Landscape Generation Using Perlin Noise

Dongho Lee, Mahin Goban, Luke Taranowski

1 Introduction

Maps are beautiful. As miniature versions of the world we live in, not only can they guide us from our origin to our destination, but also show us how the tectonic plates have drifted over hundreds of millions of years, illustrate the changes in coastlines as Planet Earth alternated between its ice ages and interglacial periods. As an Earth-bound species, it is perhaps only natural for us humans to be drawn to the vast expanses of lush green forests and miles-deep blue oceans.

For this reason, the first step of world-building is to create a map; many open-world or sandbox games feature landscape generation, including SimCity, Spelunky, Spore, Minecraft, and many more. The features which are generated can be anything: altitude, vegetation, distribution of resources, or even a complex network of caves full of ores, aquifers, and magma.

As a proof-of-concept project, we have decided to implement landscape generation by using two important concepts from Steve Rabin's course on game AI programming: altitude generation using Perlin noise and tilemap rendering using inside-outside.

2 Perlin Noise – The Backend

We will begin by implementing the "backend" part – Perlin noise. For this purpose, we have written the *PerlinNoise2D* class which works like a mathematical function:

$$PN(x,\ y) = a\ \ (x, y \in R,\ a \in (0, 1)) \tag{1}$$

The initialization steps are as follows.

1. A *PerlinNoise2D* instance is created using *seed* and *permutationLength* values: *seed* - the class utilizes predictable randomness through a seeded pseudo-random number generator – that is, the generated values for a certain position (x_0, y_0) would always be the same if it was given the same seed at instantiation – so that this feature can be used for networking, replayability, debugging, and more. *permutationLength* - the generated pattern must not repeat within the range of coordinates *(0, 0)-(L_P - 1, L_P - 1)*.
2. Create an array containing *permutationLength* gradients. A gradient is simply a 2D vector where *x, y* $\in$ *(-1, 1)*.
3. Create an array containing all integers in the range *[0, permutationLength)*, shuffled. This array will be used as a hash function.

We can see that we have generated and stored many parameters before the noise generation step takes place. What follows is the actual noise generation:

1. Pick 4 gradient vectors from the *gradients* array (the snippet omits index wrapping for brevity) – notice that the *x* coordinate is hashed to avoid creating a repeating pattern along a diagonal.

Listing 1 Getting gradients from four random places in the gradients array.

```
int x0 = floor(x);
int x1 = x0 + 1;
int y0 = floor(y);
int y1 = y0 + 1;

Vector2 g00 = gradients[perm[x0] + x0];
Vector2 g01 = gradients[perm[x0] + y1];
Vector2 g10 = gradients[perm[x1] + y0];
Vector2 g11 = gradients[perm[x1] + x1];
```

2. Compute the displacements of point *(x, y)* from its for neighboring integer coordinates.

Listing 2 Calculating displacement values from the four nearest corners.

```
float xf = x - x0;
float yf = y - y0;

Vector2 d00 = new Vector2(xf, yf);
Vector2 d01 = new Vector2(xf, yf - 1);
Vector2 d10 = new Vector2(xf - 1, yf);
Vector2 d11 = new Vector2(xf - 1, yf - 1);
```

3. Compute the dot product between each gradient and displacement values from steps 1 and 2.

Listing 3 Dot products between gradients and displacements.

```
float in00 = Dot(g00, d00);
float in01 = Dot(g01, d01);
float in10 = Dot(g10, d10);
float in11 = Dot(g11, d11);
```

4. Use “faded bilinear interpolation” (or I would like to call it, fade-blerp) to interpolate between the four dot products.

Listing 4 Faded bilinear interpolation of noise.

```
float xfade = xf*xf*xf * (xf * (xf * 6 - 15) + 10);
float yfade = yf*yf*yf * (yf * (yf * 6 - 15) + 10);

return Lerp(Lerp(in00, in10, xfade),
            Lerp(in01, in11, xfade),
            yfade) + 0.5;
```

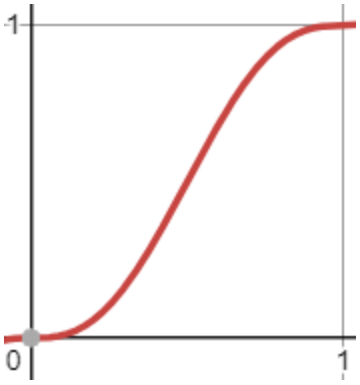

Figure 1 The S-curve generated by the fade function $f(t) = 6t^5 - 15t^4 + 10t^3$.

The resulting figure is a smooth heatmap as shown below.

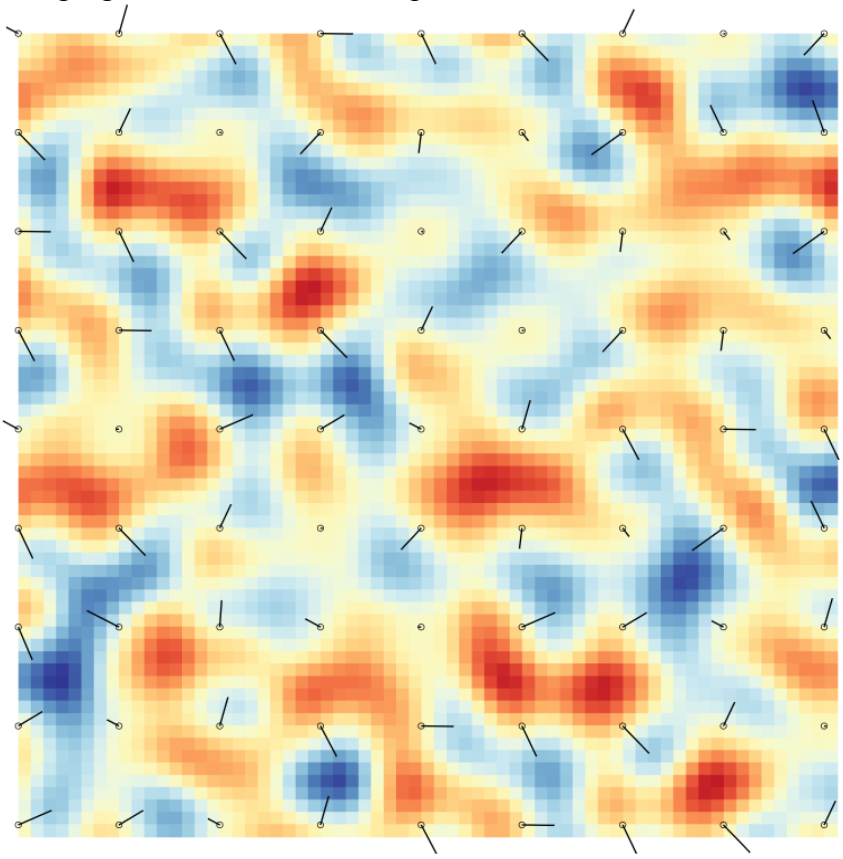

Figure 2 The heatmap generated from gradients, where blue is "hot" and red is "cold."

3 Inside-Outside – The Frontend

Now we need a way to map the generated Perlin Noise onto a tilemap. Each and every tile on the tilemap has 4 vertices, and the generated Perlin Noise values correspond to these vertices. What this means is that each singular Perlin Noise value is shared by four neighboring tiles. This sharing of vertices allows for smooth transitions between vertex values and between the

individual tiles.

Since Perlin Noise represents smooth randomness, this sharing of vertices will produce a smooth looking tilemap where tiles vertices overlap, and the landscape's randomness looks natural.

4 Tileset and the Tile class

Under the hood, we use Unity's TileMap component to generate a map. In order to populate this tilemap, we first need to initialize a tileset:

1. Decide on the number of vertex types you want.
2. Decide on the dimensions of the tilemap (2D or 3D).
3. Create sprite assets that represent all combinations of vertex types.
4. Create a class pipeline that bridges the gap between the assets and the code.

4.1 Vertex Types

In this project, we opted for two vertex types, Land and Water. We also decided to go for 2D because it avoids unnecessary complications in 3D. With these two pieces of information we are able to calculate the total number of tiles we will have in our tileset using the equation below:

$$v = \#\ vertices\ per\ tile\ unit$$
$$k = \#\ vertex\ types$$
$$n = total\ \#\ of\ tiles$$

$$n = k^v \tag{2}$$

This gives us a total of $2^4 = 16$ tiles.

4.2 Tileset

To make the correct tileset for this project, we have to first draw out each unique tile shape for the given number of vertex types. With 2 vertex types there are 6 unique shapes, the remaining 10 tiles are all rotations of these basic shapes.

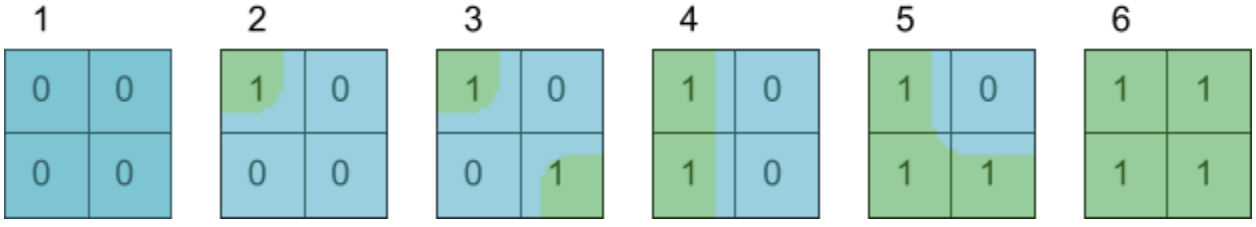

Figure 3 All 6 unique tile shapes from 2 vertex types. (0 = water, 1 = land)

As you may have noticed from Figure 3, we chose water to take precedent over land wherever the two met in a tile. This allows for nice smooth looking tilemaps; neglecting this step would defeat the purpose of using Inside-Outside. However, in the case of tile shape 3 from

Figure 3, this means our map can only have narrow water straits, and no narrow land bridges. One potential solution could involve creating two variations of this tile: one prioritizing land, and the other water. Then, choose the tile randomly from these options.

4.3 Tile Class

The Tile class implements a simple framework for storing and modifying the values of a Tile's vertices. The class has four private variables, each an integer that represents what vertex type is stored on each of its vertices. This class also enforces a convention for the rest of the pipeline: the vertices are ordered in the order of top-left, top-right, bottom-left, and bottom-right. The significance of ordering the vertices this way will become apparent in the next section.

5 Perlin Noise → Inside-Outside → Tiles

With the tileset and an intermediate class, Tile, that handles the connecting of the assets to the code finished, we can now talk about the rest of the process. The Tile class is a part of a larger class, Tilemap, that generates the Perlin Noise for all vertices on the map, and directly modifies the tiles in the tilemap to match. But how exactly do we go from Perlin Noise to the tilemap?

We implemented a pipeline that converts the generated Perlin Noise values for each tile's vertices into Inside-Outside values. Once converted, we hash these Inside-Outside values into an integer number that is used to index the correct tile and its corresponding sprite asset.

5.1 Perlin Noise → Inside-Outside

As seen in previous sections, the Perlin Noise generates values in the range (0, 1) for each vertex in the tilemap. We also store the SeaLevel parameter as a value in the range (0, 1). Using these two values, we are able to determine which vertices are to be marked as Land and which vertices are to be marked as Water with a simple check. An example code snippet is shown below:

Listing 5 Getting a tile and setting its vertex values to a vertex type

```
Vertices vertices = GetVertex(i, j);
vertices.Set(0, (values[i    , j + 1] <= seaLevel) ? 0 : 1);
vertices.Set(1, (values[i + 1, j + 1] <= seaLevel) ? 0 : 1);
vertices.Set(2, (values[i    , j    ] <= seaLevel) ? 0 : 1);
vertices.Set(3, (values[i + 1, j    ] <= seaLevel) ? 0 : 1);
SetVertex(i, j, vertices);
```

Using a double for loop, we iterate over all tiles in the tilemap. Then we get the vertices belonging to the tile at location `(i, j)`. We then set all four of its vertex values to match the values generated by the Perlin Noise at the same location, `(i, j)`, stored in the `values` array.

Once this step is done, for all tiles in the tilemap, the last step is to update the tilemap with the correct sprite assets. Recall that in section 5.3 we enforced an ordering convention. Here is where we make use of that convention.

5.2 Inside-Outside → Tiles

This is where the secret to this expandable implementation comes into play. The Tilemap class is able to handle any number of vertices and any number of vertex types. Just as long as an ordering convention is enforced, the implementation here will work. The trick is base N conversion.

For the example of 2 vertex types and 4 vertices per tile unit, we calculated that we have a total of 16 different tiles in our tileset. From the code snippet we also see that we assign a value of 0 for Water and a value of 1 for Land. This means that following the ordering convention, we can display a tile's vertex values as a binary string.

For example, a tile with Land on its top-left vertex and Water everywhere else would have a binary display string of 1000. This would also work for a larger number of vertex types. If we had three vertex types, for example, we would represent a tile's vertex values as a base 3 string. So, if we had a tile with its top-left vertex as Ice (a third vertex type after Land and Water), its bottom-left vertex as Land, and everything else as Water, its base 3 string display string would be 2010.

From here, we convert this base N display string into a decimal integer. Using this integer we can index into the correct sprite asset and its corresponding tile in the tilemap. Something to note is that this ordering convention also means that the indices associated with the sprites must also be representative of their design. If we have a sprite with index 8 (1000 as binary), it should represent the tile with Land on its top-left vertex and Water everywhere else.

5.3 Limitations

The only limitation of this implementation is the creation of the tiles. We currently do not have a way to auto-generate tiles based on the ordering convention. As a result, the tiles have to be made by hand. For the example of 2 vertex types and 4 vertices per tile unit, it only results in 16 tiles, which isn't a lot. However, increasing either number would exponentially increase the number of tiles to be created.

6 Conclusion

Generating natural looking maps with Perlin Noise and Inside-Outside requires a blend of math and art. The techniques discussed in this paper can be applied to any number of projects, from games, art, to geological simulations. It's simple and useful, and expanding on what this research already has can lead to incredible procedurally generated content.

7 References

[Simonds 23] Simonds, Ben. 2023. Perlin Noise. Observable HQ. https://observablehq.com/@bensimonds/perlin-noise (accessed November 26, 2023).