



## **Research & Development Project**

### **3D Dungeon Generator**

Hogeschool van Amsterdam  
Gameplay Engineering

Michael Spiegel  
3rd November 2022

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Research</b>	<b>4</b>
2.1	Existing algorithms for dungeon generation . . . . .	4
2.1.1	Binary Space Partitioning . . . . .	4
2.1.2	Cellular Automata . . . . .	5
2.1.3	Delaunay Triangulation . . . . .	5
2.2	Comparison of the algorithms . . . . .	6
<b>3</b>	<b>Implementation</b>	<b>8</b>
3.1	Dungeon Generator . . . . .	8
3.1.1	Creating floor and roof . . . . .	8
3.1.2	Determining Start and end room . . . . .	10
3.1.3	Creating walls . . . . .	11
3.1.4	Lighting . . . . .	14
3.1.5	Combining meshes . . . . .	15
3.1.6	Placing scenery . . . . .	15
3.2	Combat system . . . . .	16
3.2.1	Player . . . . .	16
3.2.2	Enemy . . . . .	16
3.3	Core game mechanic: Finding the key . . . . .	17
<b>4</b>	<b>Conclusion and Outlook</b>	<b>18</b>
4.1	Conclusion . . . . .	18
4.2	Outlook . . . . .	18
	<b>List of Figures</b>	<b>19</b>
	<b>List of Listings</b>	<b>20</b>
	<b>Bibliography</b>	<b>21</b>

# 1 Introduction

For my research project I chose a 3D dungeon generator that should generate a new dungeon every time the game is started. Furthermore, I decided to make it possible to explore the dungeon. For this I want to add a player that will be placed in a randomly chosen start room. To make the whole dungeon more interesting and not leave it as an empty place, I plan to add some scenery like torches and chests. Additionally the dungeon should also contain enemies for which I try to implement a combat system. The focus lies mainly on the visuals of the dungeon but the performance should also be good enough so that it is playable. As a combat system can take much time to implement and needs a lot of testing it could be possible throughout the project that I will only implement a prototype and do not keep it in the final game. Through this project I want to understand how the algorithms work that are used for dungeon generation. Furthermore, I want to get knowledge about combat systems in games.

The whole project can be found on [GitHub](#).

## 2 Research

### 2.1 Existing algorithms for dungeon generation

There exist several algorithms which can be used to create dungeons. In this chapter three algorithms will be described and in the end compared.

#### 2.1.1 Binary Space Partitioning

The binary space partitioning algorithm is very popular for creating dungeons because it is a fast algorithm and not that hard to implement. With this algorithm the dungeon is represented through a binary tree structure. The initial space will be split in each iteration of the algorithm. With this algorithm it is possible to create a variety of dungeons each time the code is run. For this it is necessary that specific parts of the split have to be randomised. The split can be horizontal or vertical. Figure 2.1 shows how a dungeon generated with Binary Space Partitioning looks like. (Williams, n.d.)

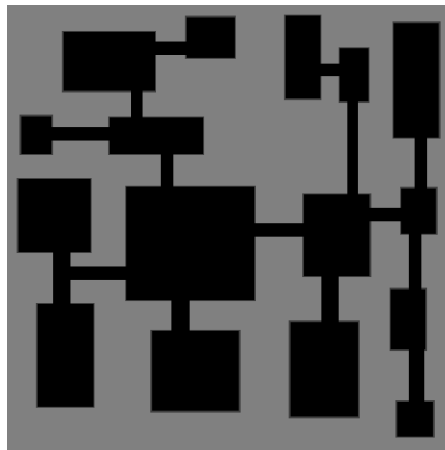


Figure 2.1: Example of a BSP generated dungeon  
Source: Williams, n.d.

### 2.1.2 Cellular Automata

The Cellular automata algorithm uses a grid of cells to build a dungeon. Each cell has a reference to its neighbour cells and a defined state for the time  $t = 0$ . With the help of predefined rules the state for  $t = t + 1$  can be calculated. Depending on the defined rules and the cell states there will be patterns that occur from time to time. A cell itself represents a space where the player can walk but also a space where the player can not walk for example rocks. As you can see in figure 2.2 Cellular Automata creates more cave like dungeons without real rooms. (Williams, n.d.)

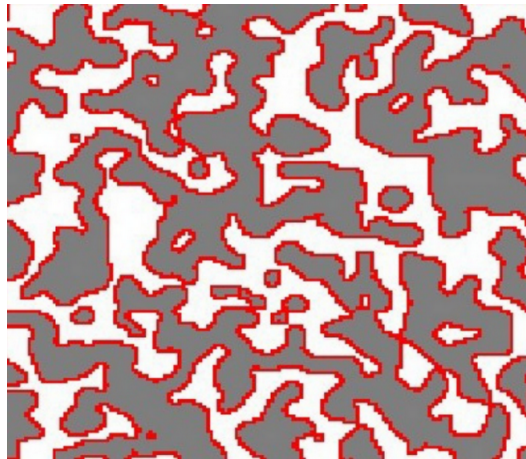


Figure 2.2: Example of a Cellular Automata generated dungeon  
Source: Williams, n.d.

### 2.1.3 Delaunay Triangulation

The last algorithm is the delaunay triangulation algorithm for dungeon generation. This algorithm also uses cells but in this case a random rectangle is created inside of each cell. After this step the algorithm separates the room and prevents them from overlapping. All cells that are exceeding a threshold become rooms. To connect all rooms a graph for all the room centre points is constructed. Additionally to the delaunay triangulation algorithm it is necessary to generate a minimum spanning tree of the originally create graph to remove cycles. In figure 2.3 you can see how a dungeon generated with the delaunay triangulation looks like. Compared to the other two algorithms it is more similar to the result of the binary space partition generated dungeon because both use rooms. (Williams, n.d.)

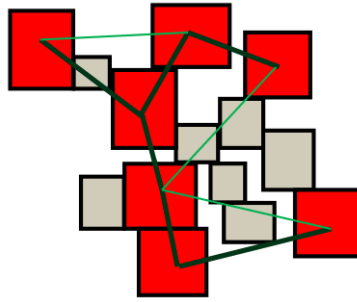


Figure 2.3: Example of a Delaunay Triangulation generated dungeon  
Source: Williams, n.d.

## 2.2 Comparison of the algorithms

Now let's compare those algorithms to explain which one fits best for the purpose of this project. The cellular automata algorithm generates more cave like dungeons which is not applicable to the goal of this project. The dungeon should consist of rooms which only the binary space partition and the delaunay triangulation algorithm provides.

To compare those two algorithms let's have a look on the performance. Figure 2.4 shows the performance of the binary space partitioning algorithm. This algorithm is slow when less rooms are used for the dungeon but has its strength in the generation of more complex dungeons that consists of a huge amount of rooms.

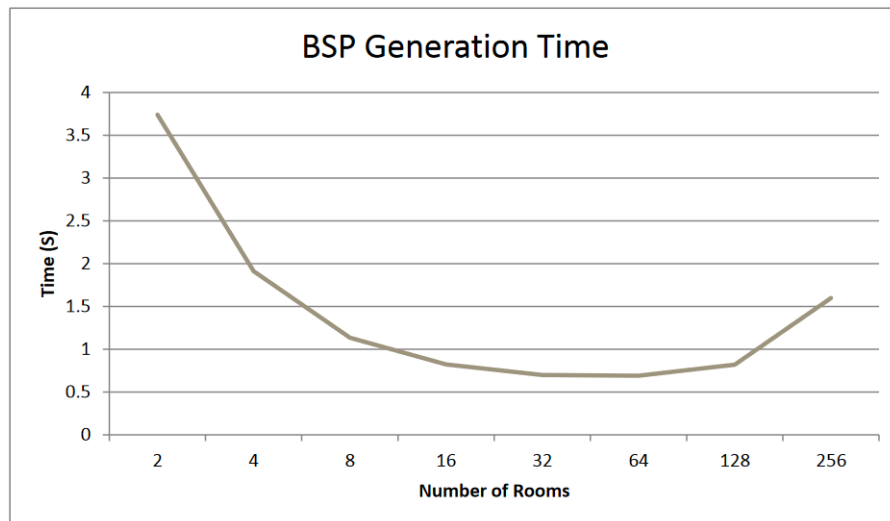


Figure 2.4: Performance of the Binary Space Partitioning algorithm  
Source: Williams, n.d.

Figure 2.5 shows that compared to the other algorithm the delaunay triangulation algorithm is faster when it comes to dungeons with less rooms. On the other side this algorithm gets slower with the increasing amount of rooms. This means, that this algorithm fits best if the dungeon will only consist of a few rooms.

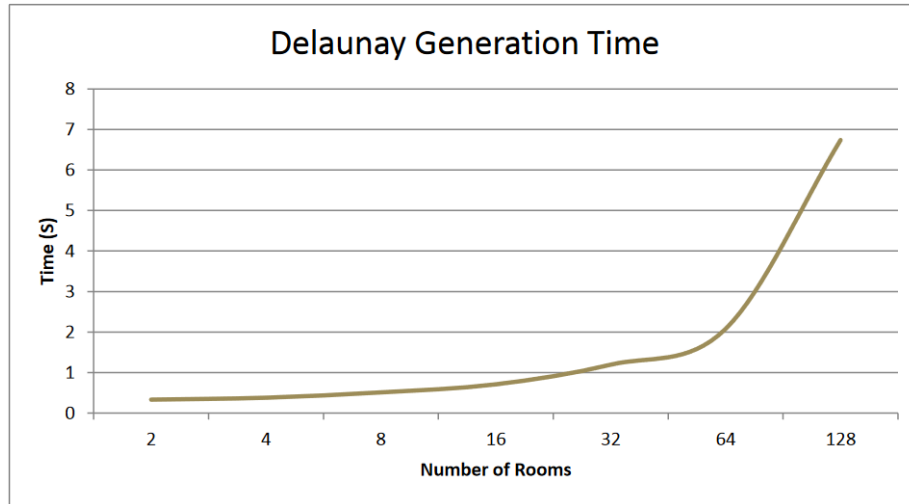


Figure 2.5: Performance of the Delaunay Triangulation algorithm  
Source: Williams, n.d.

## 3 Implementation

In the following sections it is shown how the generation of the dungeon has been implemented.

### 3.1 Dungeon Generator

#### 3.1.1 Creating floor and roof

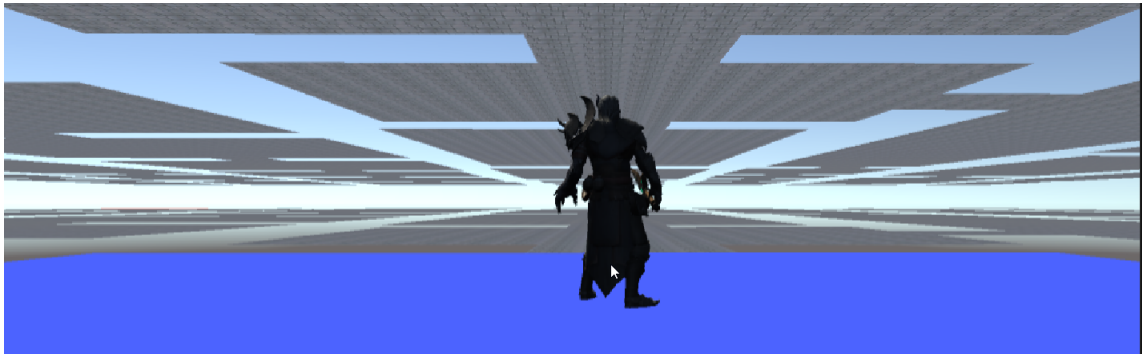


Figure 3.1: Floor and roof of the dungeon

To create the floors the BSP algorithm is used. Before creating the meshes we need to collect all the nodes and put them in the tree structure. For this, the whole space is at first divided into two spaces and then those two are divided into another two. Each space is a node and saved in the tree structure. To decide when the space has to be divided I added a minimum and maximum for the length and width of each room. To represent the tree structure I used a queue and a list for the rooms. When a node is not between the minimum and maximum values that space will be split and the two new nodes are added to the queue and list. The orientation of the room is selected randomly or based on the length and width of the room. So not all rooms are horizontal or vertical.

After collecting all the nodes, we have to find all the lowest leafs because those have to be drawn as rooms. To find them the tree is traversed until the lowest leaves are



reached. Each room node has two corner points and the other two can be calculated with the help of the existing ones.

To finally create a mesh for each room we have to loop through all the room nodes. The same nodes can also be used for the roof because it is the same with a different height. As you can see the creation of the triangles is also different. As each room is a rectangle with four corners we can draw two triangles to get a rectangle mesh.

To create the corridors each node is connected with its neighbour node. The corridors nodes are handled like room nodes and therefore also stored in the same list that we use to iterate over. The mesh creation is also the same.

```
1  // Place triangles inverted if it is not the floor
2  int[] triangles = new int[6];
3  if (isFloor)
4  {
5      triangles[0] = 0;
6      triangles[1] = 1;
7      triangles[2] = 2;
8      triangles[3] = 2;
9      triangles[4] = 1;
10     triangles[5] = 3;
11 }
12 else
13 {
14     triangles[0] = 2;
15     triangles[1] = 1;
16     triangles[2] = 0;
17     triangles[3] = 3;
18     triangles[4] = 1;
19     triangles[5] = 2;
20 }
```

Listing 1: Creation of floor and roof

### 3.1.2 Determining Start and end room

To find the start room it is not much work necessary. I only check the index while iterating over the list of rooms and choose the first created room as start room.

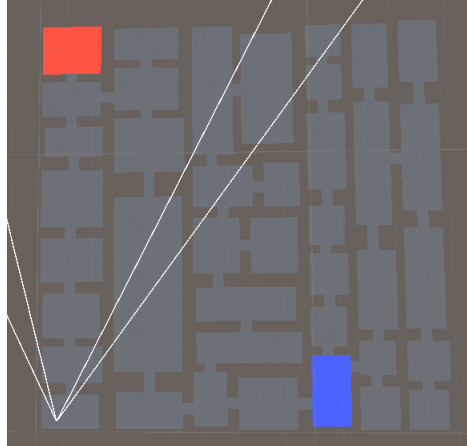


Figure 3.2: Start room (blue) and end room (red)

For the end room I thought about which rooms could be end rooms and I think the best case is when the end room is not too near to the start room. For this I calculate the distance between each room and the start room. The room that is the furthest away will be the end room.

```
1  ...
2  foreach (var (room, index) in listOfRooms.Select((room, index) => ( room, index )))
3  {
4      ...
5      // Check distance between start room and each other room
6      // to find the room which is the most distant from the start room
7      if (index > 0 && index < listOfRooms.Count / 2)
8      {
9          RoomNode currentRoom = (RoomNode)listOfRooms[index];
10         float dist = Vector3.Distance(startRoom.CentrePoint, currentRoom.CentrePoint);
11
12         if (!(maxDistance < dist)) continue;
13
14         maxDistance = dist;
15         endRoom = _dungeonFloors[index];
16     }
17 }
18 ...
```

Listing 2: Determining end room

### 3.1.3 Creating walls

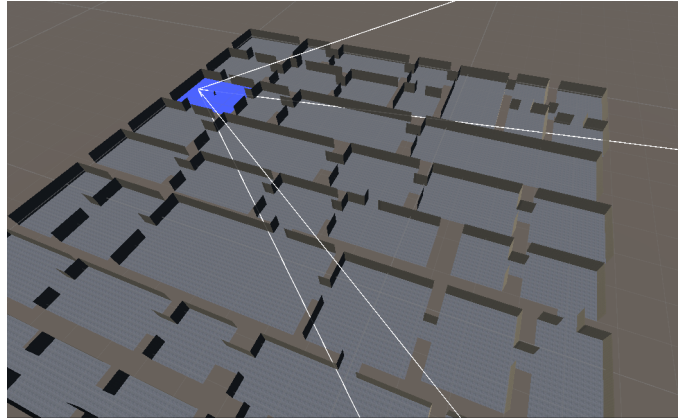


Figure 3.3: Walls of the dungeon

The creation of the rooms got a bit complicated because of the corridors. For each side of the room it is checked if the corners of the corridor are between the corners of that side of the room. If that is the case, the wall has to be divided into two walls. The corners of the corridor act then once as end point and once as start point for the two walls. If there is no corridor the wall can be created for the whole length of that room side. The corridors only get walls on two sides so they are open to walk through. In figure 3.4 you can see that the two walls will be created from roomStart to corridorStart and from corridorEnd to roomEnd.

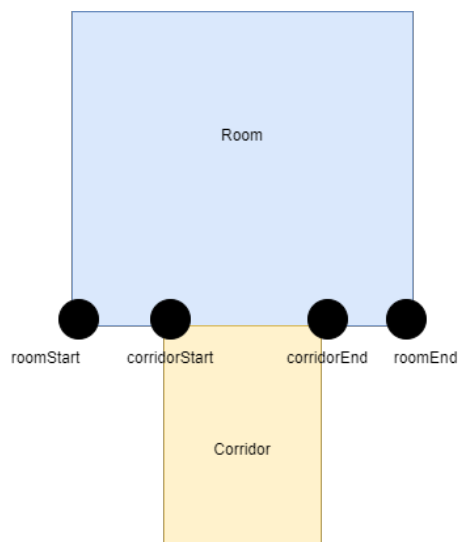


Figure 3.4: Detecting corridors between room corners

There is also a special case for creating the walls when the corridor is not placed between the room corners. Figure 3.5 shows how it looks like when the corridors are placed the the corner of the room. Depending on the room corner the walls have to be created from corridorStart to roomStart and from corridorEnd to roomEnd or from roomStart to corridorStart and from roomEnd to corridorEnd.

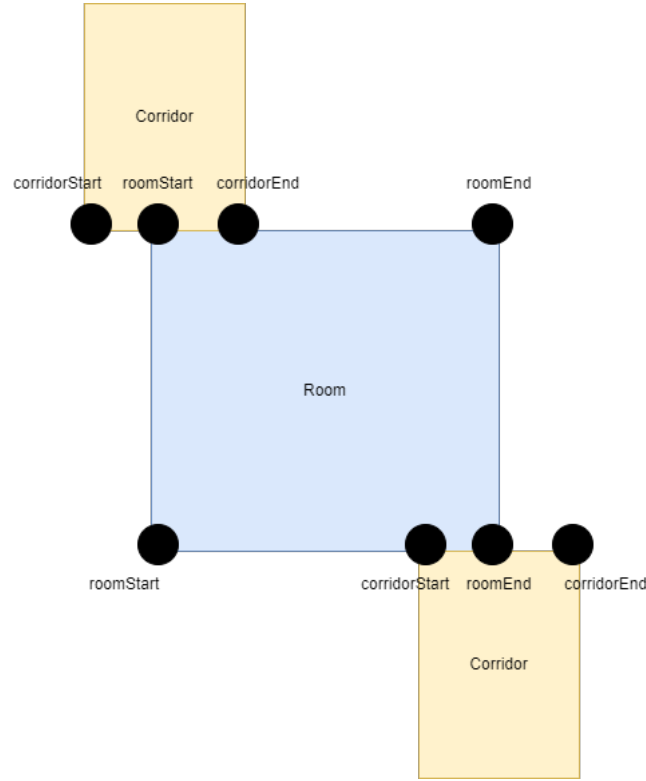


Figure 3.5: Detecting corridors at room corners

Before creating the mesh it is necessary to get the positions of the vertices. This works like the creation of a plane but instead of creating it flat, it has to be created into the y direction. Depending on the direction of the wall it will be created into the x and y direction or the z and y direction. Horizontal walls use the x direction and vertical walls the z direction.

Creating the mesh is very simple. The only difference compared to a plane in this case is, that I have to check if it is a horizontal or vertical wall for the calculation of the length. It is also important to flip the creation of the vertices. All walls have to be visible from the inside of the rooms and therefore the triangles can not be created the same way for all walls. Some walls need the triangles to be drawn clock wise and some

counter clockwise.

```
1  if (!isFlip)
2  {
3      for (int y = 0; y < dungeonHeight; y++)
4      {
5          for (int x = 0; x < length; x++)
6          {
7              triangles[tris] = vert;
8              triangles[tris + 1] = triangles[tris + 4] = vert + length + 1;
9              triangles[tris + 2] = triangles[tris + 3] = vert + 1;
10             triangles[tris + 5] = vert + length + 2;
11
12             vert++;
13             tris += 6;
14         }
15
16         vert++;
17     }
18 }
19 else
20 {
21     for (int y = 0; y < dungeonHeight; y++)
22     {
23         for (int x = 0; x < length; x++)
24         {
25             triangles[tris] = vert;
26             triangles[tris + 1] = triangles[tris + 4] = vert + 1;
27             triangles[tris + 2] = triangles[tris + 3] = vert + length + 1;
28             triangles[tris + 5] = vert + length + 2;
29
30             vert++;
31             tris += 6;
32         }
33
34         vert++;
35     }
36 }
37 ...
```

Listing 3: Creating wall mesh

### 3.1.4 Lighting

To get the positions for the lights the length of the wall is calculated and divided by the frequency of the lights. With the help of steps it will be decided if that position is used for the lights. For corridors the wall positions are ignored because with this approach it would always place the maximum amount of lights per wall. To prevent that the open space of the corridor is used to place lights, only that sides of it are used for collection the positions. As positions can occur multiple times because of overlapping between walls,

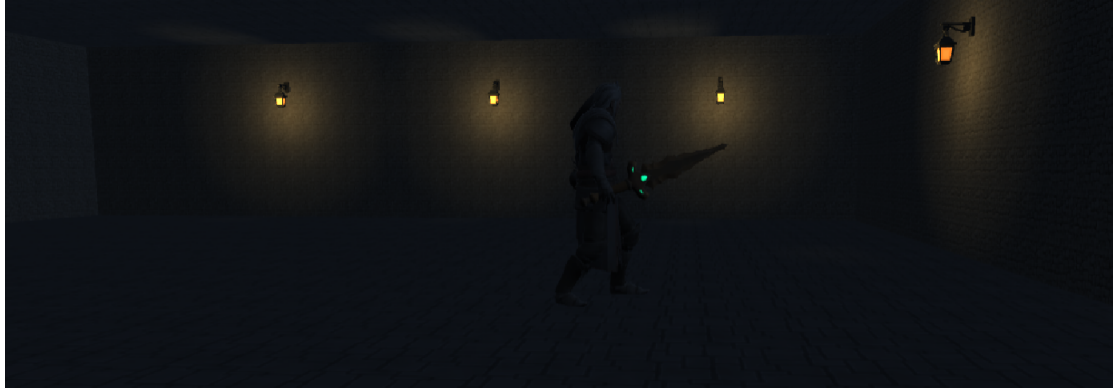


Figure 3.6: Lights in the dungeon

It is checked if a point is already in the list for the possible positions. If that is the case the position will be removed and added to another list to ignore that positions in the future. To keep track over all points I used two lists, one for the possible positions and one list for the actual positions. For the actual positions it depends if `isGettingLight` is true. Additionally to the position the rotation is added so the light prefabs are all facing inside the room. To store this data I used a Dictionary.

```

1 private void SaveLightPosition(Vector3 position, int rotation, bool isGettingLight)
2 {
3     Vector3Int point = Vector3Int.CeilToInt(position);
4
5     if (_possibleLightPositions.Contains(point))
6     {
7         _possibleLightPositions.Remove(point);
8         _ignoreLightPositions.Add(point);
9
10        if (_lightPositions.ContainsKey(point))
11        {
12            _lightPositions.Remove(point);
13        }
14    }
15    else if (!_ignoreLightPositions.Contains(point))
16    {
17        _possibleLightPositions.Add(point);
18        if (isGettingLight)
19        {
20            _lightPositions.Add(point, rotation);
21        }
22    }
23 }

```

Listing 4: Saving light positions

### 3.1.5 Combining meshes

This method is from the Unity documentation. It is slightly modified and uses a parent as input parameter and a boolean to check if it is a wall or a floor/roof.

Depending on the boolean it assigns the appropriate material and layer. The layer is needed for the third person camera of the player. Furthermore the mesh collider is added after combining the meshes.

### 3.1.6 Placing scenery

To place the scenery I iterate over all rooms except the start room. To prevent that multiple sceneries are placed at the same position, the position is added to a list. While placing the scenery it checks if the position is already in use and generates a new position if this is the case. Furthermore all the sceneries will be rotated towards the centre point of the room.

```

1 var direction = (room.CentrePoint - transform.position).normalized;
2 instantiatedScenery.transform.rotation = Quaternion.LookRotation(direction);

```

Listing 5: Rotating scenery

## 3.2 Combat system

I implemented a prototype for the combat system and asked my classmates and friends if they can test it and give me feedback. As the combat system was not matching my criteria to prevent the player from running through the dungeon and ignoring the visuals I decided to not put further work in it. The feedback was also not that good because a combat system needs much work and the focus during my project is not about the combat system. Nevertheless I want to show how I implemented my prototype.

### 3.2.1 Player

The player can start to attack with clicking the left mouse button. If the player attacks an enemy it will call a method of the script health that is attached to the enemy. The enemy will then start attacking the player. It is also possible that the enemy attacks the player without being attacked by the player before.

```
1  ...
2  // Attack
3  if (Input.GetKeyDown(KeyCode.Mouse0) && !_isMoving)
4  {
5      _isAttacking = true;
6      animator.SetBool(IsAttacking, true);
7  }
8  ...
```

Listing 6: Player attack

### 3.2.2 Enemy

The enemy has three states:

- Patrolling
- Chasing
- Attacking

In the first state, the enemy is walking around to a random destination point within the navigation mesh. The enemy enters the second state when the player is in a specific range and starts running towards the player. After the enemy reached the player the attacking state is activated and the enemy starts attacking.



```

1  ...
2  // Switch to appropriate state
3  if (!isPlayerInSightRange && !isPlayerInAttackRange && !_isAttacked && !_isDying)
4  {
5      // Patrolling
6      animator.SetBool(IsAttacking, false);
7      animator.SetBool(IsRunning, false);
8      animator.SetBool(IsWalking, true);
9      agent.speed = _walkingSpeed;
10     Patrolling();
11 }
12 else if (isPlayerInSightRange && !isPlayerInAttackRange && !_isAttacked && !_isDying)
13 {
14     // Chase Player
15     animator.SetBool(IsAttacking, false);
16     animator.SetBool(IsWalking, false);
17     animator.SetBool(IsRunning, true);
18     agent.speed = _runningSpeed;
19     ChasePlayer();
20 }
21 else if (isPlayerInSightRange && isPlayerInAttackRange && !_isAttacked && !_isDying)
22 {
23     // Attack Player
24     animator.SetBool(IsWalking, false);
25     animator.SetBool(IsRunning, false);
26     AttackPlayer();
27 }
28 ...

```

Listing 7: Enemy movement states

### 3.3 Core game mechanic: Finding the key

As the focus of the project lies on the visuals and one of my initial goals was to create a little game where the generated dungeon is used, I decided to let the player search a key in the dungeon. With the key it is possible to open a door that has to be found as well. Through the door the next level can be reached. With searching the key the player is forced to look around while exploring the dungeon. This should lead to recognizing the visuals more.

The table with the key, that is necessary to open the door, is placed in the middle of a random chosen room. The door is placed in the end room. Determining that room was explained in section 3.1.2 on page 10. To prevent that the door is placed in front of a corridor the methods for the wall creation are used to check if the wall borders on a corridor. The door will then be placed in a wall that does not border a corridor.

## 4 Conclusion and Outlook

### 4.1 Conclusion

I learned really much throughout the last few weeks. In the end the whole dungeon is generated including the walls which were prefabs at first. My understanding for procedural content generation is now much better than before the project. Beside the generation of the dungeon I also took a look on combat systems and implemented one but unfortunately it did not match my criteria because it would have needed much more work.

In general I learned a lot about developing in Unity for example adding lights, show messages through a trigger and many other things as I did not have that much experience before. Furthermore there was also a bit of game design and user testing while implementing the combat system. I think for the period of five weeks I did a good work.

### 4.2 Outlook

For the future there is still so much that can be done about this project. It would be possible to put more effort into the combat system and make the search for the key more difficult. Another thing that is still not completely finished is the game mechanic, here I would like to add more than one key per level. For this it would also be cool if the player would be able to set marks which will be displayed on the mini map to mark the door if it is found before all the keys. I have many ideas and hope that I can continue my work on that topic in the future.

# List of Figures

2.1	Example of a BSP generated dungeon . . . . .	4
2.2	Example of a Cellular Automata generated dungeon . . . . .	5
2.3	Example of a Delaunay Triangulation generated dungeon . . . . .	6
2.4	Performance of the Binary Space Partitioning algorithm . . . . .	6
2.5	Performance of the Delaunay Triangulation algorithm . . . . .	7
3.1	Floor and roof of the dungeon . . . . .	8
3.2	Start room (blue) and end room (red) . . . . .	10
3.3	Walls of the dungeon . . . . .	11
3.4	Detecting corridors between room corners . . . . .	11
3.5	Detecting corridors at room corners . . . . .	12
3.6	Lights in the dungeon . . . . .	14

## List of Listings

1	Creation of floor and roof . . . . .	9
2	Determining end room . . . . .	10
3	Creating wall mesh . . . . .	13
4	Saving light positions . . . . .	15
5	Rotating scenery . . . . .	15
6	Player attack . . . . .	16
7	Enemy movement states . . . . .	17

# Bibliography

- Brackeys. (2017). How to make a Minimap in Unity. Retrieved November 1, 2022, from <https://www.youtube.com/watch?v=28JTTXqMvOU>
- Brackeys. (2020). THIRD PERSON MOVEMENT in Unity. Retrieved October 5, 2022, from <https://www.youtube.com/watch?v=4HpC--2iowE>
- Dave / GameDevelopment. (2020). FULL 3D ENEMY AI in 6 MINUTES! || Unity Tutorial. Retrieved October 16, 2022, from <https://www.youtube.com/watch?v=UjkSFoLxesw>
- Developer, G. (2015). 7 combat systems that every game designer should study [Section: design]. Retrieved October 12, 2022, from <https://www.gamedeveloper.com/design/7-combat-systems-that-every-game-designer-should-study>
- G2A.COM. (2021). 20 Games with good combat systems | Updated 2021. Retrieved October 12, 2022, from <https://www.g2a.com/news/features/games-with-good-combat/>
- How to generate Nav Mesh floor in a procedural generated maze? - Unity Answers. (n.d.). Retrieved October 16, 2022, from <https://answers.unity.com/questions/1480956/how-to-generate-nav-mesh-floor-in-a-procedural-gen.html>
- Lambottin, S. (2012). The Fundamental Pillars of a Combat System [Section: design]. Retrieved October 4, 2022, from <https://www.gamedeveloper.com/design/the-fundamental-pillars-of-a-combat-system>
- Leismann, J. (2021). Der komplette Unity Kurs - Werde C# Spieleentwickler. Retrieved October 15, 2022, from <https://www.udemy.com/course/der-komplette-unity-kurs/>
- Linden, R., Lopes, R., & Bidarra, R. (2014). Procedural Generation of Dungeons. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6, 78–89. <https://doi.org/10.1109/TCIAIG.2013.2290371>
- NathisGreen / pcgdungeons — Bitbucket. (n.d.). Retrieved October 3, 2022, from <https://bitbucket.org/NathisGreen/pcgdungeons/src/master/>

- Roguelike Celebration. (2020). Herbert Wolverson - Procedural Map Generation Techniques. Retrieved October 3, 2022, from <https://www.youtube.com/watch?v=TILIOgWYVpI>
- Simonds, A.-D. (2016). Answer to "Can I create a light source via script in Unity?" Retrieved October 23, 2022, from <https://stackoverflow.com/a/40121589>
- Technologies, U. (n.d.). Unity - Scripting API: Mesh.CombineMeshes. Retrieved October 25, 2022, from <https://docs.unity3d.com/ScriptReference/Mesh.CombineMeshes.html>
- Valley, S. (n.d.). (10) Unity 3d procedural dungeon generator - YouTube. Retrieved September 30, 2022, from <https://www.youtube.com/playlist?list=PLcRSafycjWFfEPbSSjGMNY-goOZTuBPMW>
- Williams, N. (n.d.). An Investigation in Techniques used to Procedurally Generate Dungeon Structures. <http://www.nathanmwilliams.com/files/AnInvestigationIntoDungeonGeneration.pdf>