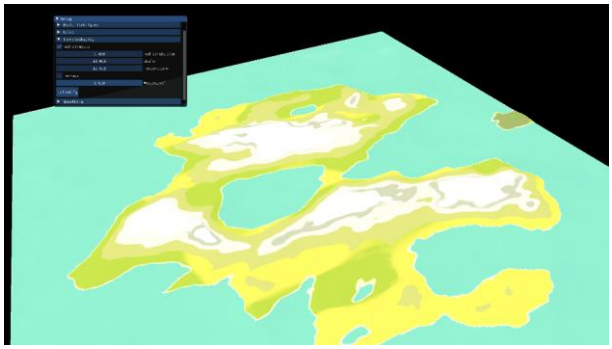
Procedural Generation Report

# Terrain Generation

Rowan Stockton - 1600490



## Outline

The application can generate varying types of terrain using different techniques be it applying different types of noise, faulting terrain, or particle deposition to carve out terrain. Once the terrain is generated it can also be stylized to shape the terrain more aesthetically. The terrain can have biomes applied to it to change the colour and look of terrain at certain points and can use several of the techniques for the terrain to create rivers as well.

## Features

### Noise Generation

#### Perlin Noise

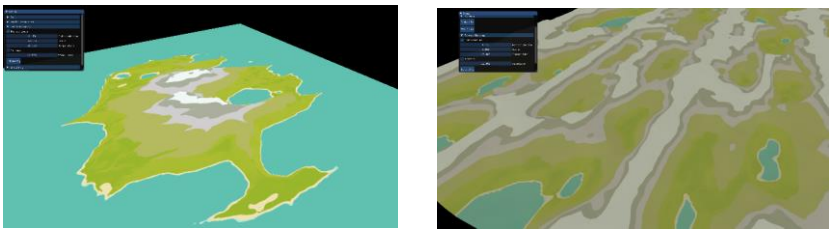
Using Perlin noise we can sample a noise map generated from Perlin noise to either generate terrain or add detail to existing terrain.

#### Fractal Brownian Motion (fbm)

We can also layer the Perlin noise to create more detailed noise maps to generate more detailed terrain. This layering of noise is known as fractal Brownian motion.

#### Ridged Noise

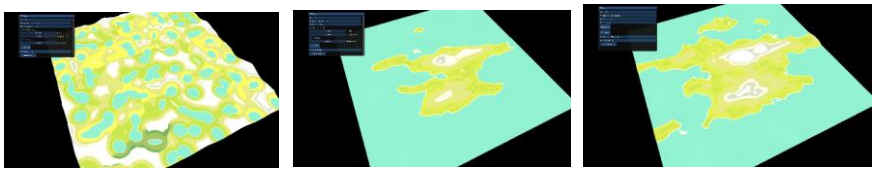
We can use other types of noise maps to also generate terrain. Ridged terrain like it sounds generates ridges that look like mountain ranges. Using this we can generate more mountain like features with sharper peaks.



Ridged Noise applied to height map and modified to be island like

#### Worley Noise

Worley noise works can make smoother ridged like maps but can be used as a sort of seasoning to slightly displace the map and add slight features.



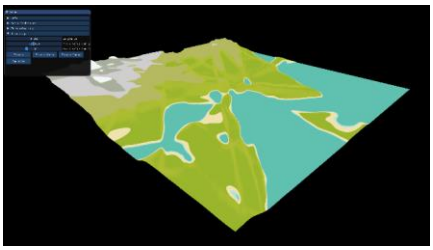
Worley Noise

Worley Noise being applied to a generated island

### Terrain shaping techniques

#### Faulting

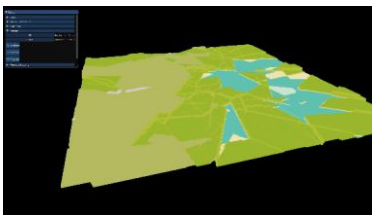
We can create fault lines and we can raise and lower the terrain on each side of the fault lines. Applied multiple times you start to tend to generate higher areas and lower areas creating block-esque terrain. The amount that each side of the fault line is shifted can be changed.



Map created through faulting and smoothing

### Voronoi

Much like Worley noise random points are selected but instead of values being determined by distance, regions are determined by the distance to each point. Then like faulting each region is then raised or lowered by a random value within a range creating a similar effect to faulting but quicker, however, it is also messier. The number of regions and the max faulting amount can be edited.



Terrain generated by faulting with Voronoi regions

### Smoothing by average

Once we have generated terrain sometimes it can be blocky or too sharp in some areas so we can average each point by getting the height value of the surrounding points and averaging that out thus smoothing the terrain.

### Slope Smoothing

Sometimes we don't want to lose the height values and instead just make ridges smoother. Slope smoothing works by checking each surrounding pair of points and finding which pair of points has the highest difference in height. Then height of the current point is set to the average of those two points. We can also specify a height range to apply this smooth sloping so we can smooth lower down areas like beaches and plains while keeping sharp ridges and drops of mountains.

Same Voronoi map smoothed by average and then lower regions smoothed by slope

## Biome Generation

The colour of the terrain should change depending on what kind of biome we're in, where higher up areas are usually colder and lower down areas warmer with also varying moisture. We can use a 2D texture to determine colour values for the biomes and depending on the height and moisture of each point in the terrain we can sample that texture for the colour value. We can generate a moisture map using a noise function like Perlin noise.

## River Generation

Very basic looking rivers can be generated using a few different methods to give a bit more character to the terrain.

### Particle deposition

This technique is meant to simulate the way a water would carve out a path by moving from a high point to a low point. A random point on the map is selected and added to a list. Then each surrounded point is checked and the one with the lowest y value is the next point added to the list and that one is then checked. This continues until in we are in a dip or at sea level simulating the path a "particle" would take down the terrain. Then the list is iterated through changing the moisture value and slightly carving out the path the "particle" took down the terrain to create rivers.

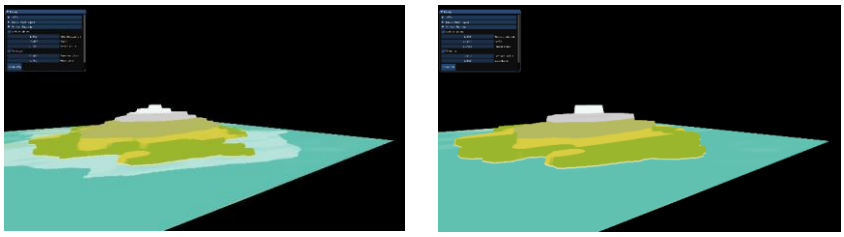
### Ridged noise

We can generate rivers by taking only the highest values of a ridge map and using those values to carve out the rivers and then change the corresponding moisture values.

## Terrain Shaping

### Terracing

We can stylize the terrain by clamping the height values in intervals creating a terraced like effect. We can also choose a value for the intervals changing how terraced the terrain is.



Terraced Terrain with different terrace values

### Redistribution

By applying a power to each point's height value, we can create more prominent mountains and valleys.

### Island Shaping

Multiplying the height based on its distance to the centre to raise terrain in the middle and lower at the edges to create and island effect.

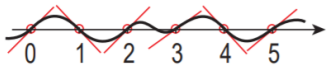
# Code/Implementation

## Noise generation

Noise can be used to generate and simulate randomness that is found in life, objects, etc. and is a useful tool in procedural generation. One of the most famous noise algorithms is Perlin noise.

### Perlin Noise

Noise can be generated by just randomising numbers and calling it a day, but this kind of noise doesn't have much cohesion and doesn't look very good, it looks random. Perlin noise creates smoother noise maps that look more natural. Perlin noise works through having a set of pseudo-random gradients positioned at even spaced points in space and then interpolating between these values to generate smooth slopes. To explain we'll look at 1D Perlin noise, but the method can be applied to the nth dimension. Given the point x between two integer points the value of x is interpolated based on the two integer values. The interpolation that is applied is not linear with distance as this would mean that the derivative of the noise function should also be continuous at the integer points. So instead a blending function is used that has zero derivatives at its end points. The blend function that was used to decide the values between the integer values is of the form $f(t) = 6t^5 - 15t^4 + 10t^3$. The result is something along the lines of:




In this diagram we can see pseudo random gradients defined at each integer and the values between each interpolated between the gradients. This blend function has a perk in the fact it has a zero second derivative at its end points meaning it could also be used for surface displacement and bump mapping.

Before it was mentioned that Perlin noise uses pseudo random values, this is so we can create the same noise maps given the same inputs but we still want it to seem random. We can have a set of unit length gradients in different directions and the position in the grid can be used to hash one of these values. (Gustavson, 2019)

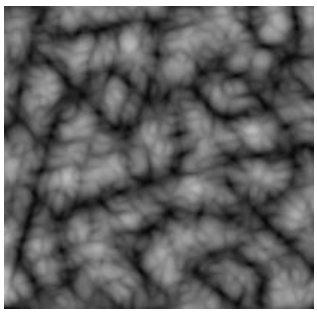
In the application a noise class is created with a function to sample this Perlin noise with predefined gradient values. Within this class there's a function to sample noise depending on the x and y position given to the sampler.

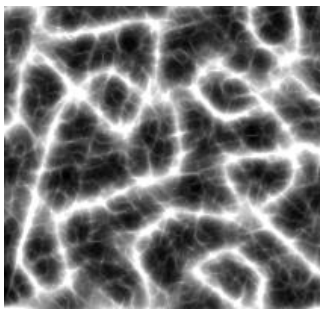
### Fractal Brownian Motion

Sampling noise is good but better noise can be created by sampling it multiple times and layering it. These layers are called octaves and depending on the number of octaves used determines how fine the noise becomes. If we keep the frequency the same however it won't create any significant changes to the noise map. Instead we can increment the frequency in regular steps with each octave, known as lacunarity. If the frequency is increasing the noise will become messier and appear more random especially if they are contributing the same amplitude, so to account for this the amplitude is also decreased in a regular increment with each octave, this is known as gain. This process is called Fractal Brownian Motion (FBM) due to the fact if you zoom in on the noise map it still looks as detailed up to a point mimicking fractal. (Lague, 2019) (Patricio Gonzalez Vivo, Fractal Brownian Motion, 2019)

## Ridged Noise

Instead of just applying the value retrieved from sampling Perlin noise the absolute value is taken thus getting rid of all the dips and having only high points. Then the value is inverted to get a ridge like height map. To create sharper peaks, we can square the height values, this is the same as applying the redistribution value mentioned later. This can be combined with fbm to create a more detailed mountain range. (Patricio Gonzalez Vivo, Fractal Brownian Motion, 2019)



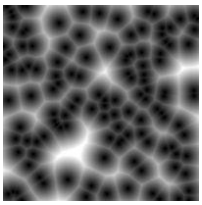


Absolute Values and inverted Sharpened of Perlin noise (Patricio Gonzalez Vivo, Fractal Brownian Motion, 2019)

**Insert image of ridges from terrain**

## Worley Noise

Worley noise works through generating a set of random points on the map. Then each height value is determined based on its distance to its nearest point. This creates a less prominent version of ridged noise but can be applied over terrain to create more interesting terrain. Worley noise is mainly used in cellular generation however and can create some rather interesting images. (Worley Noise, 2019) (Patricio Gonzalez Vivo, Cellular Noise, 2019)

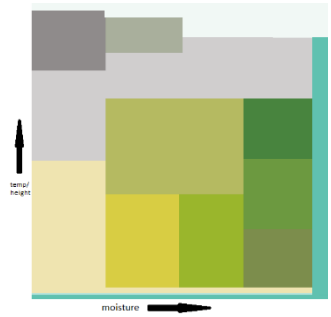


Worley Noise Map (Worley Noise, 2019)

## Biomes

To have the map look more interesting we can introduce biomes to colour the terrain differently depending on certain variables. To achieve this, we can make a biome map which we can sample using values to determine the x and y coordinate thus deciding the colour at that position. Having the Y value determined by temperature and the X value determined by moisture as these two variables are what generally defines biomes on earth. For simplicity we can have temperature defined by height where higher up regions are usually colder than those close to sea level and we can scale the value by a temperature variable. For the moisture we create a new texture which stores a noise map where the values correspond to the moisture in that region. This texture is created using DirectX's Texture2D type which is filled with the moisture data before being converted into a shader resource view. (Luna, 2011). This resource view can then be passed to the pixel shader

and using the tex coordinates of the terrain we can sample the moisture map at that point and then use the value gathered from that to get the x coordinate. To generate this texture Perlin noise is reused with a low frequency to create larger biome blobs. (Amit, Making maps with noise functions, 2019)




Texture from which the terrains colour is decided upon taking height as y value and moisture as x

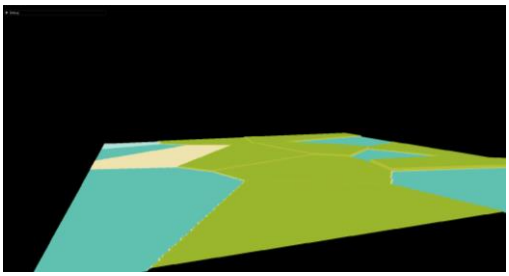
## Terrain Shaping Techniques

### Faulting

To create fault lines two points are randomly selected and the vector between them is then calculated. Looping through each point in the terrain we can calculate which side of the fault vector they lie on and raise / lower them accordingly. To calculate which side a point lies on we can calculate the cross product between the current point and the fault vector. The cross product will result in a vector which is either positive or negative in the y axis depending on the point's position in relation to the fault vector. Now we can calculate the side on which a point lies we can raise and lower them appropriately.

### Voronoi region

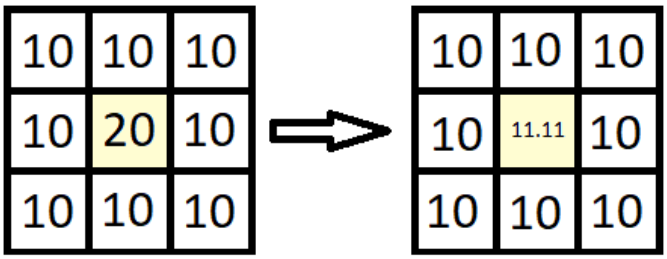
Like Worley noise a set of random points are selected, and a height value is applied to each point between 0 and the maximum fault value that has been selected. Then each point in the terrain is looped through and the closest Voronoi point to that point is found. Once found the current point's height value is set to that of the closest Voronoi point creating a blocky terrain.



Terrain shifted through Voronoi regions

### Smoothing by average

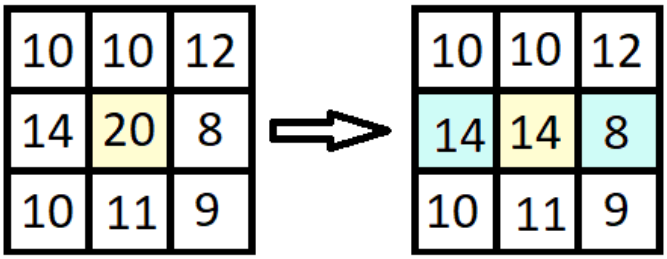
We may wish to smooth out the terrain and we can do that in two ways. First of which is smoothing by average. We can loop through each point in the terrain and find the height values of the adjacent points. By averaging out these points we can smooth out the terrain. (Foster, 2019)




Averaging out points based on surrounding cells

### Smoothing by slope

Sometime slopes may be too steep or sharp, but we mainly wish to keep the height of the terrain as it is. Instead we can smooth by slope in a similar fashion to how we smooth by average. Instead of getting the average height of all the surrounding points we can find the two points which have the greatest difference in height to the current point.



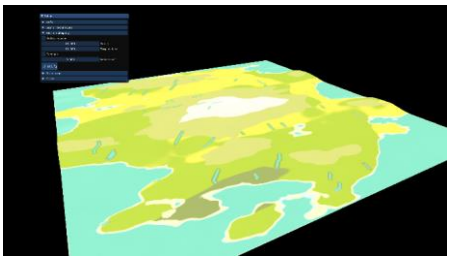

Averaging out points based on two opposite points with highest varience

In case we don't want to smooth out any mountains and keep their ridges we can designate a height range to smooth slopes in. While looping through the points to average them out we can see if they are within the range or not that we wish to smooth. If not, then no need to smooth and we can move onto the next point.

## River Generation

### Particle deposition

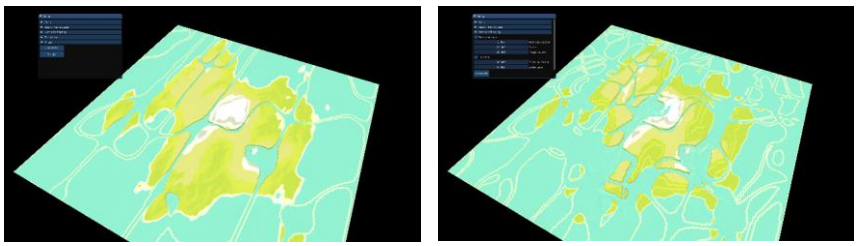
One method for creating rivers is to simulate an erosion like effect. Since rivers naturally flow from high points to low points we can drop "particles" onto the map. First, we pick a random point on the map to drop a particle and store its position. After dropping a particle, we can check the height of adjacent points to find the one with the steepest negative gradient as we want to flow downwards and store its position as well. We then check its adjacent points for the steepest negative gradient until either we hit sea level, or the point has no adjacent lower points. We can then loop through these stored points changing the moisture value so when textured the river will show blue and carving out the terrain slightly.



River Particles

### Ridged and Worley noise

We can also use the maps we generate from Worley and ridge noise to create river like patterns. By taking only the highest values we can apply that and edit the moisture values to create river like effects. Ridged noise especially tends to create more of a river like look but mainly it can be hard to simulate rivers going downhill with this technique.



River ridged

## Terrain Shaping

### Terracing

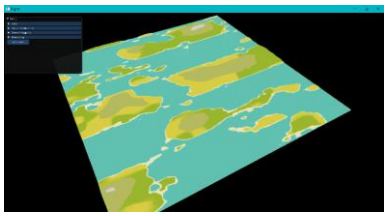
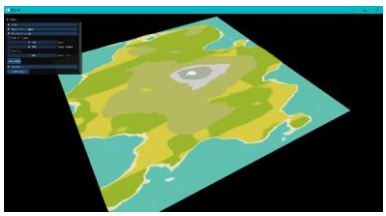
To stylize the terrain we can apply the following formula to the landscape to create a terraced terrain: $Height = \frac{Round(Height \times n)}{n}$ What this does is clamp all height values to the nearest step defined by n. This expression is applied in the vertex shader for speed and to allow dynamic changing of the n value without losing any exact data about the height map. (Amit, Making maps with noise functions, 2019)

### Redistribution

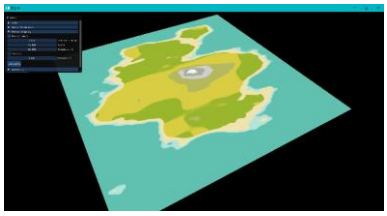
Like mentioned above we can raise the height to a power to create more prominent mountains and flatter valleys. This is through applying the equation $Height = Height^n$ where n is a decimal value that can be changed, we can accentuate the height map. Along with terracing this is also applied in the vertex shader for speed and to not lose any data about the height map. (Amit, Making maps with noise functions, 2019)

### Island Shaping

When generating a height map with noise the values are within the range -1 to 1 or 0 to 1 depending on the noise function. However, this is consistent throughout the height map but perhaps we want to create an island, well the values can be shifted to raise the middle of the map to create this island effect. By applying the equation $Height = (\ 1 + height - d)$ where d is the distance to the centre of the map, to every point we end up raising the middle.

Map before and after applying island equation



Island after redistribution, water level and scale edited

## Analysis

Looking at the code there could be several improvements, first of which the terrain class is. Working from the rasterteks terrain class that was provided in week 1 the rest of the application was built upon this class Looking back on it though there are a lot of functions in the one class that do similar things and have different names.

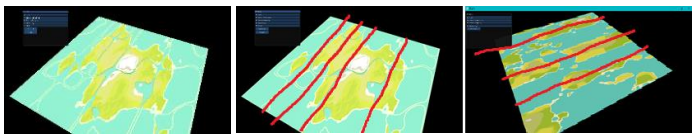
All the “generate map” functions mainly do the same thing of looping through the map and applying the value from a noise map at that point. This could be improved by having one function which takes the noise function as a parameter so that it loops through the height map and then applies the relevant noise function.

```
Func noisemap(other parameters, nosie function())
{
        Loop through height map
                Current height point = value returned from noise function()
}
```

This would clean up the class slightly and make it more functional as a new function wouldn't need to be made if other height manipulation functions were to be added.

Another point is inconsistencies in capitalisation and variable naming due to the format in the naming conventions rastertek uses and the format used in the implementation.

Perlin noise class could be changed to have static functions since they perform purely mathematical functions and don't have any other functionality other than returning values based on the noise map. This means if any other part of the program wanted to use any noise function it wouldn't need to be set up as an object within that part of the program, instead its function can just be called much like the math library in c++. On top of this the noise function would be revisited as it tended to generate features along lines which isn't a desired effect especially as it shouldn't do that.



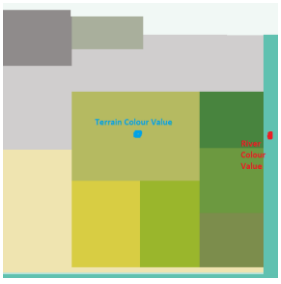
Lines in noise maps

## Reflection

Upon reflection sticking to rastertek may have been a poor choice. Due to being unfamiliar with the framework and having to try understanding what was already implemented in the example probably hindered progress at points. If the project was to be done again the framework provided by Paul last semester would be used as it was already a familiar framework with a bit of functionality already built for last semester's project.

One other problem is with the project is lighting, since we apply all these alterations to the map in the vertex shader a lot of the normal data, we pass in becomes incorrect. To fix this the normals could be recalculated in either the vertex shader or later in the geometry shader. As we know the calculations, we are performing at each vertex in the vertex shader we can recalculate the normals. Or we can get the adjacent vertex values from the geometry shader and use them to calculate the cross product between the vertices and calculate the new normals that way.

In terms of the procedural generation there could be a few improvements especially with the rivers. First of which is the texture filtering which currently is what create those strange lines with the rivers. The problem is the colour interpolating between the end of the colour map to the colour of the terrain next to the river to create a smoother texture.




The difference in sampling the colour values between river and adjacent terrain

With the particle deposition method working to an extent but you don't get the windy curves rivers usually have. And ridged noise ignored gravity too much for it too appear natural. Some alternative approaches could be taken instead. One of these methods is the approach Minecraft takes which is that all the rivers are at sea level and just carve out a path in the terrain. This could be achieved by having a route randomly generated through randomly picking a point and the next point would be decided by the current's points position. Then connecting these points and smoothing to create a windy pattern. All that's left is to carve out this route on the map creating a river.

Diagram of the method mentioned above

To explain the diagram, each square around each point is the zone in which the next point can be randomly placed. The orange line is the direct path between each point and the blue line is meant to be to what that path would look like when smoothed. The path would probably be smoothed using splines to create the nice curves thus creating a more realistic river.

Another approach to generating rivers would be the use of Voronoi regions. The map could be split up into Voronoi regions with each Voronoi region being given a height based on the average height of the points within the region. Using these height values we can generate rivers going along the edges of the Voronoi regions going from high regions to low ones while applying noise to the edges to create less linear looking rivers. (Amit, Polygonal Map Generation for Games, 2019)

One final approach is with Voronoi regions again but instead of having the river run down the edges you can get the height values of the edges and calculate the entry and exit points of the Voronoi region to dictate how the river would flow (Amit, Procedural River Growing, 2019).

## Reflection

Several things were learnt through the process of creating procedurally generated terrain mainly about procedural generation in general and DirectX.

It took a good while to get the moisture map working, probably longer than should have been spent on it but through that process more was revealed about DirectX 11 itself. Originally the plan was to write the texture map to an image and then load that image in as the functionality was already implemented for loading in images. However, writing images was harder than expected and upon with some consultation with a lecturer (Gaz) it made more sense to not write an image, save it and load it back in and that's when DirectX's texture2d was discovered. This is a shader resource that can be filled with data and then passed to the shader to be used. It's sadly not as simple as that though but tackling this problem gave greater understanding of shader resources and writing data.

Noise can be applied to so much, be it to lines to add variation, to images to add aesthetic or to terrain to add feature or just the whole map in general. Noise is so useful and there's so many types of it like. Noise can be split up into colours ranging from white to red noise depending on their frequency. This may not seem important but we can shape our noise accordingly to what we want,

for instance the most common type of noise to appear in natural things is pink noise so if we want to generate something natural we may wish to use frequencies in our noise maps that give us pink noise values. There's quite a bit to go into with noise but Red Blob games does a good article on it. (Amit, Noise, 2019)

Domain warping is this cool thing that was planned to be implemented but generating textures in direct X took too much time sadly. Basically, domain warping works by sampling noise and using the values gained from that to sample noise and this can be continued however much you want to create some awesome effects.

## References


Amit, P. (2019, Feburary). *Making maps with noise functions*. Retrieved from Red Blob Games: https://www.redblobgames.com/maps/terrain-from-noise/

Amit, P. (2019, March). *Noise*. Retrieved from Red Blob Games: https://www.redblobgames.com/articles/noise/introduction.html

Amit, P. (2019, March). *Polygonal Map Generation for Games*. Retrieved from Red Blob Games: http://www-cs-students.stanford.edu/~amitp/game-programming/polygon-map-generation/

Amit, P. (2019, March). *Procedural River Growing*. Retrieved from Red Blob games: https://www.redblobgames.com/x/1723-procedural-river-growing/

Foster, N. (2019, Feburary). *Simple terrain smoothin*. Retrieved from nic-gamedev: http://nic-gamedev.blogspot.com/2013/02/simple-terrain-smoothing.html

Gustavson, S. (2019, March). *Simplex Noise Demystified*. Retrieved from http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

Lague, S. (2019, March). *Procedural Landmass Generation (E03: Octaves)*. Retrieved from Youtube: https://www.youtube.com/watch?v=MRNFcywkUSA

Luna, F. D. (2011). 3D Game Programming with DirectX 11. In F. D. Luna, *3D Game Programming with DirectX 11* (pp. 605-606). Boston: Mercury Learning and Information.

Patricio Gonzalez Vivo, J. L. (2019, March). *Cellular Noise*. Retrieved from The Book Of Shaders: https://thebookofshaders.com/12/

Patricio Gonzalez Vivo, J. L. (2019, May). *Fractal Brownian Motion*. Retrieved from The Book Of Shaders: https://thebookofshaders.com/13/

*Worley Noise*. (2019, April). Retrieved from Wikepedia: https://en.wikipedia.org/wiki/Worley_noise