Procedural Generation in Games: Focusing on Dungeons

Zhenyuan Shen*

School of Computing and Mathematical Sciences, University of Greenwich, London, SE10 9LS, UK

ABSTRACT: This report is talking about the applications of procedural generation in games. There are many AI-generated games on the market, such as No Man's Sky. Due to the huge curiosity about it and the purpose of studying how procedural generation works, the research delved into the principles of procedural generation of dungeons. To start with the research, we first focused on a dungeon, analyzing how the dungeon is created and then we consider how the procedural generation works. Typically, all the resources are from the internet, including some academic tutorials. By separating the whole problem into three parts (terrain generation, content generation and object generation), we finally realized each part of the procedural generation, and how it works to build a simple dungeon. First, in terrain generation, we focus on a different algorithm to generate the terrain, such as fractal terrain generation, bitmap terrain generation and generating landscape by Perlin noise. In the second part, content generation usually generates loots, enemies and creates a dungeon system with various rooms and paths. And the last one is object generation, for example, which holds creatures, trees, weapons, vehicles and so on. Through these methods, we can gain a deeper understanding of the significant role of procedural generation. All in all, through the above-mentioned aspects to illustrate their respective effects on procedural generation, and guide people's preliminary cognition and understanding of procedural generation.

1. INTRODUCTION

Procedural generation is an algorithm that can automatically generate data by computer. Due to this reason, procedural generation has a key role in digital games, especially in automatically making terrain or landscape. Typically, there are three ways to generate a random terrain (Fractal Terrain Generation, Bitmap Terrain Generation and Perlin Noise). Meanwhile, in this part of the research, not only the basic algorithms we discussed, but also the comparison of differences between each generation method. In the next section, we will discuss the content generation, which will hold an example of a random dungeon python code. At the same time, a new concept will be introduced. Obviously, a piece of rogue knowledge will be emphatically explained. Then in the last section, we will discuss object generation. Normally, object generation is about 2D/3Dmodeling, such as weapons, creatures, and vehicles. Also, the different procedural process is used to decide where, when, and how the object is created in the terrain. Based on the above points and several related codes we get, we discussed how these generation methods work in procedural generation. Meanwhile, we compared different methods to find each method's pros and cons. And finally, get a clearer understanding of the procedural generation. Due to this research, we can find different AI-generation methods suit for different situations.

2. BACKGROUND

Procedural generation is algorithmically created content. Normally, three areas will use procedural generation, including terrain generation (2D/3D generated terrain), content generation (generation and management of in-game content) and object generation (generation of visual objects). In the preliminary stages, procedural generation is used in an RPG game named Advanced Dungeons Dragons.

Although the technology was not perfect at the time, this game still has generated terrain, encounters, loot, and more automatic generation. When the time goes back to the 1980s, roguelike games Occupied most of the sub-applications by procedural generation. For example, the games of Beneath Apple Manor (1978) and Rogue (1980) are the most typical masterpiece. Apparently, generate rooms, hallways, monsters, and treasures are the main part of the application in procedural generation. However, with the development of technology, increased games that are based on procedural generation are created and most of them became popular games to people, such as Borderlands, Minecraft, No man's sky and Diablo. Also, procedural generation is reflected in loot systems (random weapons and enemies in Borderlands), open world terrain generation (infinite random open world in Minecraft), space exploration and trading games (random generated 3D terrain and music in No man's sky), dungeon crawler / roguelike (random dungeon to avoid bored in Diablo).

*Corresponding author. Email: zs7008d@gre.ac.uk

Then there will be a part of how to generate a random dungeon using the above methods. Unlike other games, immersion in the game is also one of the reasons why the dungeon game type is so popular. Thanks to its fast-paced, real-time action, simplicity, and excellent graphics with 3D views, this type of game often has a limited storyline, in which the characters usually must explore an area (usually like a maze or dungeon) while killing any monsters met. (Actually, monsters, creatures, the weapon can be randomly created by object generation, and the status of them can be also created by stat generation) Also, like most computer games, dungeon games are divided into multiple levels. A level can be considered as an independent unit. The player starts from a position in the level and usually must pass the physical area is the end of the level when the current level reaches a certain point. At this point, the level has been completed and players can start the next level. At the same time, it should be noted that the following three basic elements, should be paid attention to when building a dungeon game. including representative model, a method for this model and a method of creating the actual geometry of the dungeon from this model.

3. EXPERIMENTS AND RESULTS

In this part, although we get the code of generating random dungeon, three methods to generate random terrain and making an easy rogue maze, the purpose for my research part is about automatic random dungeon generating and analyzing each of the three methods and making a comparison to them. According to the code we have, it is easy to generate an easy dungeon if you follow four steps [1]:

Generate an empty map:

```
def init_map():
    #Initializes the map of key/value pairs.
    for y in range(map_height):
        for x in range(map_width):
            map[x,y] = 0 # set every square to a wall
```

Figure 1. Generating_map

Create random rooms, making sure none of them overlap:

```
def init_rooms():
    #Initializes the rooms in the dungeon.
    total_rooms = randrange(min_rooms,max_rooms)
    for i in range(max_iters):
        for r in range(total_rooms):
            if len(rooms) >= max_rooms:
                break
    x = randrange(0,map_width)
            y = randrange(0,map_height)
    width = randrange(min_room_size,max_room_size)
            height = randrange(min_room_size,max_room_size)
            room = Room(x,y,width,height)
    if check_for_overlap(room, rooms):
                pass
            else:
                rooms.append(room)
    for room in rooms:
        for y in range(room.y, room.y+room.height):
            for x in range(room.x, room.x+room.width):
                map[x,y] = 1

def check_for_overlap(room, rooms):
    #Return false if the room overlaps any other room.
    for current_room in rooms:
        xmin1 = room.x
        xmax1 = room.x + room.width
        xmin2 = current_room.x
        xmax2 = current_room.x + current_room.width
        ymin1 = room.y
        ymax1 = room.y + room.height
        ymin2 = current_room.y
        ymax2 = current_room.y + current_room.height
        if (xmin1 <= xmax2 and xmax1 >= xmin2) and \
           (ymin1 <= ymax2 and ymax1 >= ymin2):
            return True
    return False
```

Figure 2. Random_rooms

Connect rooms at random with horizontal and vertical passageways:

```
def connect_rooms():
    #Draws passages randomly between the rooms.
    shuffle(rooms)
    for i in range(len(rooms)-1):
        roomA = rooms[i]
        roomB = rooms[i+1]
    for x in range(roomA.x,roomB.x):
        map[x,roomA.y] = 1
    for y in range(roomA.y, roomB.y):
        map[roomA.x,y] = 1
    for x in range(roomB.x,roomA.x):
        map[x,roomA.y] = 1
    for y in range(roomB.y, roomA.y):
        map[roomA.x,y] = 1
```

Figure 3. Connecting_rooms

Draw the map:

```
def draw_dungeon():
    #Draw the dungeon with cario rectangles.
    surface = cairo.ImageSurface(cairo.FORMAT_RGB24,500,500)
    ctx = cairo.Context(surface)
    for y in range(50):
        for x in range(50):
            r = randrange(1,10)
            if map[x,y] == 0:
                ctx.set_source_rgb(0.3,0.3,0.3)
            else:
                ctx.set_source_rgb(0.5,0.5,0.5)
            ctx.rectangle(x*10, y*10, 10, 10)
            ctx.fill()
    surface.write_to_png("dungeon.png")
    print("Total rooms: " + str(len(rooms)))
```

Figure 4. Drawing_map

Considering the default dungeon is a block. By imagining the dungeon as a chessboard, we can create different rooms in it, and use random numbers to control different variables of the room, such as the number of rooms, the maximum and minimum sizes. Next, by writing a function to initialize and fill the room data. After that, by drawing horizontal or vertical channels between random rooms, we can create a simpler dungeon. In addition, in

fractal terrain generation, the principle of the algorithm is simple. If it first generates two random numbers and draws a line between them. After that, find the midpoint of the line, a new node will be created at this point. Shifting up or down by a random amount of that is proportional to the height difference between the start and endpoints of the edge. And the new node forms two new lines and repeats this operation. When getting enough nodes that the gap between each has shrunk to a certain size, the process stops. Another technique [2] described involves using bitmaps instead of fractals to generate terrain. To start with, a small bitmap with several basic features on it is constructed. Scaling this map so that it meets the size of the required map. Then using image processing techniques such as Convolutional Neural Network (CNN). Using the value of neighboring pixels to calculate the new value of a certain pixel, which can be used to smooth the map and random factors can be used at this stage as well. The rest is noise, which is simply a collection of numbers that, when defined with parameters such as elevation and frequency, supplies the foundation for terrain. By using Perlin Noise, a smooth terrain will be created. At the same time, by adding noise at different frequencies and raising elevation into a power, a reasonable terrain will be generated. Through these codes, we can compare each method to find the typical features of each method, including the related applications for different situations and the shortcomings or limitations for different algorithms.

4. DISCUSSION THE RESULTS AND MATERIALS

4.1. Fractal terrain generation

It is obvious that fractal automatic terrain generation can save game developers a lot of time by reducing the amount of height data and they must generate themselves. During studying this algorithm, the reason fractals can be used to terrain is apparently. The main reason is that terrain is self-similar [3]. This statement seems abstract; however, we could imagine that the magnified subsets of the objects look like the whole to each other. Taking the mountains as an example, the horizon of a mountain is not flat, but it is rugged. If we zoomed in on a part of the hillside, it would also look uneven, just like the surface of the hillside, a single rock or stone that is part of it. Therefore, using the fractal terrain generation as the article says above, a cross-section of a rugged mountain is thus produced. One most typical case in today's game is Terraria, the terrain in Terraria is randomly generated by fractal terrain generation, which creates different terrain situations, such as mountains, riverbeds, and caves.

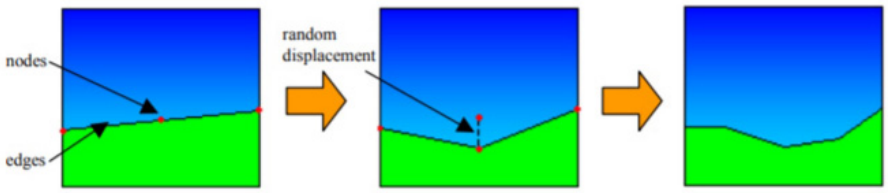

Figure 5. Fractal_terrain generation

4.2. Bitmap terrain generation

Unlike fractal terrain generation, this is not a completely random technique. There are already several first features on the map, and the process uses it to generate detailed data for each small feature, then the units that make up the map, which means the final terrain is something like the zooming terrain from the first terrain. Because of the unique features of zooming, bitmap terrain generation is usually the case for massively multiplayer game maps. Because the game designer may wish to have certain functions in several places, but do not care about the exact height of each square. Hence, they will indeed be very manually generating data for each square, which is very time-consuming. Thus, this technique helps to save plenty of time, otherwise, the game programmers need plenty of time consuming to generate code for each different terrain situation. Nevertheless, the generated data can be enhanced to make the terrain looks more exquisite the important thing is that bitmap terrain generation can be used to generate data other than the height of each tile and each pixel value can correspond to a specific terrain type, such as desert or jungle (which means dividing different types of terrain).

Figure 6. Bitmap_terrain_generation

4.3. Perlin noise

It is a common method to use a noise function to generate 2D-terrain, but normally we choose Perlin noise instead of normal noise [4]. The reason we choose Perlin noise is easy to understand. Noise is a random number generator, and the random numbers generated by ordinary noise have no rules at all (Perlin noise is pseudorandom). Therefore, the cascading mountains in nature, the texture of marble, and the undulating waves on the sea surface seem to be chaotic, but there are inherent laws to follow. Normal noise cannot simulate these natural effects. The Perlin noise algorithm makes these possible. Therefore, a set of smoothly interpolated random numbers can be obtained by using the Perlin noise algorithm, which is correlated with each other and can be used to generate random values close to nature. By seeing the random texture generated by normal noise and the texture generated by Perlin noise, it can be found that the texture generated by Perlin noise is

more natural and smoother, with obvious transition effects between random values.

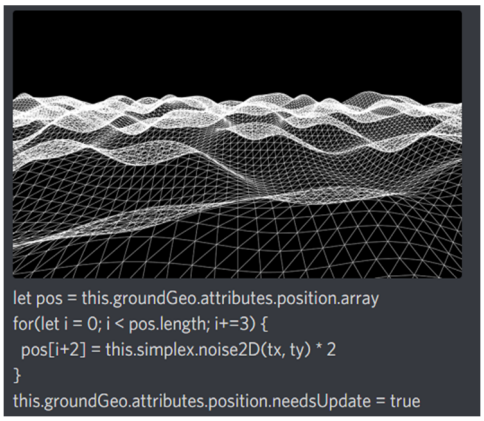

Figure 7. Normal_noise

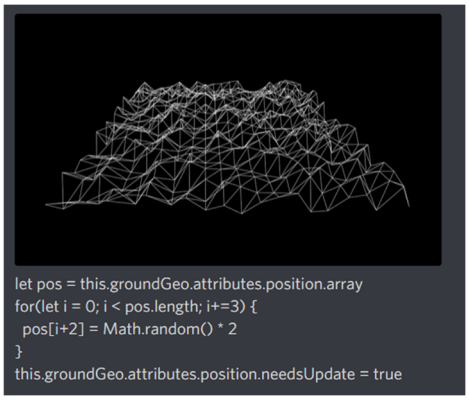

Figure 8. Perlin_noise

4.4. Avoid overlap and random dungeons

In the code we have, to improve our dungeon and ensure that our dungeon has some regular rooms, we need to check that no two rooms overlap. We can do this by checking the location and size of the room. When we create a new room, we check if it overlaps with any room we have already created. If it is, we will reject it and try to create another one.

Then let's focus on the real situation in games. For example, in Civilization [5], the game world is spilled up into squares and each square is a certain type of terrains, such as grassland, forest, or ocean. Without random content generation, civilization, and other similar games, greatly reduce replay-ability. People will soon lose their interest in games. By exploring the game, the world will become boring and boring because the players will already know what they will find, and over time it will be clear for a player to use what kind of strategy to produce the same results repeatedly. In this way, the lifetime of the game will reduce. While in Diablo [6], it has randomly generated dungeons that the game character must journey through on a quest. However, the resulting dungeon was discovered by many people too similar. Besides, these dungeons can only be generated by applying certain restrictions. By using random dungeons, the game's playability has been improved, and it can give people more freshness.

5. CONCLUSION

In this report, we first analyzed procedural generation and divided it into three parts, including terrain generation, content generation and object generation. In the background part, several reality games are showed to explain how these three generation methods work in a popular game. And then we talked about how to generate a dungeon. The first section discusses the fundamental elements of typical dungeon generation. Then we created some code to analyze them. In our experiment part, the research focused on the differences for different terrain generation methods and the principles of each generation method to analysis. And finally, taking the existing game applications as an example, the role of procedural generation in the dungeon is analyzed.

As a result, using procedural generation can greatly reduce the money and time consuming and it can randomly generate different content for players. Meanwhile, choosing a suitable method to generate terrain will get better feedback. For example, fractal terrain generation will give a random terrain in a quick way and with a little number of codes to program, while bitmap terrain generation will create a large-scale feature terrain, which can make reduce the time consuming for programmers and if necessary, the terrain can be perfectly fine. As for the Perlin noise, it can generate a smoother and closer to the natural terrain. For future work, we decided to combine the codes and generate a real dungeon. After that, focus on the different dimension dungeons, especially generating different dimension terrains.

In conclusion, this paper discusses several aspects of procedural generation, including terrain generation, content generation, object generation, and Perlin noise, to gain a better understanding and preliminary understanding of procedural generation. And by trying to build dungeons, people can experience the actual application of procedural generation.

REFERENCES

1. J. Petty Harold, Create a Random dungeonwith Python, https://python.plainenglish.io/create-a-random-dungeon-with-pythonf17118c1eebd, 2021.
2. K. Martin, Using Bitmaps for Automatic Generation of Large-Scale Terrain Models, Gamasutra. Available from: http://www.gamasutra.com/features/20000427/martinpfv.html, 2000.
3. R. Voss, Fractals in Nature: Characterization, Measurement, and Simulation., SIGGRAPH, 1987.
4. Xingge, Perlin Noise vs Normal Noise, https://dev.xingway.com/threejs-perline-noise/, 2020.
5. T. Chown, The Case for Random Maps, Games Domain., Available from: http://www.gamesdomain.com/gdreview/depart/nov98/rmg.html, 2000.
6. Blizzard, Diablo., Available from: http://www.blizzard.com/, 1998.