# Terrain Generation using Simplex Noise – Adam Joyce

In this write-up I will cover the concepts and algorithms I used to generate cave-like structures in two dimensional terrain.

## Simplex Noise – noise.h

I began by identifying a type of noise generation that would be suitable to create the cave's that would be located within my terrain. Fairly early on I came across Perlin Noise – an academy award winning type of gradient noise developed by Ken Perlin in 1983. Perlin Noise comes with a couple of key drawbacks: it has a large computational overhead at higher dimensions, and it can produce visually significant directional artifacts. Since I am working with two dimensions, the first point is moot, but having the potential to expand the project efficiently into higher dimensions is beneficial. Fortunately, Perlin developed another noise algorithm in 2001 which he called Simplex Noise. Simplex Noise effectively alleviated the key two problems that plagued Perlin Noise, thus producing better scalability and cleaner noise images. For these reasons I chose to go with Simplex noise for my implementation.

There are four main steps involved in implementing Simplex Noise:

- Coordinate Skewing
- Simplicial Subdivision
- Gradient Selection
- Kernel Summation

## Coordinate Skewing and Simplicial Subdivision

When using Simplex Noise the aim is to pick the simplest shape that can be repeated to fill the entire coordinate space for a given N-dimension. For two dimensions, a triangle is the shape with the fewest number of corners that fits this criteria. Equilateral triangles are traditionally used, with several of these side-by-side creating a grid of rhombi. These rhombi can be thought of as regular axis-aligned hypercubes that have had their coordinates skewed. Therefore to determine which triangular simplex a coordinate point falls into we need to skew our coordinate grid to form a grid of regular axis-aligned hypercubes. We can then examine the transformed coordinates x and y to determine which hypercube the point falls into. Furthermore, by also comparing the magnitudes of x and y we can distinguish if the point falls into the upper or lower triangle of the square. Figure 1 depicts this process.

Once we have determined which simplex the point falls within, that simplex is composed of the vertices of an ordered edge traversal. If the coordinate falls into the lower triangle the edge traversal reads `(0,0) -> (1,0) -> (1,1)` while if it is in the upper corner it reads `(0,0) -> (0,1) -> (1,1)`.

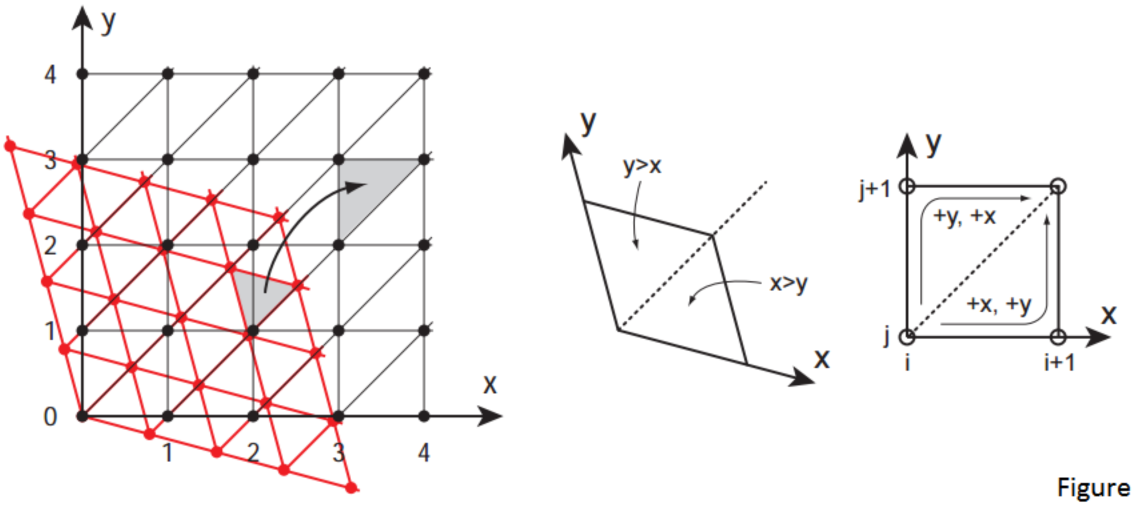

Figure 1

## Gradient Selection and Kernel Summation

Each simplex vertex is then added back to the un-skewed base coordinate and hashed into a pseudo-random gradient direction. For this project I used a permutation table to implement the hash.

Unlike Perlin Noise which involves sequential interpolation along each dimension, Simplex Noise instead uses straight summation of the 'contributions' from each vertex corner. The contribution of each vertex corner is the multiplication of the extrapolation of the gradient ramp and a radially-symmetric attenuation function. This radius attenuation is carefully selected so that the influence from each corner vertex reaches zero before crossing a boundary to the next simplex.

The influence from each corner can be visualised as depicted in Figure 2.

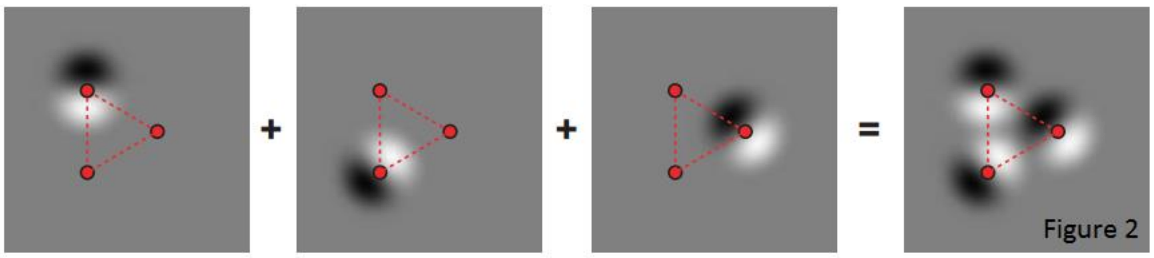

Figure 2

This process can be applied to each pixel in a grid to generate a single image of Simplex Noise (see Figure 3). My implementation of Simplex Noise can be found in the 'noise.h' header file.

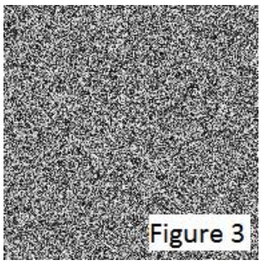

Figure 3

## Fractional Brownian Motion – noise.h

A single image of Simplex Noise is a good start, but it is also difficult to differentiate from purely random noise. The noise currently has a very high frequency which results in it jumping quickly between values for each pixel. We can produce a smoother looking image by reducing the noise's frequency. Since frequency = velocity / wavelength and there is no convenient way to reduce the noise's wavelength we need to decrease the velocity instead. This is achieved by scaling the x and y values we give the noise function by a small scale value. Figure 4 shows the result.

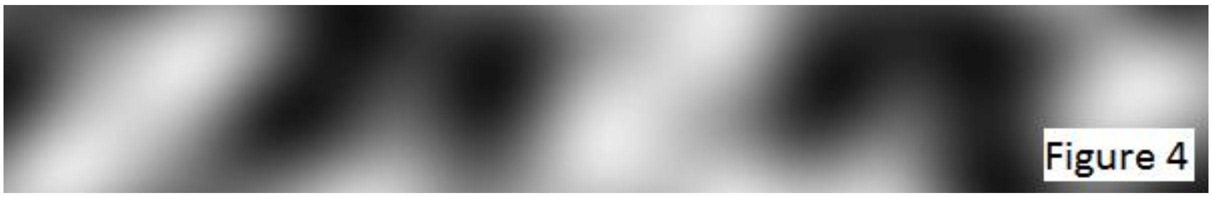
Figure 4

While an improvement, it lacks variation. This is where the Fractional Brownian Motion (fBm) method comes in. We generate multiple noise 'octaves' decreasing the amplitude and doubling the frequency of each successive octave. We then sum all the octave together, take an average, and normalise the resultant values. In my implementation of fBm I use a variable 'persistence' to control the rate at which the amplitude decreases. By using a scale value of 0.007f (the initial frequency), and a persistence value of 0.5f, Figure 5 is achieved over a process of 16 iterative octaves.

Figure 5

This concludes my noise class in 'noise.h'.

## Terrain Generator – terrain_generator.h

My terrain generator implements a three-step process to create the two dimensional terrain output. It begins by generating a 256x256 terrain grid of two dimensional tile objects, where each tile represents a single block of ground.

I then generate a 'height line' for my terrain by running a single one dimensional line of 256 x coordinates through my Simplex Noise function. Instead of using the results (that are normalised between 0 and 255) as luminance values, I instead use them as the highest y coordinates in my 256x256 grid of terrain for their given x coordinate.

Finally I generate noise values for all blocks less than or equal to my terrain height line, using a threshold value to determine if the block should be visible or not. Here I also ensure that all blocks with y coordinate values greater than that of the height line value for their x coordinate are also disabled from view.

I have included some interactivity in the generator application by allowing the number of octaves, luminance threshold value, scale/frequency value, and the persistence value to be incremented/decremented to produce a newly drawn graph. The user can also generate a new height line for the terrain. All controls keys can be found in the bottom left hand corner overlay when running the application.

There is also some commented out Windows-only code used to generate images of both the simplex noise and height line in a Windows console window.

## Future Development

Unfortunately due to time constraints and working alone, there are a number of features I did not get to implement in my generator. Next I would focus on running a gradient function vertically downwards through my terrain. This would slowly increase the size and frequency of the caves, thus eliminating the gaping holes currently found on the surface of the terrain. It is also important to implement more cave structures generated using different noise/variables to add more variance to the terrain.

Further improvements would focus on the implementation a player character the user could control to 'mine' away blocks of the terrain as well as implementing a system to generate different kinds of terrain biomes.

## Youtube Video

Here is a link to a demonstration video of my application:

https://www.youtube.com/watch?v=VRXcbeJUsYE

## References

https://en.wikipedia.org/wiki/Simplex_noise

http://webstaff.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

http://cmaher.github.io/posts/working-with-simplex-noise/

http://accidentalnoise.sourceforge.net/minecraftworlds.html