R&D Assignment

3D Dungeon Generator

Hogeschool van Amsterdam
Gameplay Engineering

Michael Spiegel
27th October 2022

Contents

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 walk through 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. 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 can run with 60 FPS on average hardware. 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 put more effort in the dungeon itself. 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.)

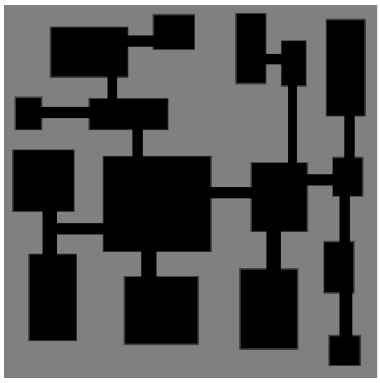

Figure 2.1: Example of a BSP generated dungeon
Source: Williams, n.d.

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 it self 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.)

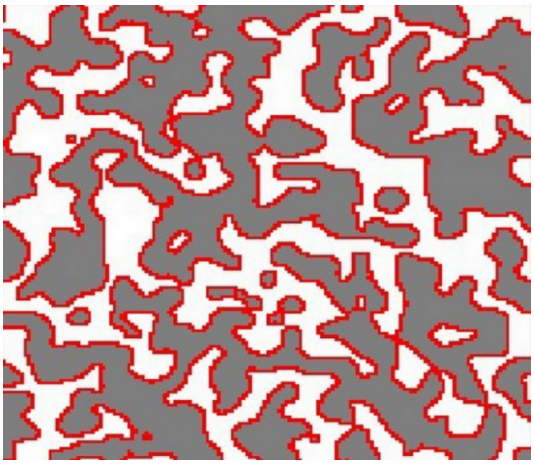

Figure 2.2: Example of a Cellular Automata generated dungeon
Source: Williams, n.d.

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.)

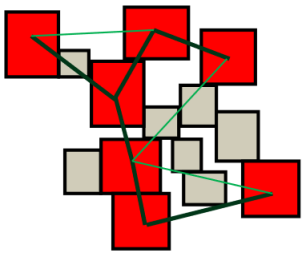

Figure 2.3: Example of a Delaunay Triangulation generated dungeon
Source: Williams, n.d.

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.

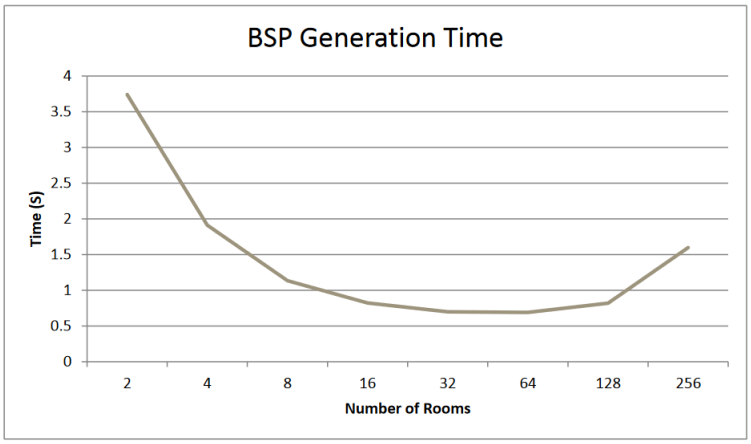

Figure 2.4: Performance of the Binary Space Partitioning algorithm
Source: Williams, n.d.

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.

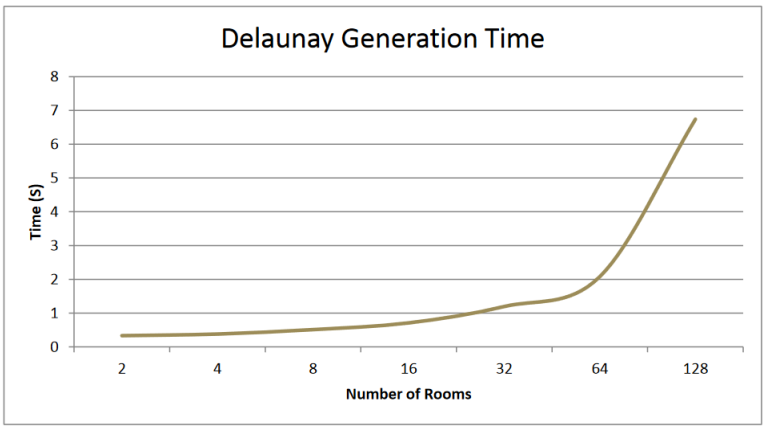

Figure 2.5: Performance of the Delaunay Triangulation algorithm
Source: Williams, n.d.

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

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 collection 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.

```
// Place triangles inverted if it is not the floor
int[] triangles = new int[6];
if (isFloor)
{
    triangles[0] = 0;
    triangles[1] = 1;
    triangles[2] = 2;
    triangles[3] = 2;
    triangles[4] = 1;
    triangles[5] = 3;
}
else
{
    triangles[0] = 2;
    triangles[1] = 1;
    triangles[2] = 0;
    triangles[3] = 3;
    triangles[4] = 1;
    triangles[5] = 2;
}
```

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.

```
foreach (var (room, index) in listOfRooms.Select((room, index) => ( room, index )))
{
    // Choose first generated room as start room
    if (index == 0)
    {
        _dungeonFloors.Add(
                CreateFloorMesh(
                    room.BottomLeftAreaCorner,
                    room.TopRightAreaCorner,
                    startRoomMaterial,
                    0,
                    true,
                    index
                )
```

```
            );
    }
    ...
}
```

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.

```
...
foreach (var (room, index) in listOfRooms.Select((room, index) => ( room, index )))
{
    ...
    // Check distance between start room and each other room
    // to find the room which is the most distant from the start room
    if (index > 0 && index < listOfRooms.Count / 2)
    {
        RoomNode currentRoom = (RoomNode)listOfRooms[index];
        float dist = Vector3.Distance(startRoom.CentrePoint, currentRoom.CentrePoint);

        if (!(maxDistance < dist)) continue;

        maxDistance = dist;
        endRoom = _dungeonFloors[index];
    }
}
...
```

3.1.3 Creating walls

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.

```
// Bottom horizontal wall
if (corridor.TopLeftAreaCorner.y == room.BottomLeftAreaCorner.y &&
    room.BottomLeftAreaCorner.x < corridor.TopLeftAreaCorner.x &&
    corridor.TopLeftAreaCorner.x < room.BottomRightAreaCorner.x &&
    room.BottomLeftAreaCorner.x < corridor.TopRightAreaCorner.x &&
```

```
    corridor.TopRightAreaCorner.x < room.BottomRightAreaCorner.x)
{
    isCorridorBetweenHorizontalBottom = true;

    vertices = CollectHorizontalWallVertices(
        room.BottomLeftAreaCorner, corridor.TopLeftAreaCorner
    );
    CreateWallMesh(
        vertices, room.BottomLeftAreaCorner, corridor.TopLeftAreaCorner, true,true
    );
    vertices = CollectHorizontalWallVertices(
        corridor.TopRightAreaCorner, room.BottomRightAreaCorner
    );
    CreateWallMesh(
        vertices, corridor.TopRightAreaCorner, room.BottomRightAreaCorner, true, true
    );
}
...
```

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.

```
private List<Vector3> CollectHorizontalWallVertices(
        Vector2Int startCorner, Vector2Int endCorner
    )
{
    var vertices = new List<Vector3>();

    // Start in one corner of the room and go the the other corner of the room
    for (var height = 0; height <= dungeonHeight; height++)
    {
        for (var x = startCorner.x; x <= endCorner.x; x++)
        {
            var vertex = new Vector3(x, height, startCorner.y);
            vertices.Add(vertex);
        }
    }

    return vertices;
}
```

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 clock wise or counter clockwise for all walls.

```
int length = isHorizontal ? endCorner.x - startCorner.x :
                            endCorner.y - startCorner.y;
...
if (!isFlip)
{
    for (int y = 0; y < dungeonHeight; y++)
    {
        for (int x = 0; x < length; x++)
        {
            triangles[tris] = vert;
            triangles[tris + 1] = triangles[tris + 4] = vert + length + 1;
            triangles[tris + 2] = triangles[tris + 3] = vert + 1;
            triangles[tris + 5] = vert + length + 2;

            vert++;
            tris += 6;
        }

        vert++;
    }
}
else
{
    for (int y = 0; y < dungeonHeight; y++)
    {
        for (int x = 0; x < length; x++)
        {
            triangles[tris] = vert;
            triangles[tris + 1] = triangles[tris + 4] = vert + 1;
            triangles[tris + 2] = triangles[tris + 3] = vert + length + 1;
            triangles[tris + 5] = vert + length + 2;

            vert++;
            tris += 6;
        }

        vert++;
    }
```

```
}
...
```

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.

```
var bottomLeftCorner = new Vector3(
    room.BottomLeftAreaCorner.x,
    0,
    room.BottomLeftAreaCorner.y
);

var horizontalBottomRotation = 0;

var horizontalBottomLength = bottomRightCorner.x - bottomLeftCorner.x;

// Horizontal bottom
bool isGettingLight = false;
int lightFrequencyTemp = lightFrequency;
int distancePerLight = (int)Math.Ceiling(horizontalBottomLength / lightFrequencyTemp);
int steps = 1;

for (var x = (int)bottomLeftCorner.x; x <= (int)bottomRightCorner.x; x++)
{
    var position = new Vector3(x, 0, bottomLeftCorner.z);

    if (x == (int)bottomLeftCorner.x + (distancePerLight * steps))
    {
        steps++;
        isGettingLight = true;
    }

    SaveLightPosition(position, horizontalBottomRotation, isGettingLight);
    isGettingLight = false;
}
...
```

As positions can occur multiple times because of overlapping between walls, 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.

```
private void SaveLightPosition(Vector3 position, int rotation, bool isGettingLight)
{
    Vector3Int point = Vector3Int.CeilToInt(position);

    if (_possibleLightPositions.Contains(point))
    {
        _possibleLightPositions.Remove(point);
        _ignoreLightPositions.Add(point);

        if (_lightPositions.ContainsKey(point))
        {
            _lightPositions.Remove(point);
        }
    }
    else if(!_ignoreLightPositions.Contains(point))
    {
        _possibleLightPositions.Add(point);
        if (isGettingLight)
        {
            _lightPositions.Add(point, rotation);
        }
    }
}
```

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.

```
private void CombineMeshes(GameObject parent, bool isFloorOrRoof)
{
    Vector3 position = parent.transform.position;

    MeshFilter[] meshFilters = parent.GetComponentsInChildren<MeshFilter>();
    CombineInstance[] combine = new CombineInstance[meshFilters.Length];

    int i = 0;
```

```
9       while (i < meshFilters.Length)
10      {
11          combine[i].mesh = meshFilters[i].sharedMesh;
12          combine[i].transform = meshFilters[i].transform.localToWorldMatrix;
13          meshFilters[i].gameObject.SetActive(false);
14
15          i++;
16      }
17      parent.transform.GetComponent<MeshFilter>().mesh = new Mesh();
18      parent.transform.GetComponent<MeshFilter>().mesh.CombineMeshes(combine);
19      parent.isStatic = true;
20
21      // Add material
22      parent.GetComponent<Renderer>().material = isFloorOrRoof ? floorMaterial : wallMaterial;
23      // Add collider
24      parent.AddComponent<MeshCollider>();
25      // Add layer
26      parent.layer = isFloorOrRoof ? Layer.GroundLayer : Layer.WallLayer;
27
28      parent.transform.position = position;
29      parent.transform.gameObject.SetActive(true);
30  }
```

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)
```

```
{
    _isAttacking = true;
    animator.SetBool(IsAttacking, true);
}
...
```

3.2.2 Enemy

The enemy has three states:

1. Patroling
2. Chasing
3. 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.

```
...
// Switch to appropriate state
if (!isPlayerInSightRange && !isPlayerInAttackRange && !_isAttacked && !_isDying)
{
    // Patroling
    animator.SetBool(IsAttacking, false);
    animator.SetBool(IsRunning, false);
    animator.SetBool(IsWalking, true);
    agent.speed = _walkingSpeed;
    Patroling();
}
else if (isPlayerInSightRange && !isPlayerInAttackRange && !_isAttacked && !_isDying)
{
    // Chase Player
    animator.SetBool(IsAttacking, false);
    animator.SetBool(IsWalking, false);
    animator.SetBool(IsRunning, true);
    agent.speed = _runningSpeed;
    ChasePlayer();
}
else if (isPlayerInSightRange && isPlayerInAttackRange && !_isAttacked && !_isDying)
{
    // Attack Player
    animator.SetBool(IsWalking, false);
```

```
25      animator.SetBool(IsRunning, false);
26      AttackPlayer();
27  }
28  ...
```

List of Figures

Bibliography

Williams, N. (n.d.). An Investigation in Techniques used to Procedurally Generate Dungeon Structures. http://www.nathanmwilliams.com/files/AnInvestigationIntoDungeonGeneration.pdf