Procedurally Generating Terrain

Travis Archer
Morningside College
Sioux City, Iowa 51106
tra001@morningside.edu

Abstract

While noise is generally a nuisance in everyday life, noise can be especially useful for certain people. Procedural Content Generation (PCG) almost always uses some form of noise, and games especially are benefiting from noise. Games use noise to add realism to hand-crafted models, but realistic landscapes generated procedurally are finding greater use. This paper explains methods of procedural generation techniques for creating random landscapes, including noise algorithms, erosion algorithms, water modeling, and vegetation simulation techniques. These algorithms include the Diamond-Square Algorithm, Midpoint Displacement, Value Noise, Perlin Noise, Simplex Noise, Cell Noise (Whorley Noise), thermal erosion, hydraulic erosion, and various vegetation modeling techniques. This paper aims to consolidate information on all these algorithms, as well as provide information on which algorithms are best for which situations. These algorithms range from very easy to implement, to extremely difficult algorithms like Simplex Noise. All algorithms are compared and contrasted with each other based on speed, resource requirements, and visual quality. Levels of sharpness, isotropicness, detail, balance, scalability, randomness, and other factors determine visual quality. Implementation challenges for each algorithm are also discussed, as well as the accuracy of each algorithm with respect to physical models. Specific applications of each algorithm are covered, as well as limitations of each. Simplex Noise is found to have the best balance of quality and speed of the noise algorithms, while hydraulic erosion is found to give the best quality of all tested erosion algorithms. Optimal and interesting conditions are discussed for select algorithms, especially Cell Noise. Uses for noise in higher dimensions is also discussed, as well as methods for creating it. Abstract uses for noise outside of textures are also explored, such as model movement, level layout, path planning, increasing robustness of learning algorithms, and even primitive artificial intelligence models. All code was created in C++, and is released open source for public use. Directions for finding and using this code are included.

Travis Archer
Morningside College
Sioux City, Iowa 51106
tra001@morningside.edu

1 Uses for Noise

In the everyday world, noise is a naturally occurring nuisance that is generally covered up as much as possible. However, in the field of Computer Science, especially 3D modeling, noise has become increasingly useful. Purely digital creations like 3D models are usually perfectly smooth and crisp. That's fine for some things, but doesn't look very realistic. Objects in the real world rarely look perfectly smooth because they possess texture. Irregular bumps and nicks make 3D models look much more realistic, but are difficult and time consuming to make by hand. Noise algorithms create pseudo-random textures quickly with little or no interaction required from the user.

While texturing is easily the most popular use for noise, it has many other uses as well. Noise is commonly added to data sets to make classification algorithms more robust, for quickly creating graphs, and has many other uses. One field I am particularly interested in is procedurally generated landscapes. These are realistic images of artificial landscapes generated pseudo-randomly using various layered techniques. These landscapes are often used for games, but are often pursued for strictly artistic purposes. There are many different techniques for generating these landscapes, and most are created by using a few of these algorithms together. Generation of these landscapes often doesn't stop with noise; often erosion, vegetation, and water models are applied to increase realism. All of these culminate to form images like Figure 1.

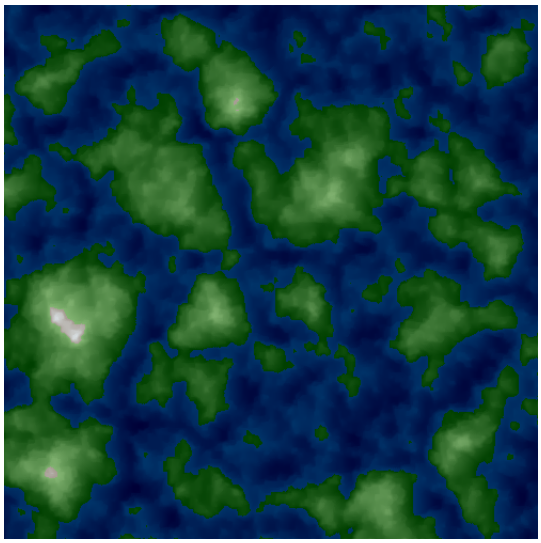

Figure 1: Single octave Cell Noise, 12 octaves of Simplex Noise, 3 iterations of Inverse Thermal Erosion, 300 iterations of Hydraulic Erosion, and Simple Color Mapping. Created in 14 seconds at 513 x 513 pixels.

In my research I implemented many different algorithms, and compared them based on a few standards: speed (processing time), memory usage, and quality. The speed grades for the noise algorithms indicate how many images (at 513 x 513 pixels) can be created in one second on the test hardware. Quality is mostly subjective and depends on the application, but is largely tied to the prevalence of visual artifacts. I also grade the algorithms based

on ease of implementation, however I do not include it in the total score. Each algorithm is compared using these variables, and optimal uses for each are discussed. Each program was implemented in native C++, and images were output to bitmap files.

2 Generating Noise

The basis of almost every procedurally generated landscape is a noise algorithm. Without one, every mountain, valley, rock, and pebble must be crafted by hand. This is certainly possible, but not very feasible for larger maps. Noise creates all of this much faster than a human can dream of, and with greater detail. There are many different algorithms for generating noise, and I will cover some of the more popular algorithms.

One very important term to clarify is dimension. For the purposes of this paper, when I discuss an n-dimensional figure, I refer to a figure that could be stored in an n-dimensional array. So a 1D figure would be a line on a graph, line a sine-wave. A 2D figure is a heightmap, where each point on a 2-dimensional surface has a height value associated with it. This may seem counter-intuitive, but a heightmap is not truly three-dimensional (it is referred to as 2.5D) and so I round down. This seems to be the de facto standard, and I have stuck to it.

2.1 Midpoint Displacement

Perhaps the easiest of all noise algorithms to grasp is Midpoint Displacement. Midpoint displacement works quite simply. First, begin with a straight line between two points. At the midpoint of this line, the value becomes the average of the two outer points plus a random value (called an error). Continue this recursively for each line segment created, until the desired level of detail is reached.[4] It really is this simple, with one minor addition: the range of the random error value should be determined by the length of the line segment. Otherwise, every midpoint could jump erratically. By shrinking the range, beginning features are large but flat, while later features are small, but detailed. Combined together, an image is formed with large defining features, and small detailed features.

This is easy enough to visualize as a one dimensional line, however it scales very easily to higher dimensions. When creating a 2D landscape, subdivide into squares, and each of the four midpoints on the edges of a square are computed as before. However, the single midpoint at the center of the square itself must be calculated as well, and is found by averaging the four corners and adding an error value.
If the height and width of the output image can be expressed as $2^n + 1$, such as 129, 257, 513, or 1025 then the subdividing will eventually calculate every pixel exactly once. The program is usually stopped there so that interpolation between pixels isn't needed. This greatly increases speed while maintaining almost the same level of quality. However, this puts a constraint on the program, as a 200 x 200 image cannot be created without some work-arounds.

Speed: 64 Speed is the major draw of Midpoint Displacement. Midpoint Displacement the fastest of the algorithms tested in this paper. On my test machine a 513 x 513 image takes 0.0156 seconds to create.
Memory: 2 While Midpoint displacement may have incredible speed, it uses the most memory of any algorithm. Because every point must be stored in memory (for averaging with its neighbors), memory becomes a constraint. Most operating systems will not allocate a 2049 x 2049 array of floating point numbers, so linked-lists or other fancy-footwork must be used for very large images. On the other hand, nothing else has to be stored outside of the image surface. Later algorithms like Perlin Noise require a permutation table to be kept in memory, which never makes it into the image itself.
Quality: 4 Midpoint Displacement produces acceptable images. However, major visual artifacts render the image anisotropic. Isotropic means the image has no directional features, so the angle of a rotated image would be impossible to discern, while anisotropic is the opposite. Midpoint displacement generally has visible linear artifacts, and if the image were rotated the angle would be very easy to find.
Total: 4 Blazingly fast, but quality is acceptable at best, and memory is a constraint. While memory is cheap and plentiful, there are better algorithms for the job.
Ease of Implementation: 8 This is one of the easiest algorithm to understand. If recursion is used, the algorithm is even easier, although the call stack of most operating systems wouldn't allow an image much larger than 129 x 129. The iterative design is still quite tame, but may be ugly to look at.

2.2 Diamond-Square

The Diamond-Square algorithm is an improvement on the Midpoint Displacement algorithm. The directional artifacts of Midpoint Displacement come from using two points to calculate some points while using 4 points to calculate others.

Diamond-Square fixes this by using 4 points to calculate all points. The midpoint of each square is determined using the four points of the surrounding square, as in normal Midpoint Displacement. To determine the other midpoints, the four points of the surrounding diamond are used, hence the name. Directional artifacts don't appear because the square used for reference is rotated (into a diamond) on alternating pixels. This breaks the linear artifacts found in normal Midpoint Displacement.[5]

One problem that quickly becomes apparent while implementing this are edge cases. On edge pixels, during the diamond step, only 3 of the four points exist. There are a few ways to mitigate this. First, ignore the missing value and average only the three points. This is generally used, and gives good results. Another option is to just add in a constant value, usually the value used to set up the four original corners of the image. Unfortunately, this often leaves artifacts called 'pinches', which look like creases on the edge of the image.

One interesting option is to wrap the points, so that the fourth corner is found by pretending the pixel in question is on the opposite edge of the image. This increases the randomness, but also makes the image itself wrap. Should the image be replicated next to itself, the

edges would line up seamlessly. While this is undesirable in most situations, it is required in some applications, and is difficult to reproduce by hand.

Speed: 57 Diamond-Square is nearly as fast as Midpoint Displacement. On my test machine a 513 x 513 image takes 0.173 seconds to render. While Midpoint displacement is slightly faster, both create a large image in less than a second.
Memory: 2 The memory requirements and limitations of the Diamond-Square algorithm are identical to those of Midpoint Displacement. The entire image must be stored in memory because pixels must access neighboring values.
Quality: 8 Diamond-Square produces much better quality images than Midpoint Displacement. The images are isotropic, and can be free of artifacts. Pinching can occur, but this can be prevented. This is also the only algorithm that can create wrapping images without a lot of extra effort.
Total: 5 Almost as fast as Midpoint Displacement, but with much better quality.
Ease of Implementation: 6 This algorithm is fairly easy if Midpoint Displacement is already understood. Otherwise, this is a difficult algorithm to jump into.

2.3 Value Noise

Value Noise alone is almost trivial to understand and implement. However, unlike Midpoint Displacement, Value Noise is not natively fractal. A fractal image has many components, but the most important is self-similarity, where small features are similar to large features. If the zoom level of an image cannot be determined, then the image is fractal. Value Noise, Perlin Noise, and the other algorithms to follow, do not display this trait.

To remedy this, a function called Fractional Brownian Motion is used. The idea of fractional Brownian Motion (fBm) is to sum together many octaves of noise, each with increased frequency and decreased amplitude. This translates to layers of noise, each with increasing detail.[1] In an image with sixteen octaves, this is at least sixteen times the work to create an image. All of the remaining algorithms utilize fBm, and for each test I continued iteration until the added detail was unnoticeable.

Value Noise itself is incredibly easy to implement. Set an arbitrary grid across the surface. For each grid-point, assign a random value. For every pixel in-between these grid-points, the value is an interpolation between the closest grid-points. In one dimension, this means an interpolation between the two closest grid-points. In 2D, between the four closest grid-points. To find the closest grid-points, simply round the location variables up and down.[2] While this is simple, the interpolation function must be chosen carefully. The three usual functions are linear, cosine, and cubic. Linear interpolation, also called LERP, is used widely outside of noise generation. It gives fast results, but creates jagged lines. Cosine interpolation is slightly slower, but gives rounded results. Cubic interpolation is extremely slow, but gives perfect results. Which to use is largely dependent on the application. For real-time creation of noise-maps, linear is necessary. If time is plentiful, cubic interpolation should be used.

One interesting enhancement of Value Noise comes from the random number function. If the function can take coordinate inputs and consistently give the same result, while still maintaining randomness between coordinates, the grid doesn't need to be stored in memory. Each pixel can be rendered independently from every other pixel. Random number generators like this are abundant, and by using one the need for memory nearly vanishes. A bitmap of values can be created with each pixel being calculated then written immediately, without the entire map being stored in memory first.

Speed: 3.7 Value Noise alone is very fast, however when compounded with fBm it becomes quite slow.
Memory: 10 Value Noise can be implemented to determine each pixel on the fly, without needing to store anything in memory. Perlin Noise, as seen later, can also do this, but must store a small permutation table and gradient table as well.
Quality: 6 - 8 Depending on the interpolation function, quality ranges from grainy to good. Speed and quality are inversely proportional, so greater quality means slower speed.
Total: A good quality algorithm, but slower than Midpoint displacement. However, it doesn't need to use any memory, which means the image size is not constrained by the amount of memory on the system.
Ease of Implementation: 10 This algorithm is very easy to use, yet is quite customizable. Value Noise alone is incredibly easy to understand, and fractional Brownian motion is also easy to implement.

2.4 Perlin Noise

Perlin Noise is the standard in the noise industry, and with good reason. It is very fast, needs little memory, and has good quality. However, it is not nearly as straight-forward as the previous algorithms.

Perlin Noise is similar to Value Noise in many ways. Perlin Noise, like Value Noise, uses an arbitrary grid. Each pixel between those points are interpolated based on the value of grid-points and distance to them. The similarities end there, as Perlin Noise uses a gradient vector at each grid-point instead of a height value. That means that every grid-point is at 0.0, but has a force coming out of it to influence the pixels in-between.[2]

To set up Perlin Noise, two tables are created: the gradient table and the permutation table. The gradient table is usually very small, 8 entries at most for 2D noise. Each entry is a vector pointing in a random direction. Normally, the gradient table contains equally spaced vectors, since a vector is chosen randomly from the table for each grid-point. The permutation table is usually much larger, often 256 entries in a 2D image. This is used in place of a random number generator for the program. The table of size n is filled with the numbers 0 through $n - 1$. Then the table is randomly shuffled.

For each pixel the following steps are taken:

1. Find the surrounding grid-points, and the gradients associated with them

2. For each gradient, get the dot-product between it and the distance vector to that point

3. Linearly interpolate between the dot-products.

Step 1 is done in largely the same way as Value Noise: round the location variables up and down to determine the surrounding grid-points. Then use the coordinate values to find a random number in the permutation table with the formula $grad = gradient[permutation[x+ permutation[y+permutation[z...]]]]$. For example, if the coordinates of the grid-point are 3,5, then the random number would be determined by $grad = gradient[permutation[3 + permutation[5]]]$ Because it is all done with array-indexing, the operation is incredibly fast, unlike other random number generators that require divisions, multiplications, and prime numbers.

Step 2 is slightly more difficult. The gradient vectors are made of a series of directional components. Another vector is created using the distance components between the pixel in question and the grid-point. The dot-product of the two is taken, and this process is repeated with each of the surrounding grid-points.

Finally, linear interpolation is used between the values found in Step 2. A fade function is used to amplify the contributions from the corners.

This is all done for each pixel, and for each octave if fBm is used. While Perlin Noise seems much more complex than Value Noise, it is optimized to reduce division calls and use addition and multiplication calls instead.

Speed: 1.5 Slower than the fastest Value Noise, but much faster than Value Noise using Cubic interpolation.
Memory: 8 Just like Value Noise, the pixel data doesn't need to be stored in memory. However, Perlin Noise uses a small amount of memory for the permutation and gradient tables. The gradient table can be replaced by some careful bit-fiddling, reducing the memory load even further.
Quality: 9 Great quality, with good control like Value Noise.
Total: 6 Great algorithm, but it is being replaced in the industry by Simplex Noise.
Ease of Implementation: 4 Not nearly as straight-forward as the previous algorithms. The idea behind it is easy enough to grasp though.

2.5 Simplex Noise

Simplex Noise was created to overcome the weaknesses of Perlin Noise. Perlin Noise is fine in one, two, or three dimensions, but gets extremely slow in higher dimensions. In fact, Perlin Noise is $O(n * 2^n)$, with n being the dimension. While this is exponential time, Simplex noise is $O(n^2)$, making it polynomial time.[3]

The main weakness of Perlin Noise is that it uses a hypercube as the grid. In 2D this is a square, in 3D a cube, and so on. In a given dimension this equates to 2^n corners for

each pixel to use. The problem is compounded by the fact that linear interpolation only works in 1 dimension. To linearly interpolate four points, as in 2D Perlin Noise, two parallel sides must be separately interpolated, then those two results must be interpolated as well. This equates to $2^n - 1$ linear interpolations, which in 2D is 3, in 3D is 7, in 4D is 15, and in 15D is 32767 linear interpolations per pixel.

Simplex Noise solves this by uses simplices instead of hypercubes. A simplex is the shape with the fewest corners in a dimension. In 2D it is a triangle, in 3D a pyramid, and in n-dimensions it is a shape with $n + 1$ corners. This is the first advantage of Simplex Noise: far fewer corners to reference for each pixel. While a point in 15D Perlin Noise would reference 32768 grid-points, a pixel in 15D Simplex Noise will only reference 16 points.

The second enhancement of Simplex Noise is that no linear interpolations are required. A radial attenuation function is used to weigh each input, once for each grid-point. While Perlin Noise requires $2^n - 1$ linear interpolations for each pixel, Simplex Noise requires only $n + 1$ radial attenuations.

The basic idea of Simplex Noise in 2D is to replace the grid of squares with a grid of equilateral triangles. This may seem simple enough, but finding the position of the three closest grid-points is no easy matter. With a grid of squares, just round the position of the pixel up and down to find the four surrounding corners. There is no easy way to do this with a grid of simplices. The standard way to do this is to do a non-uniform skew of the simplex grid until it resembles a grid of split squares, using the following code:

```
n = dimensions;
general_skew = (sqrt(n+1) -1)/n;

skewed_x = normal_x + (normal_x + normal_y + ... + normal_n) * general_skew
skewed_y = normal_y + (normal_x + normal_y + ... + normal_n) * general_skew
...
skewed_n = normal_n + (normal_x + normal_y + ... + normal_n) * general_skew
```

This is run on the coordinates of the pixel in question. From there, two of the three corners can be found by rounding both coordinates down and both up. Then the pixel can either be in the upper-left triangle or the lower-right triangle. If the relative x value is greater than the relative y value, then the pixel is in the lower-right triangle, and the corner can be found by rounding the x value up and the y value down. If the y value is greater than the x value, the corner can be found by rounding the x value down and the y value up. These integer values are used to associate the three corners with three independent gradients, using the same permutation table and gradient table from Perlin Noise.

Once the coordinates for the corners in skewed space are found, they must be skewed back to normal simplex space to find the distance to them. To do this, the following code is run on each of the three corner coordinates:

```
n = dimensions;
```

```
general_unskew = (n+1-sqrt(n+1))/(n*(n+1));

normal_x = skewed_x + (normal_x + normal_y + ... + normal_n) * general_unske
normal_y = skewed_y + (normal_x + normal_y + ... + normal_n) * general_unske
...
normal_n = skewed_n + (normal_x + normal_y + ... + normal_n) * general_unske
```

Once the coordinates of the closest grid-points in normal space are known, the distance vectors can be easily computed. Just as in Perlin Noise, the dot product of the distance vector and gradient vector is computed for each corner. Then the radial attenuation function is performed on each corner:

```
radius = min( 0.0, 0.5 - pow(x_distance,2) - pow(y_distance,2) ... - pow(n_

radial_attenuation_result = pow(radius,4) * dot_product_result;
```

The attenuation function limits the contribution from each corner based on distance, while cutting the contribution of distant corners to zero. Finally, the contributions from each corner are summed for the final result.

Speed: 3.2 Better than Perlin Noise, especially in higher dimensions.
Memory: 8 Just like Perlin Noise, no pixels need to be stored in memory, but a small permutation table and gradient table are needed.
Quality: 10 Best quality of all the algorithms.
Total: In all, the best quality and acceptable speed.
Ease of Implementation: 2 Very difficult to understand, much less implement. Mistakes are easy to make, but very difficult to track down.

2.6 Cell / Whorley Noise

Cell Noise, also called Whorley Noise, alone is not very useful for terrain generation. It creates very artificial structures, usually with harsh jagged lines or bubbly surfaces. However, combined with the natural appearance of another noise function it can create distinctive landscapes.

The basic idea of Cell Noise is to scatter random points onto a surface, then the value of any given pixel is the distance to the n^{th}-closest point. The choice of 'n' radically alters the resulting image.[6] Using an 'n' of 1 is referred to as F1, while an 'n' of 2 is referred to as F2. Combining these is common, such as F2-F1, as is altering the distance function. The most common is Euclidean, the standard $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$, although another common distance function is Manhattan or City Block distance, which is just $|x_1 - x_2| + |y_1 - y_2|$. Less common functions are Chebychev and Quadratic, which still create interesting images.

Optimizations can be done, such as setting a grid on the surface, and only checking the distance to points in the adjacent squares. Memory usage ranges based on implementation,

but generally the coordinates of the points are stored somehow.

Speed: 1 Speed varies widely depending on the distance function and whether or not fBm is used. In all, it is generally slower than Value Noise, making it the slowest in this list.
Memory: 4 Varies depending on the implementation. Usually uses more memory than Perlin Noise but less than Midpoint Displacement.
Quality: 4 Different choices of F (F1, F2, etc.) and distance functions result in wildly different images, but none are very convincing alone.
Total: 3 A very interesting noise function, but not fit to create landscapes on its own.

2.7 Conclusions

In all, Simplex Noise gives the greatest quality while still giving good speed. If memory isn't an issue, Diamond-Square is extremely fast, but is constrained by the size limitations, as it must be $2^n + 1$ pixels wide and tall. While Simplex Noise is recommended for most tasks, it is also very difficult to understand and debug.

3 Erosion Algorithms

Noise alone can create interesting landscapes, but erosion can help add an extra layer of realism. Below are three of the most effective erosion algorithms. In the end I found that Hydraulic Erosion, when used correctly, gives the best quality, but is very slow.

3.1 Thermal Erosion

Thermal erosion models gravity eroding cliffs that are too steep. If the angle is too sharp, soil will fall to a lower area. Thermal erosion is fairly simple to model. First, define the difference T, which is the maximum difference allowed before gravity takes over. For all pixels:

1. Get the difference in height between this pixel and the neighboring pixel
2. If the difference is greater than T, remove some amount of soil from the taller pixel and deposit it into the lower pixel.

Very simple, yet it works quickly. This is done repeatedly until the desired amount of erosion is reached.[5] The speed of this algorithm can be improved further by changing the neighborhood type. The three standard type are the Moore neighborhood, the Von Neumann neighborhood, and the rotated Von Neumann neighborhood. While the Moore neighborhood provides the best results, it is also the slowest. The rotated Von Neumann neighborhood gives good results while increasing speed.

Speed: 10 Very fast, and it requires very few iterations to get results.
Memory: 10 Doesn't require any extra memory, outside of the surface to work on.
Quality: 3 Not very convincing, but may be acceptable for some applications.

Total: 7 While it is very fast, the quality is unacceptable in most situations.
Ease of Implementation: 8 Easy to implement, but the parameters need to be honed to get usable results.

3.2 Hydraulic Erosion

Hydraulic Erosion models rainwater picking up soil, washing down into basins, then evaporating and depositing soil.[5] Hydraulic erosion provides good quality results, but is very slow. Hydraulic Erosion also requires large amounts of memory. While the other erosion algorithms work directly with the source image, Hydraulic Erosion requires a water table and a sediment table, meaning it may use up to 3 times the memory of Thermal Erosion.

To begin the algorithm, a water table is set up, which is an array the size of the image that holds the amount of water in each pixel. A sediment table is also used to track the amount of sediment within the water. A few variables must also be defined:

- $rain\_amount$: how much rain falls per iteration
- $solubility$: how much soil is eroded by one unit of water
- $evaporation$: what percentage of the water evaporates each iteration
- $capacity$: how much sediment one unit of water can hold

By adjusting these variables the effects of the erosion will change. High evaporation moves soil short distances, while low evaporation makes water pool.

Once these are defined, the process can be broken into four parts that are performed on each pixel:

1. Rainfall: add $rain_amount$ units of water to each pixel of the rain table.
2. Erosion: move $solubility * water[pixel]$ soil from the base image into the sediment table.
3. Movement: move water downhill if possible
4. Evaporation/Deposition: remove $evaporation * water[pixel]$ water from the pixel. If the sediment in $sediment[pixel]$ exceeds $capacity * water[pixel]$, move soil from the sediment map into the base image until it no longer exceeds the capacity.

This is performed for every pixel, for every iteration. Hydraulic erosion tends to work slowly, but produces good results. The image does not look washed out, but has obviously been eroded.

Speed: 3 Very slow per iteration, and it usually takes many iterations to get usable results.
Memory: 2 Uses a lot of memory, up to thee times the memory needed by Thermal Erosion.

Quality: 10 Very appealing results. Looks much more realistic than Thermal Erosion.
Total: 5 Great quality, but the speed and memory requirements are very high.
Ease of Implementation: 4 Difficult to grasp, and errors are elusive.

3.3 Inverse Thermal Erosion

Inverse Thermal Erosion was described by J. Olsen as an improvement on Thermal Erosion.[5] For gaming, plateaus and cliffs are much more desirable than rolling hills. Thermal Erosion destroys cliffs and rarely creates plateaus. By flipping the erosion condition the opposite effect can be created: hills are eroded into flat plateaus, and cliffs are sharpened.

Thermal Erosion erodes a pixel if the height difference between the pixel and its neighbor is greater than T. Inverse Thermal Erosion erodes a pixel only if the height difference between the two pixels is *less than or equal* to T.

This tiny change results in terrain that is much better for gaming. However, from a straight overhead view it looks washed-out. Visual artifacts become apparent if too many iterations are run, manifesting as thin wandering lines. These are formed because the algorithm preserves cliffs and erodes hills. If a feature has a cliff on one side and hills on the other, the hill will erode until only the cliff-face is left. Careful control of the iterations will mitigate this, but the algorithm is far less forgiving than Hydraulic Erosion.

Speed: 10 Just as fast as Thermal Erosion. **Memory: 10** Uses little memory outside of the base image. **Quality: 6** The quality is perfect for many games, but is not as realistic as Hydraulic Erosion. **Total: 8** Great speed and memory usage, but the quality isn't always acceptable. **Ease of Implementation: 8** This is the Thermal Erosion algorithm with one $>$ symbol changed to a $<=$ symbol.

3.4 Conclusions

Hydraulic Erosion certainly provides the best results, but takes the most time. However, on my test machine 500 iterations of Hydraulic Erosion on a 500 x 500 image takes less than 5 seconds. Inverse Thermal Erosion can provide somewhat similar results after only 2 iterations, but the quality is still below Hydraulic Erosion. If time and space are available, I recommend a combination of Inverse Thermal Erosion and Hydraulic Erosion, to create plateaus and cliffs, then soften them. To reintroduce detail into the terrain while maintaining the larger eroded features, a high frequency layer of noise can be added after the erosion functions are finished.

4 Vegetation Modeling

While noise and erosion may create realistic features, without color the terrain is completely alien to the eye. Without color noise just creates clouds. Vegetation modeling is

important when creating a procedurally generated terrain.

4.1 Simple Color Mapping

The simplest method for adding color to a height-map is to relate portions of height to colors or gradients of colors. A simple rule like "The bottom 50% of the image will go from Dark Blue to Light Blue" creates the illusion of oceans. This can be hard-coded, or a separate color-map image can be used.
This is the technique used in each of the above figures, and is acceptable for testing algorithms. The quality is surprisingly good, while the time taken on coloring a pixel with this method is nearly negligible, and is done with one pass over the base image.

4.2 Terrain Mandated

More sophisticated rules can be used with great results. Generally water is defined with simple rules as above, but vegetation takes slope and height into account. A pixel gains vegetation if the difference in height between it and the tallest/shortest neighbor is less than T. At elevations above H vegetation dies off and is likely replaced by white snow pixels. Depending on the scope of the terrain map, areas in the extreme North and South of an image could be similarly replaced by snow pixels. What these rules have in common is that they are performed statically, in one pass. In this way, they do not model complex ecosystems, but simple functions that are largely independent of pixels not immediately neighboring the pixel in question.

4.3 Dynamic Modeling

At the extreme, dynamic systems such as rainfall and water-flow could control where vegetation grows. This would be done iteratively, with vegetation springing up next to sources of water, and dying in areas where water is scarce. It could then spread using simple rules, likely in combination with the terrain-mandated rules above, but would be done over many iterations. Looping would stop when some threshold percentage of the image is neither water nor vegetation. While this may create more ecologically realistic models than the static methods above, the visual results often do not outweigh the loss of speed incurred by the complexity of dynamic systems.

5 Overall Conclusions

Procedural terrain generation is a useful and exciting area, yet interest is scarce. Information on all these systems is certainly available, but is scattered. Noise algorithms are used in many areas and are especially fragmented. This paper is meant as in introduction to the core concepts of procedural terrain generation. Exploration is key to creating interesting landscapes. The source code for all of the noise and erosion algorithms, as well as more detailed explanations of each, are available at Google Code, http://code.google.com/, under the title 'fractalterraingeneration'. The source code is all written in C++, and is freely

available. As memory and processing power become cheaper and plentiful, Simplex Noise, Hydraulic Erosion, and dynamic vegetation modeling will hopefully become more usable and widespread.

References

[1] Dasgupta, A. *Fractional Brownian motion: its properties and applications to stochastic integration*, University of North Carolina at Chapel Hill. 1997.

[2] Ebert, D., F.K. Musgrave, D. Peachey, K. Perlin and S. Worley. *Texturing and Modeling: A Procedural Approach*, Academic Press.1994.

[3] Gustavson, S. (2005). *Simplex Noise Demystified*. Technical Report, Linkping University, Sweden.

[4] Norros, I., Mannersalo, P., and J.L. Wang, *Simulation of fractional Brownian motion with condi- tionalized random midpoint displacement*, Advances in Performance Analysis, 2 (1999), pp. 77101.

[5] Olsen, J. (2004). *Realtime Procedural Terrain Generation*. Technical Report, University of Southern Denmark.

[6] Steven Worley. (1996). *A cellular texture basis function. In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques (SIGGRAPH '96)*. ACM, New York, NY, USA, 291-294.