CS238: Procedural Generation Algorithms for Terrains

Jennifer Ma

May 29, 2014

1 Introduction

It has been 32 years since Ken Perlin won an Academy Award for his development of Perlin Noise while working on the movie Tron. Since then, not much has changed in the world of procedural terrain generation. One reason is that the inherent "randomness" of procedural techniques hinders its implementation in the real world. Perlin Noise allows the user some degree of control, which is immensely useful when designing worlds, and games in which the user has to have a coherent and non-volatile experience. However, Procedural Generation Techniques are fast and most importantly easy to understand and use. I implemented the following algorithms by using heightmaps. They are simple 2D arrays of values that represent the heights of the terrain. Limitations of this approach include the inability to produce caves or overhangs.

All of the computing was done on a Intel Core i7, using one core.

Intel Core i7-4750HQ Crystal Well @ 2GHz

Graphics	Intel Iris Pro Graphics NVIDIA GeForce GT
# of Cores	4
L3 Cache Size	6 MB
L4 Cache Size	128 MB

2 Procedural Generation Techniques

2.1 Diamond-Square

Diamond Square is a step up from Midpoint Displacement. By including a step that involves diamond-shaped squares, hence its name, it reduces some of the artifacts that can appear with midpoint. It starts by seeding the four corners of a map in which a new point will have the average of its squares or diamonds plus a random offset. It is still slightly flawed since it still requires $2^n + 1$ values for the terrain to have an equal amount of randomness distribution.

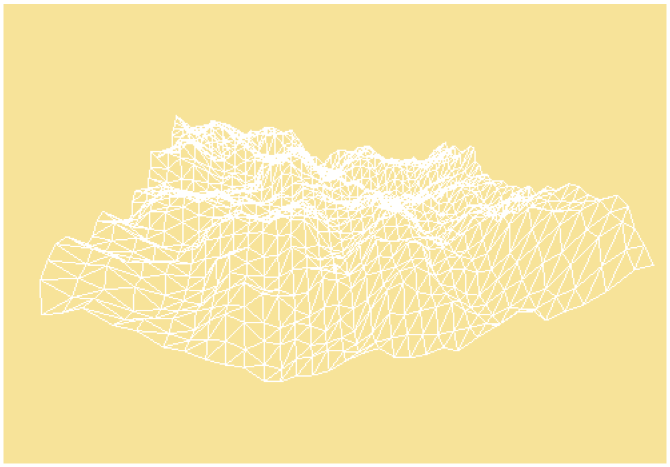

Figure 1: Diamond-Square with size of 33x33

Generation Algorithm	257	513	1025
Diamond-Square (sec)	0.00255479	0.01000666	0.02155986
Midpoint (sec)	0.00290157	0.01139190	0.04575593

Table 1: Diamond-Square vs Midpoint for sizes of $2^n + 1$

2.2 Midpoint Displacement

Midpoint is arguably the simplest of all the techniques. Then create a middle point by averaging the four corners and added a random value. Create further midpoints of those four sides by averaging the two corners each of the sides is between. Rinse and repeat.

```
float total_corners = height(i, j)
+ height(i + incr, j)
+ height(i + incr, j + incr)
+ height(i, j + incr);
midpoint[(i+incr/2)][j+incr/2] = total_corners/4;
midpoint[(i+incr/2)][j+incr/2]+= random(range);
```

It also produces the most visual artifacts because of the clunky squared shaped generation path. It also requires $2^n + 1$ terrain sizes and rectangular maps take a little bit more time to implement.

Both are amazingly fast but at larger sizes, diamond-square starts to edge out a little bit but not enough to completely discount Midpoint since the standard deviation at that size is 0.002383 for Midpoint and 0.0014637 for Diamond-Square.

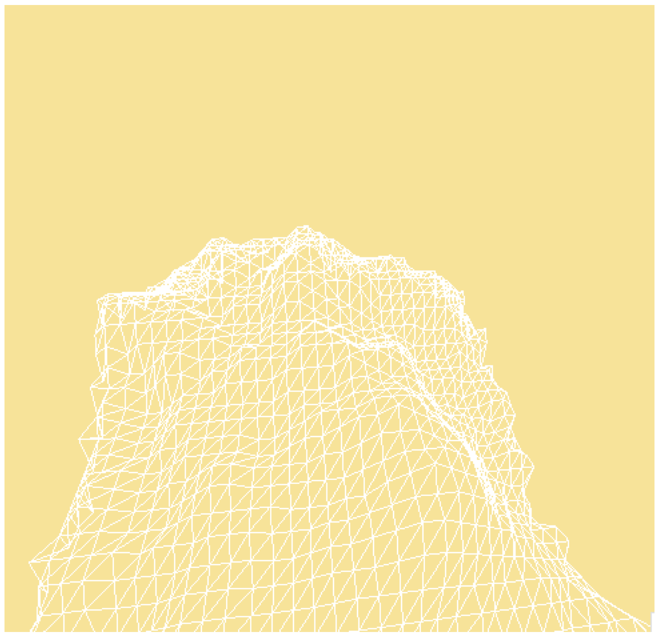

Figure 2: Midpoint with size of 33x33

2.3 Perlin Noise

Perlin Noise is used for generating coherent as well as pseudo-random noise within a map. I used the 2D version of Ken Perlin's award winning code. We have four points that surround a fractional point (non-whole xy) value. We take those four points and assign it to a pseudorandom gradient. Ken Perlin solved the issue of how to maintain control of the pseudorandom gradient by building a table of numbers between 0 and 255, each number unrepeated. At each point, we create a vector that goes from the point to the original fractional xy value. By calculating the dot product we can define the influence of each gradient. Ken Perlin introduces the s-curve, which exaggerates values so that if values are close to zero, the value becomes extremely close to zero. Finally, we can interpolate to get a weighted sum and obtain a final value for each pixel in a grid. [1]

```
//dot products that get the noise from each corner.
float n1 = dot(gradients[g1], fx, fy);
float n2 = dot(gradients[g2], fx - 1.0f, fy);
float n3 = dot(gradients[g3], fx, fy - 1.0f);
float n4 = dot(gradients[g4], fx - 1.0f, fy - 1.0f);
//fade value calculated by Ken Perlin
float sx = fx * fx * fx * (fx * ( 6 * fx - 15) + 10);
```

[1]http://www.mrl.nyu.edu/~perlin/doc/oscar.html#noise

Algorithm	Time (s)	SD (s)
Perlin Noise	3.48676574	0.04743041
Midpoint	0.19477008	0.00499284
Diamond-Square	0.10201522	0.00463706

Table 2: 2049x2049

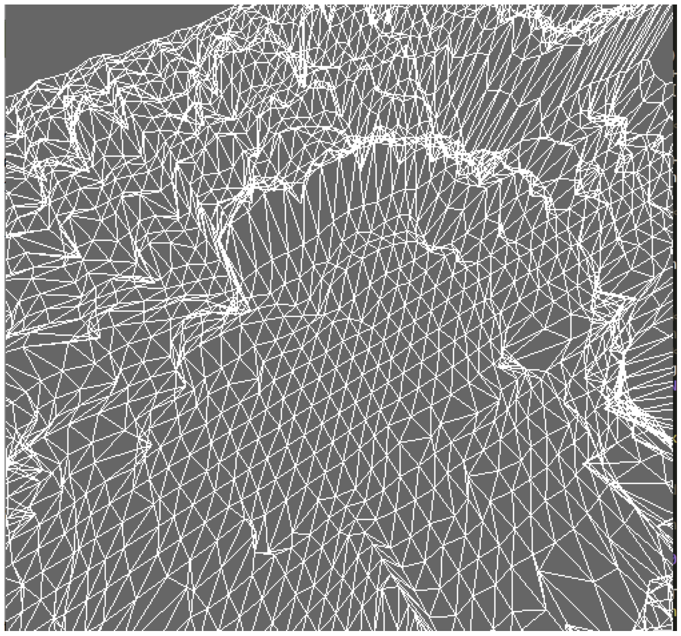

Figure 3: Perlin Noise with size of 55x55

```
    float sy = fx * fx * fx * (fx * ( 6 * fy - 15) + 10);
```

It is comparably fast with less artifacts and impressive detail. However, Perlin wrote up a new and improved form of this algorithm called Simplex Algorithm that works incredible speeds above 3 dimensions. Perlin Noise is roughly $O(2^n)$ while simplex is $O(n^2)$.[2]

2.4 Fault

The fault algorithm is the simplest one to understand and implement, even simpler than Diamond-Square. We create a line that divides the terrain not symmetrically or asymmetrically but with enough randomness to seem natural. Using cos(random) and sin(random) creates this effect. Then we iterate for each point on the map and move the heights by certain ranges. Points on one side

[2]http://www.noisemachine.com/talk1/32.html

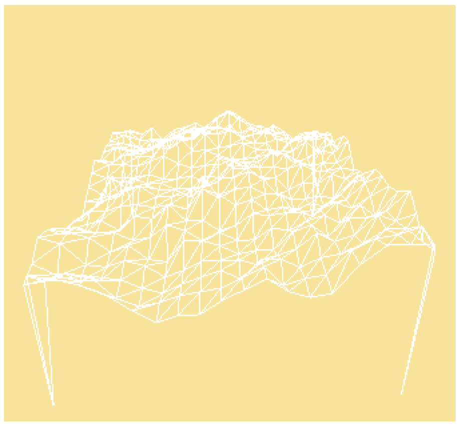

Figure 4: Fault Line Algorithm with size 30x30

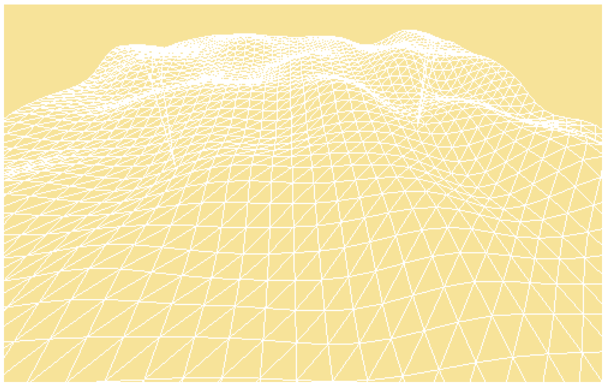

Figure 5: Fault Circle Algorithm with size 50x50

of the line will be increased and points on another side will be decreased. If we had cut the map randomly at the origin we would have ended up with a mountain on one side and a basin on another. A simple riff on dividing the terrain with lines, we can also divide the terrain with circles. This is a fun algorithm but is not really useful for controlling artifacts in terrain. However, we can quickly generate contrasting landscapes such as grand mountains and deep valleys through choice placement of the dividing line. Generating rolling hills with the circle algorithm is incredibly fast and can be useful in certain cases.

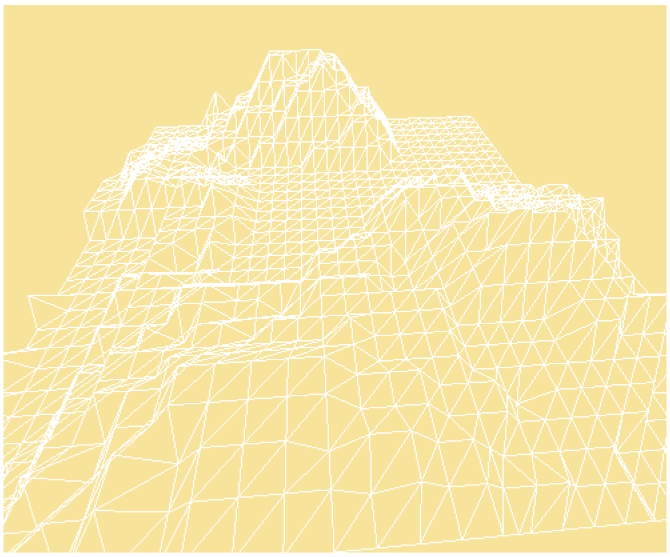

Figure 6: Particle Deposition with size 30x30

Generation Algorithm	100	500	1000	2000
Fault (sec)	0.01863903	0.4409342	1.75177268	7.0642421
Particle Deposition (sec)	0.00012555	0.00016218	0.00019312	0.00020136

Table 3: Fault vs Particle Deposition

2.5 Particle Deposition

Particle Deposition as stated in Game Programming Gems[3]was originally used for creating volcanic ridges but it can be modified for regular terrain generation. Particles can easily and naturally simulate the effect of thermal, wind and water erosion. We first step across a heightmap and seed each point. We then check heights of adjacent cells. If a lower height is found, then we move to that position. Repeat until there is no lower point.

The main point of attraction for particle deposition is its blazing speed. It can create a terrain map in under 1 second for most map sizes. However its visual artifacts are a major deterrent to it being widely used. As seen below, smoothing is an extremely nice technique that adjusts the height according to its neighbors. By iterating down both rows and columns we can achieve a nice rounding effect on the mountains without a lot of the jaggedness that makes particle deposition difficult to work with.

[3]DeLoura, Mark A.. Game programming gems. Boston, Mass.: Charles River Media, 2000. Print.

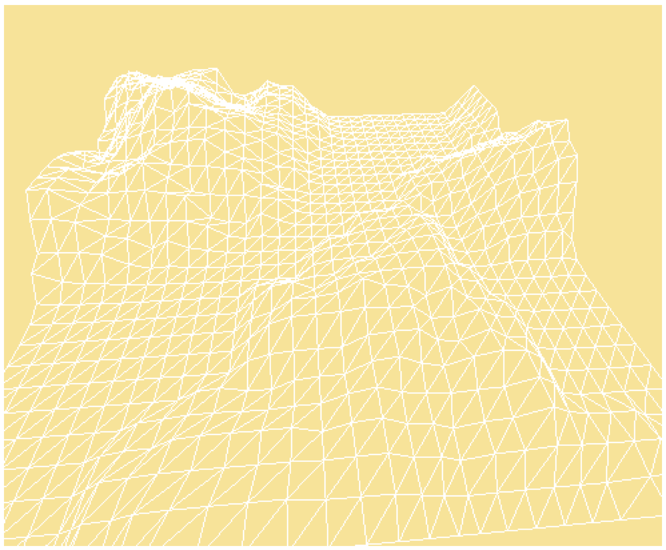

Figure 7: Particle Deposition with Smoothing

Water Erosion (s)	100	500	1000
5000 iterations	3.6853712	92.8556331	355.116676

Table 4: Water Erosion Times

2.6 Water Erosion

Erosion is incredibly slow, having to run for thousands of iterations to have a believable effect. One possible path I could take is to try to port calculations to the GPU and see if that increases speed somewhat. I started my heightmap by using the fault algorithm because it required only one function. For each cell I added a certain amount of rainfall, which was represented by adding a small float to a newly initialized heightmap. Then take away a small bit, simulating erosion. Finally we have to move water that is uphill to water, downhill since water flows downwards.

I tried to account for errors in the computations by creating an empty file and timing how long it took to execute it. An empty file takes .0000001 seconds to run, a negligible amount of time.

In terms of averages and standard deviations. I ran all of my files for a minimum of 1000 times to maintain a low standard deviation. The standard deviation is small around the average, denoting hopefully accurate numbers.

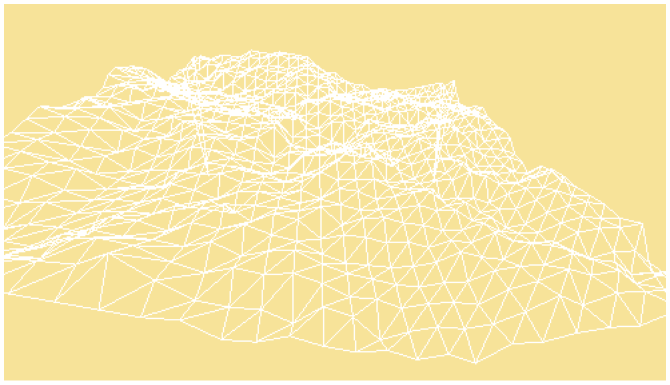

Figure 8: Before Erosion

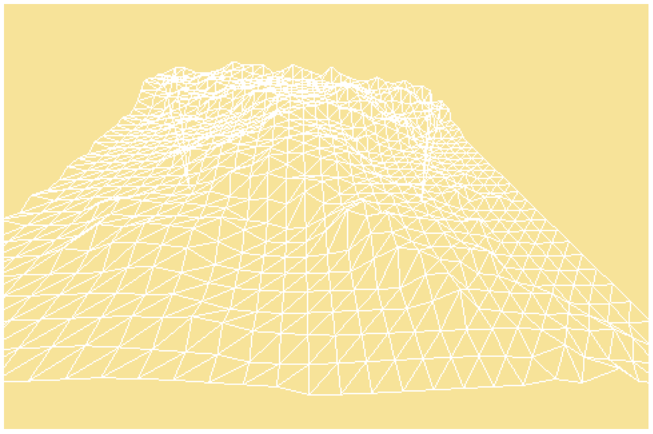

Figure 9: After Erosion

Algorithm (500x500)	Time (s)	SD (s)
Perlin Noise	0.21890201	0.00581298
Fault	0.44093426	0.00855291
Particle Deposition	0.00016218	0.00002107

Table 5: Error Data

3 Error Analysis

I accounted for errors in the computations by creating an empty file and timing its execution time. An empty file takes 0.0000001 seconds to run. The number is so negligible that it can be discounted.

In terms of averages and standard deviation. I ran all of my programs for a minimum of 1000 times. Here is a sampling of standard deviations for the different algorithms.

4 Conclusion

Procedural Generation methods are extremely popular because it is fast and creates semi-realistic worlds. I used heightmap generation while the shift now is to building cityscapes which procedural generation with heightmaps cannot do as well. Ways I can improve my algorithms is to try parallel programming on the CPU and on the GPU.

If I could continue this project, there are three more algorithms or processes I would like to explore. I would like to implement Norishige Chiba's Velocity Fields for Mountain Scenery[4]as well as recursive Wang tiles for blue noise.[5]

[4]An erosion model based on velocity fields for the visual simulation of mountain scenery. Norishige Chiba, Kazunobu Muraoka, and Kunihiko Fujita. Journal of Visualization and Computer Animation 9(4):185-194 (1998)

[5]Kopf, Johannes, Daniel Cohen-Or, Oliver Deussen, and Dani Lischinski. "Recursive Wang tiles for real-time blue noise." ACM Transactions on Graphics 25.3 (2006): 509. Print.