper tiles for left and right walls and constructing the gate in the middle. A loop that fills the front and top of the building between them is run as many times as the width minus the already created columns. We use a randomized algorithm that first requires enough space around it for the player to move to locate the castle. Second, it evaluates better locations closer to the right top corner of the map. We use a loop that samples 20 possible positions (if the generated position does not meet the requirement of the proper space around it, the step is repeated without incrementing the iterator until the correct coordinates are sampled) and selects the best one among them. This ensures better gameplay as the map is always slightly different, but the path to the castle is usually long enough. The castle size is hardcoded in our algorithm, and in the final version of the game, we decided to use the width of 11, which means that there are 6 columns generated in the mentioned loop. The castle can be seen in figure 13.

Aesthetic elements:
Other elements located on the map are twisted trees composed of 3 tiles and that require to be surrounded by grass, and small houses composed of 6 tiles, having the same condition before being placed. The locations are sampled randomly. The number of the generated elements is 20 for the first type and 10 for the second. These values are hardcoded in the algorithm.

Inventory items:
The items that have been positioned on the game are the following:

1. Hearts: Give extra health
2. Swords (normal, wooden, and golden): Give extra damage power
3. Ax: Give extra damage power
4. Boots: Give endurance against attacks
5. Helmet: Also, give endurance against attacks

These items have been positioned randomly through the walkable areas of the maps. However, constraints have been applied to avoid the positioning of two items in the same grid. Essentially, if the random position obtained for the item has been filled by another item, or is within a non-walkable area, then a new position is obtained.

Non-Player Characters:
The NPCs have been positioned randomly in groups of four. The gathering of NPCs in teams augments the game's

difficulty since it is easier to defeat an enemy that is wandering alone instead of having to confront a whole team. In parallel, the groups can patrol roads and important weapons successfully. Finally, the boss of that guard is positioned in the gates of the castle.

It should be mentioned that both NPCs and the inventory items on the map are Prefabs, meaning instances of the specific gameobject. They are instantiated in the map creation procedure at the beginning of the game from a specific gameobject. Thus, the number of NPCs or hearts during the game can be adjusted based on the player's preferences.

2.2.5 Tile Allocation

After the PCG algorithm has assigned a terrain type to every cell in the map, the next step is to assign tiles to the tilemap based on the generated map of terrain types. As described in section 2.1, there are three different tilemaps to assign the tiles: ground tilemap, collision tilemap, and aesthetic tilemap. This section describes the allocation of tiles to the tilemaps using the generated terrain map as input.

The simple approach is to assign a specific tile to each terrain type (e.g., select a grass tile, and assign it to the grass terrain type). Then for each terrain type, also assign a tilemap:

- $Water \rightarrow collision$
- $Grass \rightarrow ground$
- $Tree \rightarrow collision$
- $Cliff/rock \rightarrow collision$
- $Bridge \rightarrow ground$
- $Castle \rightarrow collision$
- $House \rightarrow collision$

Finally, for each cell in the terrain map, assign the tile corresponding to that terrain type to the tilemap assigned to the terrain type. This approach provides a functional map but considers neither 1) the need for the aesthetic layer nor 2) tile transitions, as described in section 2.1.

The first can be solved using a switch-case statement, taking as input the terrain type. Algorithm 1 shows a sample of the case statements used; the full number of case statements is quite large. To summarize the algorithm, each terrain type has a predefined tile and tilemap to place the tile on. If the tile is placed with a transparent background (i.e., the image texture does not fill the entire square), an additional tile is needed to be placed on a lower tilemap. The tilemaps are ordered from lowest to highest as ground, collision, aesthetic, where the lower tilemaps are drawn first to the screen, and the higher tilemaps are drawn on top. Generally, the tiles that we assign that have a transparent background are located on land, so that we may place a grass tile on the ground layer to compensate for the transparency of the tile.

Algorithm 1: Tile Assignment based on Terrain Type

input : TerrainType T, 2D Coordinates (x,y), Tilelist Tiles, Tilemaps tileMapCollision tileMapGround tileMapAesthetic
output: Tile allocation to Tilemaps

```
1  switch(T)
2    case Terrain.WATER::
3      tileMapCollision(x, y) = Tiles(WATER)
4    case Terrain.GRASS:
5      tileMapGround(x, y) = Tiles(GRASS)
6    case Terrain.TREE:
7      tileMapGround(x, y) = Tiles(GRASS)
8      tileMapCollision(x, y) = Tiles(TREE)
9    case Terrain.ROCK:
10     tileMapCollision(x, y) = Tiles(ROCK)
11   case Terrain.BRIDGE:
12     tileMapGround(x, y) = Tiles(BRIDGE)
```

To solve the second issue regarding tile transitions, a set of rules was added for the tile selection. For example, rather than each location consisting of the tree terrain type having the same tile, a set of tiles were used representing the outside edges of the forest. Thus, when a group of tree tiles is clustered together, the trees' exterior contains transition tiles to provide a more realistic appearance. The transition tiles were included only for the tree tiles, resulting in smooth transitions from grass-tree. Figure 7 shows the rules used to determine whether the tile should be an edge tile or not and which edge tile (top, bottom, left, right, top-right, etc.).

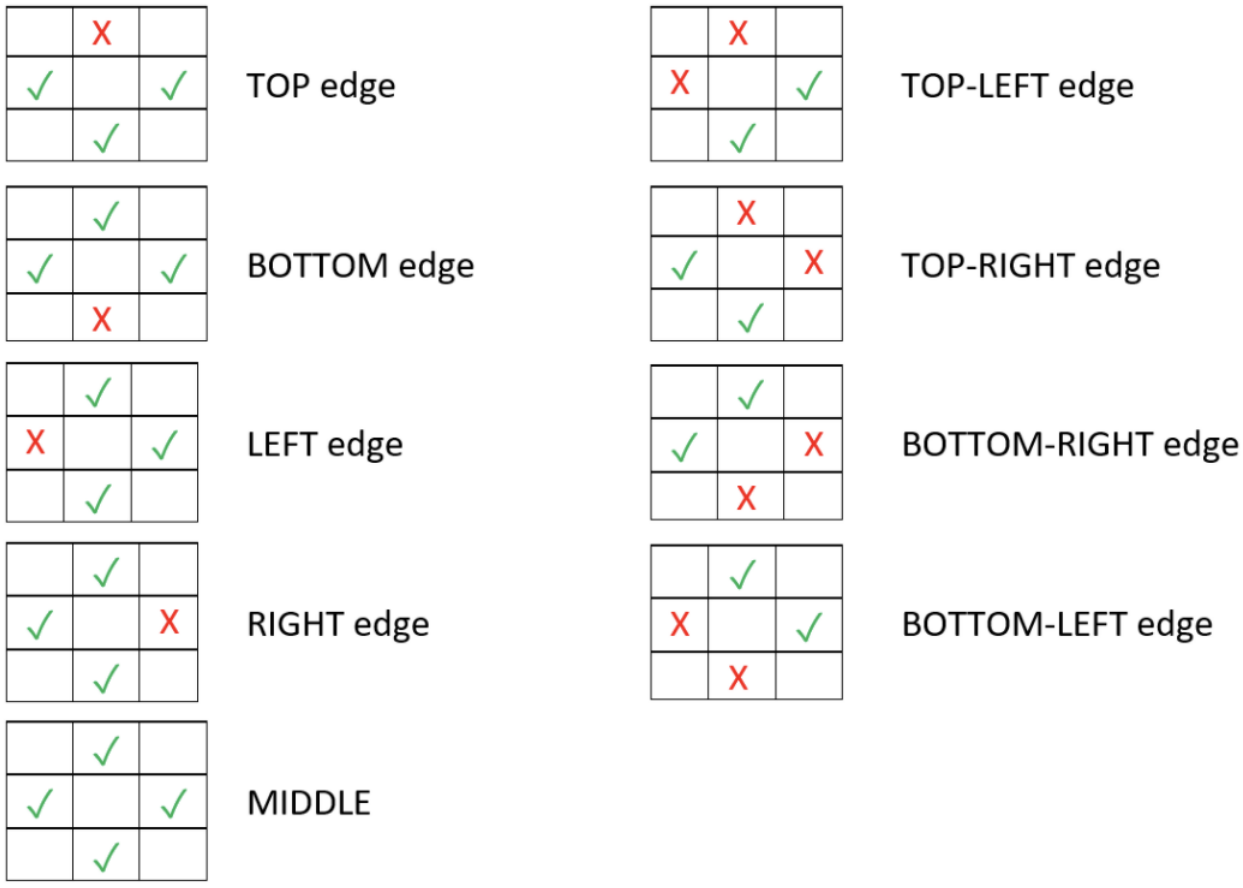

Figure 7: Rules for creating transition tiles. There are nine rules for eight different edge tiles, as well as one non-edge tile. Each grid represents the local neighborhood around a query tile. A red X means that the cell in the X's direction does not share a terrain type with the query tile. A green checkmark represents that the tile in the checkmark direction does share a terrain type with the query tile. Cells that do not contain any marks are ignored in the condition. The satisfied rule determines the tile to place on the tilemap.

2.2.6 Levels of Difficulty

Before the game begins, the player can change the difficulty level from the easy one selected by default to normal or hard. The factor values listed in Table 1 apply for calculating the max health of the player (the original value of 500 divided by the factor), the health of the NPCs (the original values are multiplied by the factor), and damage (also multiplication). The final values were selected empirically by assessing their impact on the gameplay. Using higher values than the ones in the hard level, resulted in the player being killed too easily. Thus, this scenario was aborted since it was too challenging.

	Factor
Easy	1.0
Normal	1.1
Hard	1.2

Table 1: Difficulty factor on each level

3 Controlling the Non-Player Characters (NPC)

3.1 Finite State Machines

A finite state machine (FSM) is a mathematical abstraction used to design algorithms. The state machines can change from one state to another in response to some inputs. In our case, they are deterministic since, given an input, they go to a state deterministically. In our case, following the design patterns of abstract classes, the script "Character" controls the states, and the script "States" is an abstract class that the actual states of the NPCs are following.

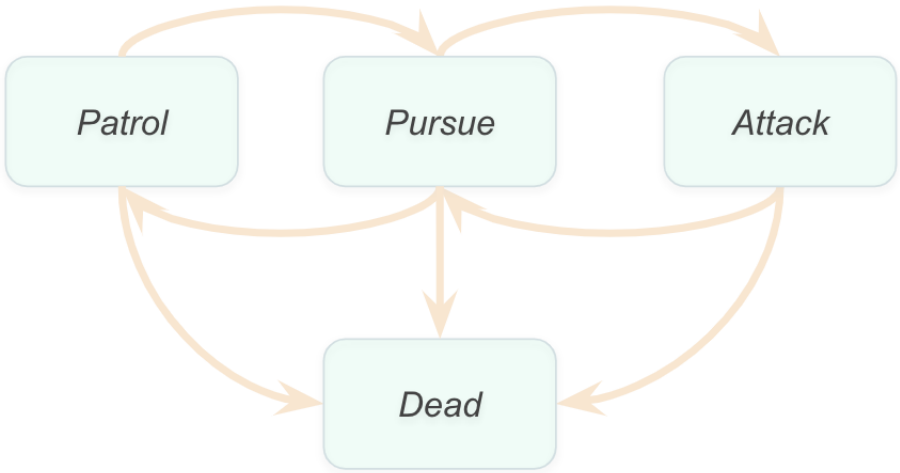

Figure 8: States of the NPC with their transitional conditions.

In figure 8, the FSM consists of the following states:

1. **Patrol State**: The NPCs take small steps from an initial point to a specific point near the patrol area. This state's main routine calculates the distance between the player and the NPC and the enemy's navigation, that will be analyzed in the following section.

2. **Pursue State**: The NPCs start following the player when they are aware of his presence. In this state, the distance between the player and the enemy is again calculated since it holds the condition for the enemy's transition back to the patrol state. Besides this control, the enemy has to follow the player in order to attack him. This procedure is also analyzed in the next section.

3. **Attack State**: When the distance between player and NPC is close, the battle begins. The speed of the hits is determined by the constant variable hit time. A public function in the player's object is invoked in order for the hit to be applied and subsequently alter the player's health. The damage that the player can cause, such as the enemies' initial health, depends on the game's difficulty.

4. **Dead State**: The NPC enters this state when his health is less or equal to zero. The NPCs are instantiations, and their pointers are passed to the Dead State function to disappear the enemy from the scene after its death. In case that the NPC is the Boss of the game, a subroutine is invoked for the calculation of the player's position. If the position of the player is inside the castle, then the game is finished.

The transition between these states as well as the allowed transitions are illustrated in figure 9. In parallel to the states' update, the NPCs also update their animation to demonstrate a realistic behavior. The animation controller handles the behavior transitions, meaning the states Idle, Run, Combat Idle, Attack, and Death, defined with the white arrows. Also, the controller holds extra parameters, and their initialization leads to the activation of the animation states.

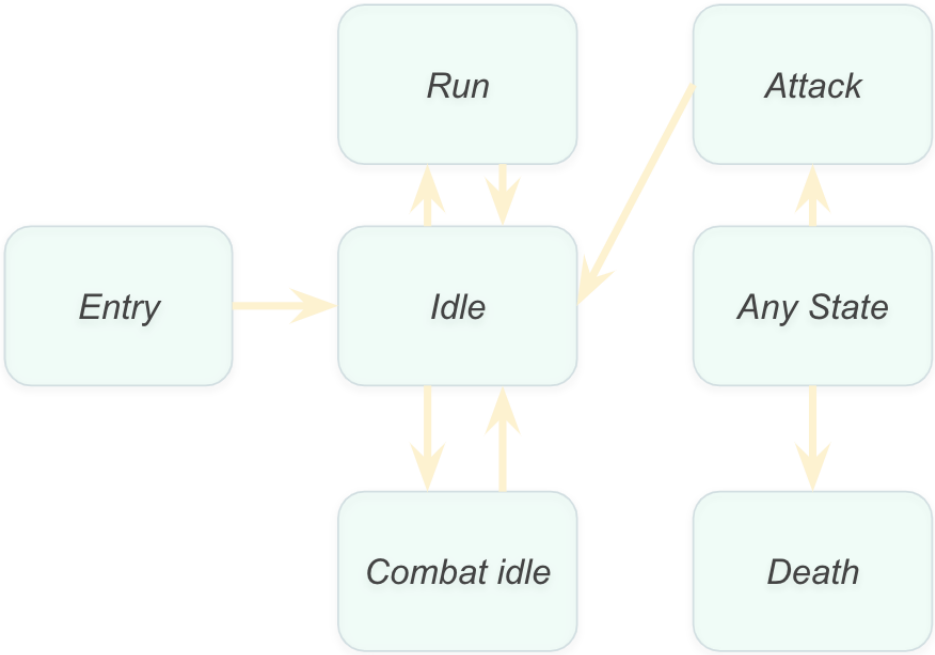

Figure 9: Animation controller of NPCs.

In our case, the enemy sprite in different poses is depicted in figure 10. In the transition of the FSM from the pursuit state to the attack state, the parameter that holds the condition for the transition of the run animation state to the attack animation state is activated. Hence, the game has illustrated a transition from the third to the second sprite, which resembles a real attack movement.

Figure 10: Enemy's animation

3.2 Navigation of the NPCs

The NPCs have to be moved both in the patrol state and in the pursuit state. First, they choose one random point in a radius and traverse to it, while in the latter move towards the player's position. However, the path that they will traverse should be specifically given and, at the same time, aware of the collision blocks.

An initial idea for a pathfinding algorithm was a navigation mesh object. However, the integration of such a component with a dynamically created 2-dimensional terrain was not feasible. Thus, the A-star algorithm was employed.
A star is a graph traversal and path search algorithm published by Peter Hart, Nils Nilsson, and Bertram Raphael, which stands out for completeness, optimality, and optimal efficiency.[1] However, one major practical drawback is its space complexity ($O(b^d)$), as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally outperformed by algorithms that can pre-process the graph to attain better performance. [5]

A star is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance traveled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

In our game, the first step for applying the A star algorithm was creating the graph out of the TimeMaps. The graph nodes correspond to the walkable tiles, which are mentioned in section 2.2.5, and if two tiles are neighbors, then an edge is added to the graph. Given this graph, the A-star algorithm could be successfully executed.

However, the NPC could not directly move at tiles in the map since such behavior would be sharp and unnatural. The strategy that was performed is the following. In the case of the NPCs' patrol state, the A-star has as inputs their current position and the random walkable point in a circle and produces an array of walkable grid points that show the desirable path. This array was transformed into a list that popped a new grid point after the previous was reached out. Thus, the transition of one grid point to the next performed smoothly in a time depended on the speed of the character. Correspondingly, in the pursuit state, the A-star has inputs the NPC's current position and the player's position, and the movement follows the same procedure.

4 Game Design

4.1 Scene Manager

The game consists of four different scenes:

1. MainMenu: the initial scene that is depicted on the initialization of the game
2. Victory: the scene that is shown when the player wins the game
3. GameOver: the scene that is shown when the player loses the game
4. GameScene: the map procedurally generated along with the components descriptive in section 2.1.

At the beginning of the game, the MainMenu is illustrated. This scene includes a canvas on which a text gameobject with our game title is depicted, along with the buttons of selecting the game's difficulty and the button of the selection of the beginning of the game. On the Game Start button, the onClick function activates the MainMenu script, which loads the GameScene. In this scene, the map is generated. Also, this scene holds the logic and the whole functionality of our game. In the first end-game condition, meaning the winning, the scene Victory is loaded, which only contains a button that returns us in the MainMenu scene. On the second end-game condition, the scene GameOver is loaded, which contains the same button as before returning us in the MainMenu scene.

4.2 Player Characteristics

The player has 8 total available actions that are listed below:

1. Move right: right key arrow
2. Move left: left key arrow
3. Move up: up key arrow

4. Move down: down key arrow
5. Stay idle: do not press any key
6. Attack: space key
7. Retrieve weapon: W key
8. Leave weapon: Q key

The animation controller of the player is depicted in figure 11.

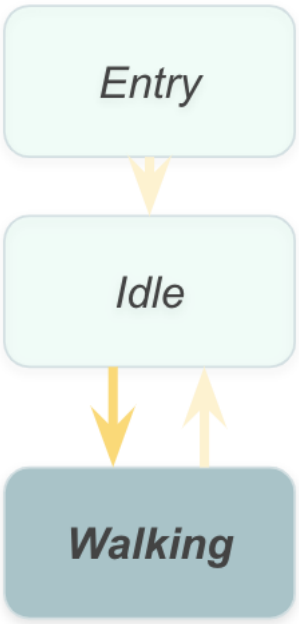

Figure 11: Player's animation controller

The transition from an idle position sprite to the walking position sprite creates an illusion of walking.
In the creation of the player gameobject, the following properties are initializing:

- Health: The player's life is augmented with the hearts' consumption and is diminishing with the NPC's hits.
- Damage: the amount of health that a hit can cause to the NPCs. The ax, the knife, and the different swords augment the damage.
- Endurance: the amount of hit damage that the player can endure. After every hit, the endurance amount diminishes the hit strength of the opponent. Weapons such as boots, shield, and helmet augment endurance.

It should be mentioned that the player can not hold more than two weapons in his hands. Thus, if he already has the knife and a sword, he can not obtain a second sword. For this reason, the action "Q" (leave the weapon) is important. For instance, if the player holds two weapons in his hands already, but the golden sword is visible (which causes more damage than the other weapons), he could leave one of his weapons to retrieve the golden sword. This strategy will augment his damage towards his opponents. This condition does not apply to the helmet or the boots since the player does not have to hold them in his hands.

4.3 NPC Characteristics

The NPCs have similar characteristics to the player. They are instantiated with a specific amount of health and damage (which varies based on the game's difficulty). However, the NPCs can not retrieve the weapons nor consume the hearts.

The size of the figure of the NPCs is similar to the size of the player. However, the boss that guards the castle is highly scaled.

5 Experimental Set-up

5.1 Parameters of the grid map

Our final grid map was a square with a width equal to 200. Whereas larger grid sizes were attempted, our computers' computational power resulted in lags that made the game unpleasant.

5.2 Parameters of the player

In table 2, the parameters of the player are depicted:

Parameter	**Value**
Max Health	500
Damage	20
Endurance	0
Attack Time	0.5s

Table 2: Player's Parameters

5.3 Parameters of the NPC

In table 3, the parameters of the enemies are depicted:

Parameter	**Value**
Max Health	100
Damage	10
Endurance	0
Attack Time	1.0s

Table 3: NPC's Parameters

In table 4, the parameters of the boss are depicted:

Parameter	**Value**
Max Health	200
Damage	30
Endurance	0
Attack Time	1.0s

Table 4: Boss's Parameters

5.4 Parameters of the static elements

In table 5, the extra elements on the map and the amount of health, damage, and endurance are depicted.

Elements	Health	Damage	Endurance
Hearts	20	0	0
Shield	0	0	5
Knife	0	5	0
Sword	0	20	0
Wooden Sword	0	5	0
Golden Sword	0	20	0
Boots	0	0	2

Table 5: Static element's Parameters

6 Simulation Results

6.1 Game Snapshots

In figure 12, four different map generations are illustrated. They follow the same patterns, such as the water, the forest, and the cliff formations. The borders consist of lines of trees in order for the player to not see the end of the map.

The castle with the boss guarding it is shown in figure 13. The boss character is highly scaled in comparison to the simple soldiers. In the figures 14, some weapons that are scattered on the game are depicted.

Figure 13: Castle with the guard Boss

Figure 14: Illustration of weapon items

Additionally, in figure 15, there are some snapshots of moments of the battle. In the first figure, the battle has not yet begun; thus, both the characters have full lives, which is verified from the health bar on top of each player. In the second figure, both the player and the NPC have lost a significant amount of life. Then, another snapshot is depicted, where multiple enemies pursue the player. Finally, the last snapshot shows the player moving towards the hearts since he is heavily injured (low health).

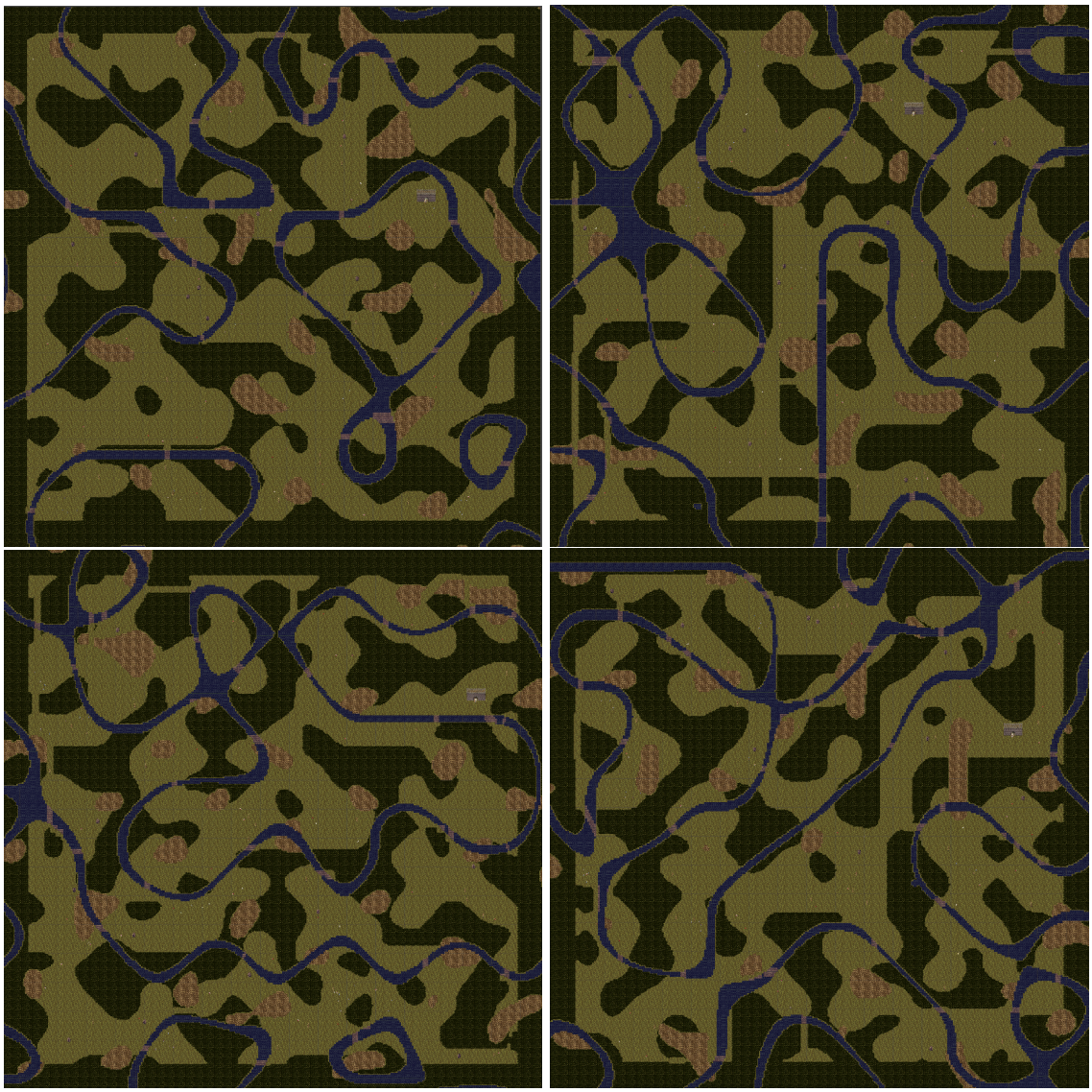

Figure 12: Map Generation

Figure 15: Snapshots during Battle situation

7 Conclusions

In this work, we were able to successfully create a functional action-adventure game. The two main features of our game are a top-down style 2D tilemap, generated using PCG at run-time, as well as non-player characters that serve as enemy combatants against the player.

During the process, we had the opportunity to experiment with Unity platform and to learn how it could be used to facilitate the game development process. Creating a game from scratch in any programming language would take a lot of time and while using Unity, the whole process is boosted. Although we appreciate the advantages characterizing the platform, there is one downside that caused some problems. Comparing to standard and commonly used version control systems, for instance Git, in Unity it is not possible to see the exact changes made in the code in each commit. If one wants to find which lines were altered, two files need to be copied to an external tool for the changes to be highlighted.

The A-star algorithm proved a successful alternative to the NavMesh Surfaces. The functionality of the NavMesh agents in the Unity engine has a limitation when it refers to 2-dimensional terrains. Whereas recently, a new 2D NavMesh component has been constructed, it is not yet functional for our procedural content generation project since the collisions and path-finding should be discovered in the runtime after the map's generation.

An interesting observation was that the player's dynamic rigid bodies and the NPCs were creating intense physical reactions. Especially after their collision, the NPCs were catapulted a great distance in the opposite direction. The simulation of the real-world dynamics by the unity machine was astonishing.

Furthermore, we observed that the increase in the map size results in time lags that disrupt the gameplay. This problem's probable reason is the A star algorithm, which has huge space complexity, as we analyzed in section 3.2. Some significant improvements can be made to improve its performance, for instance storing as much intermediate data of the computed graph as possible, and not repeating the same calculations many times. Another improvement could be to compute graphs only for the surrounding areas. In the case of our game, the final size of the map was perfectly enough, if not even too large. In this situation, it was not a burning issue.

In our future work, we plan to explore the Unity system's design patterns further since we realized how computationally heavy a simple play could become. Finally, we intend to approach not only aesthetic tricks that would improve the game's content but also algorithmic AI approaches that will enhance the substantial quality of the game, such as reinforcement learning agents that will cooperate in order to kill the player in a more advanced manner (such as patrolling the weapons of power, gathering in larger teams and isolating the player).

References

[1] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[2] Zabin. Castle tiles for rpg's, Feb 2018.

[3] Hyptosis and Zabin. Lots of free 2d tiles and sprites by hyptosis, Mar 2018.

[4] Ken. Perlin. Making noise, Oct 2007. noisemachine.com.

[5] Wikipedia contributors. A* search algorithm — Wikipedia, the free encyclopedia, 2021.