

# Procedural Low-Poly Terrain and Village Generation

Zijie Zhu

Yiheng Zhang

## 1. Motivation

When generating a scene, the traditional and most straightforward approach is to specify the shape and position of each object mesh. While artists and programmers have complete control over the scene under this approach, it also requires every detail of the scene to be manually specified as the input data. To create a large scene, we would need huge input files that consume too much space. In applications that render a lot of large background scenes, especially in video games, another approach is often favored — procedural generation. Procedural generation creates content algorithmically, usually utilizing randomness or rule-based systems. Reducing the scene generation to a program with a few basic meshes greatly cuts down on the required space for the scene. In addition, the randomness of procedural generation creates highly complex and realistic scenes that are no less visually pleasing than those generated manually. With procedural generation it is also possible to create seemingly infinite content —

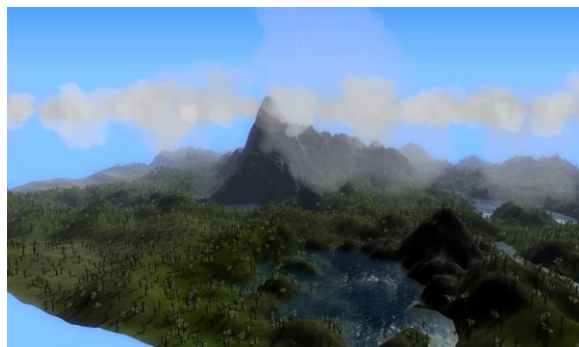


Figure 1. An example of realistic procedural terrain generation.

generating endless new content around the camera as the camera moves.

notice that while there are many available research papers on either procedural city or terrain generation, the experimentation of combining the two topics is relatively unexplored. Therefore, we decided that our project will be a fun combination with cute aesthetics represented in low-polygon forms.







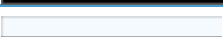

To gain some insights into the subject, we researched other people’s work on both procedural city modeling and procedural terrain generation. We soon decide to use the Unity3D engine for this project for its nice work environment and large community support. Fortunately, we also found two YouTube tutorials<sup>2,3</sup> on procedural village and terrain generation respectively, which greatly help us jump start with the project.

## 2. Background

As the game “No Man’s Sky” comes out, we were deeply fascinated by the procedural planet and terrain generation in the game. From planet to planet, the player could endlessly explore the universe in a realistic setting. Therefore, we decided that procedural generation would be an interesting topic to explore. After some research, we mainly focused on procedural city generation and procedural terrain generation. We

## 3. Procedural Terrain Generation

First, we initialize a Perlin noise map with specified width and height. For each point on the noise map, the Perlin noise ranges from  $[0, 1]$  and reflects the relative altitude of that location. Then, we specify the noise interval for each color to generate a color map. For example, in our final example, we have the following eight color intervals for different terrain types:

Terrain Type	Noise Interval	Color
<b>Ocean</b>	[0.00, 0.25]	
<b>Deep Water</b>	[0.25, 0.35]	
<b>Shallow Water</b>	[0.35, 0.40]	
<b>Beach</b>	[0.40, 0.45]	
<b>Ground</b>	[0.45, 0.55]	
<b>Valley</b>	[0.55, 0.65]	
<b>Mountain</b>	[0.65, 0.80]	
<b>Snow</b>	[0.80, 1.00]	

After we have the color map, we then connect the grid points on the map and triangulate them into surfaces. Based on each point's Perlin noise value, we scale it to a proper height value to generate the 3D meshes. However, there are two things to notice:

1. Since the water level should be even, the projected height of water is the same, no matter it's "ocean," "deep water" or "shallow water."
2. For later village generation, we want to generate plateaus for a particular height interval (we call it the "livable height interval").

Therefore, the function from noise value to height value looks like the following image. Notice the curve is almost flat at  $[0, 0.4]$  (water level) and  $[0.5, 0.6]$  (village level).

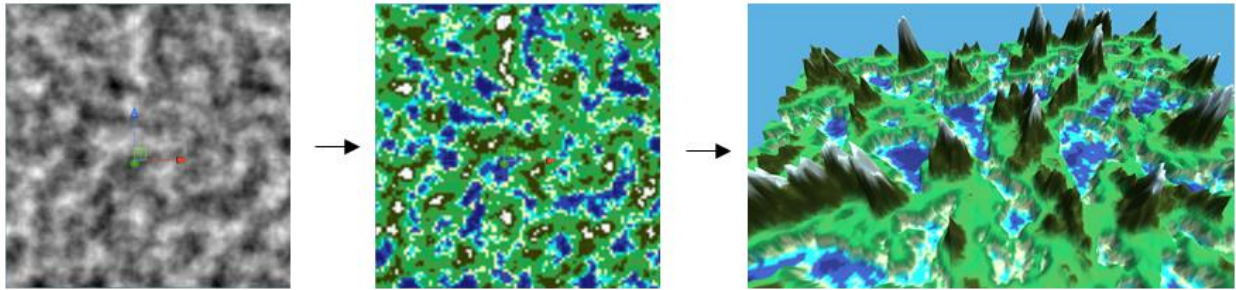


Figure 2. From top to bottom: noise map, color map, 3D meshes, and height function.

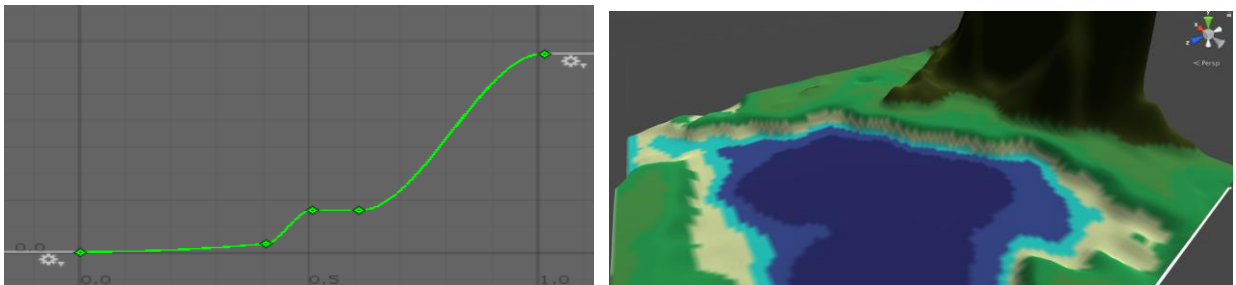


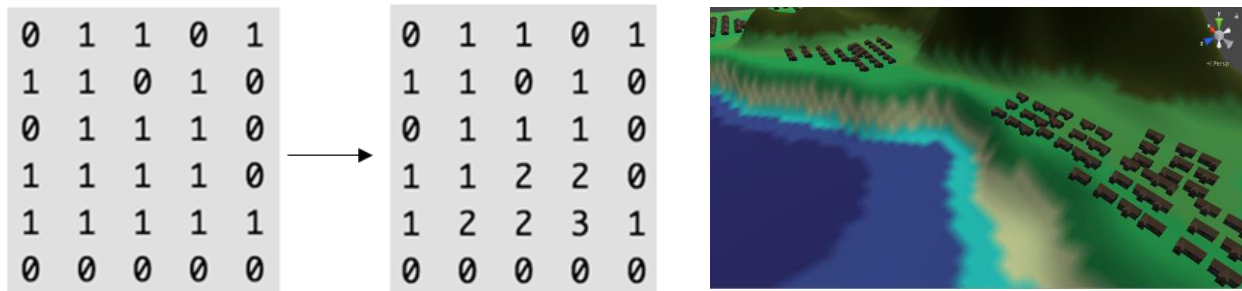
Figure 3. The height function and result. Notice that the function is almost flat at  $[0, 0.4]$  because the water level should be even, and that then function is perfectly flat at  $[0.5, 0.6]$  because we will generate villages procedurally at this height level. The figure on the right demonstrates the results, where we have flat areas to generate the villages and even water level.

The above is the general approach we construct the procedural landscape. More specifically, we could construct the Perlin map differently by using different seeds and different noise scale. Also, we could change the level of details of the terrain by only sampling every  $n$ -th grid point in the surface construction step. For results on these details, please see the attached demonstration video below.

## 4. Procedural Village Generation

For the method of procedural village generation, we first consider the paper by Parish and Müller<sup>1</sup>. While the L-System method they introduced generates highly complex and interesting cities, it does not fit into our project because 1) using the L-System have to keenly aware of the terrain heights and restrain itself correspondingly, and 2) we only need to generate procedural villages, not skyscrapers that require rules to lay each floor up. Therefore, we adopted a simpler grid layout for the procedural villages, which we will explain in more details below.

Once we have created flat areas on the map, we need to find suitable village locations within those areas. Since our village uses a square grid, we used an algorithm to find the squares of length greater than a certain threshold on our map. First we generated a 2D array representing whether each position on the map is inside the livable height interval ([0.5, 0.6]). Then the algorithm creates a copy of the map's 2D array and traverses that copy while incrementing a position's value if it could be part of a square. In essence, the algorithm takes in a 2D array of 1's and 0's and outputs another 2D array with the square's length at the position of the square's bottom-right corner. This process is demonstrated below; the value '3' at the position (4, 3) indicates that a square of length 3 can be created with (4, 3) as the bottom-right corner.



With this example output array, we can easily find the location of square of length greater than or equal to 3. One caveat is that the order in which we traverse the map to generate the first array is not the same as the order in which the algorithm traverses the array; so the position (4, 3) in the output array actually corresponds to the position (3, mapHeight - 4) on the map.

Another problem that we had to solve was avoiding overlapping villages, which meant checking for collision while we are finding the potential square locations. Our solution was to keep another 2d array that represents the occupied positions on the map; every time we find a new suitable square, we check to see if any of its points are already occupied, if so we discard the square. This ensures that all the squares we keep do not share any points with each other. Once we had these squares, we simply used the method from the procedural city tutorial and replaced the building meshes to build a village within each square. The result is shown above.

## 5. Decorating the Terrain

After generating the terrain, we randomly place decorations on the terrain based on its height. See the below table for more details. Then, we also added low-polygon clouds at random positions above the

<sup>1</sup>Parish, Yoav I. H., and Pascal Müller. "Procedural Modeling of Cities" (2001) ETH Zürich.

terrain, animated bird flocks flying through mountains, and a dragon always landing on the highest mountain and glaring at the terrain below.

Map Height	Terrain Type	Fishing Boat	Bush	Tree	Mushroom	Sheep
[0.00, 0.35]	Water	0.005	0	0	0	0
[0.50, 0.75]	Land	0	0.05	0.03	0.04	0.03



Figure 4. From top to bottom: Low-polygon clouds, dragon on top of the highest mountain, fishing boat model, sheep / tree / bush / mushroom model, generated village, animated bird flocks.

## 6. Video Result

Our presentation video is available at this link: <https://www.youtube.com/watch?v=RWJGp0i0bT4>.

## 7. Conclusion and Extensions

Our project demonstrates the power of procedural generation in creating large scenes. With less than five hundred lines of C# code in Unity3D, we were able to generate interesting procedural terrains and embedded villages, along with other romantic and idyllic decorations. Possible future extensions of this project include using L-system to generate more complicated and organic village layouts, and to use complex rule-systems to build multi-story buildings. In addition, to add to “coolness” of the project comparing to procedural games like “No Man’s Sky,” we could extend the project to generate procedural planets using 3D Perlin noises, or simply projecting 2D noises to spherical surfaces. Another possible extension is to generate endless terrain that is automatically generated or destroyed based on where the camera is.