Dynamic Terrain Generation and Simplification

Master of Science in Computer Science Project Paper

Chris Johnston, fsctj2@uaf.edu
Department of Computer Science
University of Alaska Fairbanks
2006/04/24

Introduction

The interactive generation and rendering of procedurally synthesized terrain is an interesting, useful, and still largely unsolved problem. In this paper, we present a terrain rendering program that incrementally generates the terrain at interactive rates.

Limitations of Prior Work

In the scope of computer graphics there are many examples of terrain centric software available. Each implementation suffers from one or more of the following undesirable characteristics. Undesirable characteristics are those that we find to be shortcomings in the implementations used in the software.

Large disk space consumption.
Long render times.
Long generation times.
Not Traversable.
Short view distance.
Popping. (When terrain detail suddenly appears in front of the camera.)

Our Proposed Solutions:

- Large disk space consumption:
  - Dynamically generate the terrain.
- Long render times:
  - Distance based terrain simplification.
- Long generation times:
  - Generate terrain data as needed.
- Not Traversable:
  - Generate terrain data based on position and update based on movement.
- Short View Distance:
  - This links to long render times. The terrain simplification method chosen must remove this characteristic as well. Our goal is 200 miles or 320 kilometers.
- Popping:
  - Level of detail transition areas need to be kept as far away from the camera as possible and differences in detail are minimized while still maintaining performance.

The overall goal of this project is to show that it is possible to procedurally render terrain with a 200 mile view distance and still avoid the above undesirable characteristics or show that it is not possible and explain why. Answering the question,
"Is it possible to create a program such that no disk loads occur from a real-time renderable, traversable world larger than Earth with a view distance of 200 miles?"

General Information

Axis Information:

-x = West
x = East
-y = Down
y = Up
-z = North
z = South

Height Generation

In order to generate heights for each point composing the terrain we needed a random number generator. The generator must accept as input an x and z terrain position and output a number to be used as y, the height. In addition to the above, the output of the generator must be void of visually repeating patterns as seen in Image 1 below.

Image 1: Output from a random number generator mapped to the values 0 to 255.

```
float rand(int x, int z)
{
        int n;
        n = x + z * 57;
        n = (n<<13)^n;
        float temp = ((float)(1.0f - ( (n * (n * n * 15731 + 789221) + 1376312589) & 0x7fffffff ))) / 1073741824.0f;
        return temp + 1.0f;
}
```

Code 1: A basic random number generator with 2D input.

Problems with the Initial Implementation

Random Number Generation

Up until my first presentation we explored many different possible random number generators. One of the requirements for the software was that if the camera is at a position x, z on one machine and the camera is at the same x, z position on a completely different machine, both machine users must see the same terrain. In order to generate the same terrain heights on different platforms we have to re-seed the generator with the x, z point in the terrain mesh for each point in the terrain in order to get that terrain point's y value. An example of one attempt can be seen in Code 1 above. We were unable to find a generator that would not produce repeating patterns. At my first presentation Dr. Hartman pointed out that we did not have a random number problem, we in fact had a hashing problem. It is already known that random number generators make terrible hashing functions, thus our undesirable output. We then decided to use the SHA-1 hashing algorithm for reasons that will be described later in this paper.

Interpolation Function and Zones

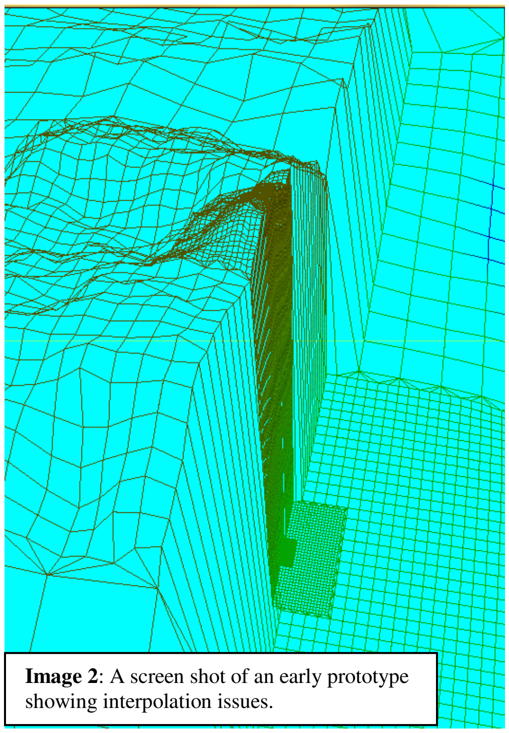


Image 2: A screen shot of an early prototype showing interpolation issues.

The initial implementation was to use points and zones. The world would consist of zones and each zone would consist of zone points thus giving each terrain point 5 degrees, x, y, z, zone_x, and zone_z. The idea was to eliminate the issue of running out of floating point precision by using the zones to limit the size of x and z values. The issue that came about from this implementation was the interpolation of the values used to generate the heights across zone boundaries. The boundaries of zones were not continuous; mountains would suddenly drop into an abyss in a single meter as seen in Image 2. The problem is that zone(0,0) point(ZONESIZE,0) and zone(1,0) point(0,0) must get exactly the same height value.

We could accomplish this by combining the zone and point into a single "double" variable. Unfortunately this would eliminate the purpose of the usage of zones since in combining the two we are now constrained by the resolution of doubles. Since the proposed solution to the interpolation problem limits us to the resolution of a double we decided that the simplest solution was to use doubles for the x and z positions and eliminate the use of zones all together. IEEE doubles have a 52-bit mantissa. That means they can exactly represent integers at least up to 2^{52} which is greater than 10^{15}. Ten to the 15th is a million billion. So with each unit as meters, that's a trillion kilometers. The Earth is 40,000 km in circumference. So doubles should be enough for a world millions of times larger than the Earth.

Function Application

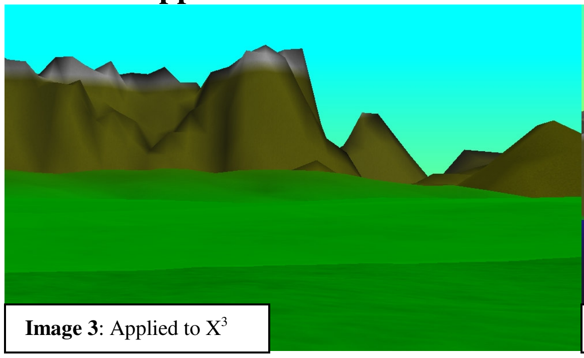


Image 3: Applied to X^3

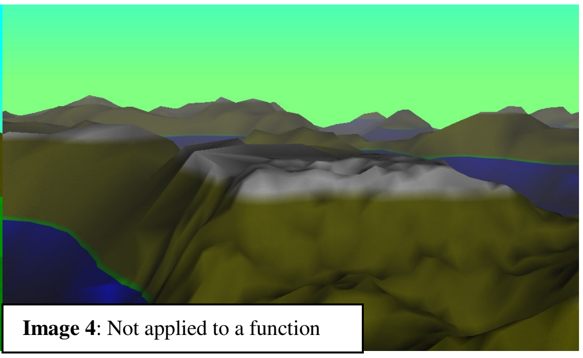


Image 4: Not applied to a function

While at first glance both of the above images look “good” or “pretty” it is important to notice than in Image 4 there are mountains and lakes, but no visible plains. It turned out that in order to generate virtual terrain that would resemble terrain found in reality we would have to apply a nonlinear function to the output to generate heights. Two questions spawned from this. The first was obvious, “What function should we map to?” The second was not so obvious and involved the terrain height generation function, to be described later in detail. To generalize, the terrain height generator uses a sum-of-noise algorithm. The question was, “Do we apply the function at each layer of the noise before they are summed or do we apply the function to the sum of the layers?” We tested both possibilities and discovered that applying the function at each layer before the summation was not the proper method to get the results we wanted. The output of the prototype was so similar to Image 4 that a screen shot of the run has not been included. The visual result of applying the function before the sum was an effect at high resolutions. Meaning that when we viewed the terrain close up it was visible that the function application of the function was affecting the output, but the purpose was to change the terrain at a low resolution level. The sum of the noise layers is what is used to generate the final heights, so we tried applying the function to the summed random number values and the result can be seen in Image 3. This resulted in the visual effect we wanted, producing a world of mountains, plains, and bodies of water.

Terrain Representation

Using the nested grid structure placed many constraints on the way numbers were used and what numbers were used to represent the terrain. There were many tries with many different combinations of parameters before we reached our final representation scheme. In a later section Terrain Representation our two major attempts are noted. The first scheme was used to generate the demo for my second presentation and the second scheme is what the demo currently uses and the reasoning behind the change is noted there.

Final Implementation of Noise

Secure Hash Algorithm - 1 (SHA-1)

A quality found in good hash algorithms is that hash collisions are minimized. This means that each input will generate a unique output. We are taking the output of the hash algorithm and using it to generate numbers between -1.0 and 1.0, this means that a value appearing more than once is inevitable. While collisions are inevitable SHA-1 provides us with the assurance that pattern repetition will be avoided. This property is described by the following statement, "There are more than one 6 kilometer mountains, but only one Mt. McKinley." Dr. Lawlor already had a working implementation of the Secure Hash Algorithm 1 or SHA-1 available. While there may be other algorithms that are just as effective and more efficient we went with SHA-1 because the implementation was readily available. This algorithm accepts up to 16 32-bit words and outputs 5 32-bit words. For the initial prototypes the input words consisted of x and z positions as well as zone_x and zone_z positions, the final implementation only inputs the x and z positions. The first word of the hash key is converted to a value between -1.0 and 1.0 and output as the result. This implementation gives us a 2D pseudo-random number generator which gives output void of visually repeating patterns.

Introduction to Noise

Terrain is made up of various levels of detail. Take for example the outline of a mountain range. It contains large variations in height (the mountains), medium variations (hills), small variations (boulders), tiny variations (stones), and so on. This fractal nature can be captured using Dr. Ken Perlin's ideas about adding up noisy functions at a range of different scales. Dr. Perlin's works were in the area of procedural texture generation, but it is a simple transition to go from color representation to height representation. Image Sequence 1 gives a brief view of how the sum of noise works.

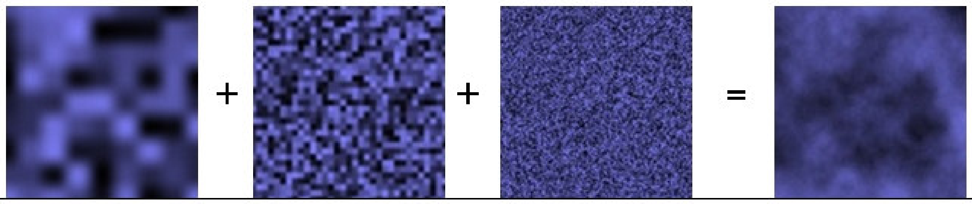
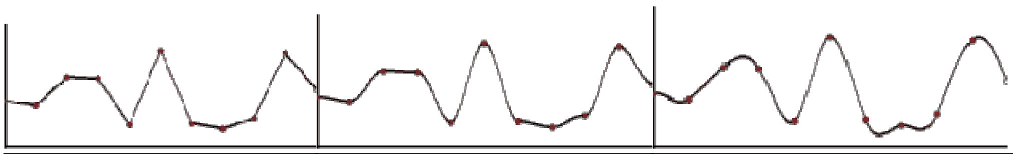


Image Sequence 1: Sum of noise example. Dark spots are low elevation, light spots are high elevation.

Interpolation

Our SHA-1 number generator will provide us with the numbers needed to generate the heights for terrain points, what it does not provide is a way to smoothly progress between these values. In order to produce more natural transitions between terrain heights we need an interpolation function. There are three types of common interpolation functions that come up when such a problem needs to be solved, Linear, Cosine, and Cubic. Linear is the simplest and least costly of the three, but its results are far from desirable when it comes to rendering terrain. As shown in Graph 1 linear interpolation results in jagged peaks and abrupt changes in elevation. One of the requirements set was that "popping" needs to be minimized or entirely removed (if possible), in order for this to happen the interpolation function must generate a value for the point that closely represents the actual value at that point should we generate it. Linear interpolation makes no attempt to “guess” values at points, thus “popping” will not be reduced. Cosine interpolations seen in Graph 2 results in a smoother transition between values when compared to Linear, but abrupt changes are still visible. The Cubic Interpolation shown in Graph 3 results in the smoothest transition from point to point and theoretically should produce the least amount of “popping” when compared to the other two methods. Cubic Interpolation provides the least amount of spatial discontinuities and this property combined with the idea that amplitudes (described in detail in the next section) are decreased as the levels of detail increase the probability that the interpolated value represents the actual value is higher than it is with the other two algorithms.

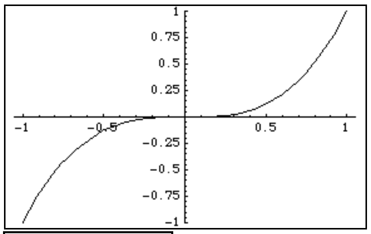


Graph 1: Linear Interpolation	**Graph 2:** Cosine Interpolation	**Graph 3:** Cubic Interpolation

Frequencies and Amplitudes

The frequency determines how often the height generation function is called. Since our smallest resolution is one meter a frequency of 1.0 means that every meter the function is called in order to get the height for that point, a frequency of 0.5 means that the function will be called every two meters and all points between are generated by the interpolation function, and so on. Therefore the smaller the frequency the data becomes more dependent on the interpolation function to fill in the gaps. The larger the frequency the data becomes dependant on the height generation function. It is important to note that the interpolation function is less costly than the height generation function. If this were not the case then interpolation would be a waste and we could simply generate all points at the highest resolution.

In Image Sequence 1 there are three layers that are summed together in order to produce the final result. Each one of these layers is called an octave. In order to understand the role of amplitude we first have to explain the role of octaves in our terrain generation. The first image in Image Sequence 1 has very little detail and would represent height values of terrain that is distant from the camera. We add octaves of increasing detail one by one to the sum of the previous octaves as the distance to the camera decreases. Amplitude comes into play when we go to add the octaves together. The amplitude determines exactly how much an octave is allowed to alter the sum of the previous octaves. For example, let us assume that for each octave we left the amplitude constant at 1.0. If we were to look at a single point and follow it through the summation process we might see that at one point in the summation it had a height of 1.0, but the next octave in the summation gives it at a value of -1.0, so at this point it will be given a height of 0.0. Since we are adding octaves in order to increase resolution as we move closer to structures in the terrain this would result visually in a mountain suddenly dropping to a plain as we got closer to it. By decreasing the amplitude with each addition of a new octave we decrease the new octave’s effect on the previous sum. This means that changes to terrain thousands of meters away can be meters of difference while changes to terrain that are close to the camera will be within centimeters of difference.



Graph 4: X^3

Applying a Function to the Output

As stated in the “Pitfalls” section above, our initial output did not have characteristics resembling true terrain. In order to achieve this we needed to apply a function to the output. We chose X^3 for two reasons, the first is that it preserves the sign of the input and the second is that X^3’s graph (Graph 4) shows that it will produce plains, mountains, and (given a negative input) lakes. This project is in no way attempting to prove that X^3 is the function that describes terrain on Earth, it was simple to implement and fit our needs. The final section of this paper describes improvements and further projects that can spawn from this one and possible function choices that may increase the realism of the output.

Putting it all Together

The pseudo code for the full height generation function follows:

```
Random(x, z)
{
    word1 = x
    word2 = z
    outputWord1 = First word of SHA-1 hash of word1 and word2.
    return outputWord1 as a number between -1.0 and 1.0
}

CubicInterpolation(a, b, e)
{
    factorB = $3e^2-2e^3$
    factorA = 1.0 - factory
    return (a*factorA)+(b*factorB)
}

InterpolatedNoise(x, z)
{
    integerX = IntegerPart(x)
    integerZ = IntegerPart(z)
    fractionX = x - integerX
    fractionZ = z - integerZ
    x0z0 = Random(integerX, integerZ)
    x1z0 = Random(integerX+1, integerZ)
    x1z1 = Random(integerX+1, integerZ+1)
    x0z1 = Random(integerX, integerZ+1)
    v1 = CubicInterpolation(x0z0, x1z0, fractionX)
    v2 = CubicInterpolation(x0z1, x1z1, fractionX)
    return CubicInterpolation(v1, v2, fractionZ)
}
```

(continued on next page)

```
HeightAtXZ(x, z, octaves)
```

```
{
1       total = 0.0
2       frequency = 0.000030517578125
3       amplitude = 1.0
4       for each octave
5               augment = InterpolatedNoise(x*frequency, z*frequency)
6               augment = augment * amplitude
7               if (total + augment > 1.0) total = 1.0
8               else if (total + augment < -1.0) total = -1.0
9               else total = total + augment
10              amplitude = amplitude / 2.0
11              frequency = frequency * 2.0
12      total = $total^3$
13      return total * MaximumTerrainHeight
}
```

Let us begin with Lines 2 and 3 of HeightAtXZ in the pseudo code, the amplitude and the frequency. For generating the terrain heights we used a starting frequency of 0.000030517578125 or 2^{-15} and starting amplitude of 1.0. For each progressive octave the frequency is doubled and the amplitude is halved. Unfortunately when it comes to computer graphics many decisions are made based on the "Because it looks good." argument. These numbers are the result of several different runs and produced the terrain seen in Image 10. We found this output to be acceptable and so these values were set. By no means are these the only values that work, for more information see the final section on improvements and future projects.

The InterpolatedNoise function makes use of the Cubic Interpolation method discussed earlier and shown in Graph 3 and the SHA-1 algorithm. Lines 5 through 8 of the InterpolatedNoise function gather the noise values of surrounding points using the SHA-1 number generator and use their CubicInterpolation values in order to generate the noise value for the requested point.

The final step is to apply X^3 to the output. Since the overall output is between -1.0 and 1.0 we can simply multiply the output by itself twice and still have a number between -1.0 and 1.0 as seen on Line 12 of the function HeightAtXZ. This number is then multiplied by whatever value we wish the highest point in the virtual world as seen on Line 13 of the same function. We chose 12,000 meters. Again, this is an arbitrary number and can be changed to suit different needs. We now have the height for the x z point fed into the function.

Terrain Representation

Nested Grid Structure

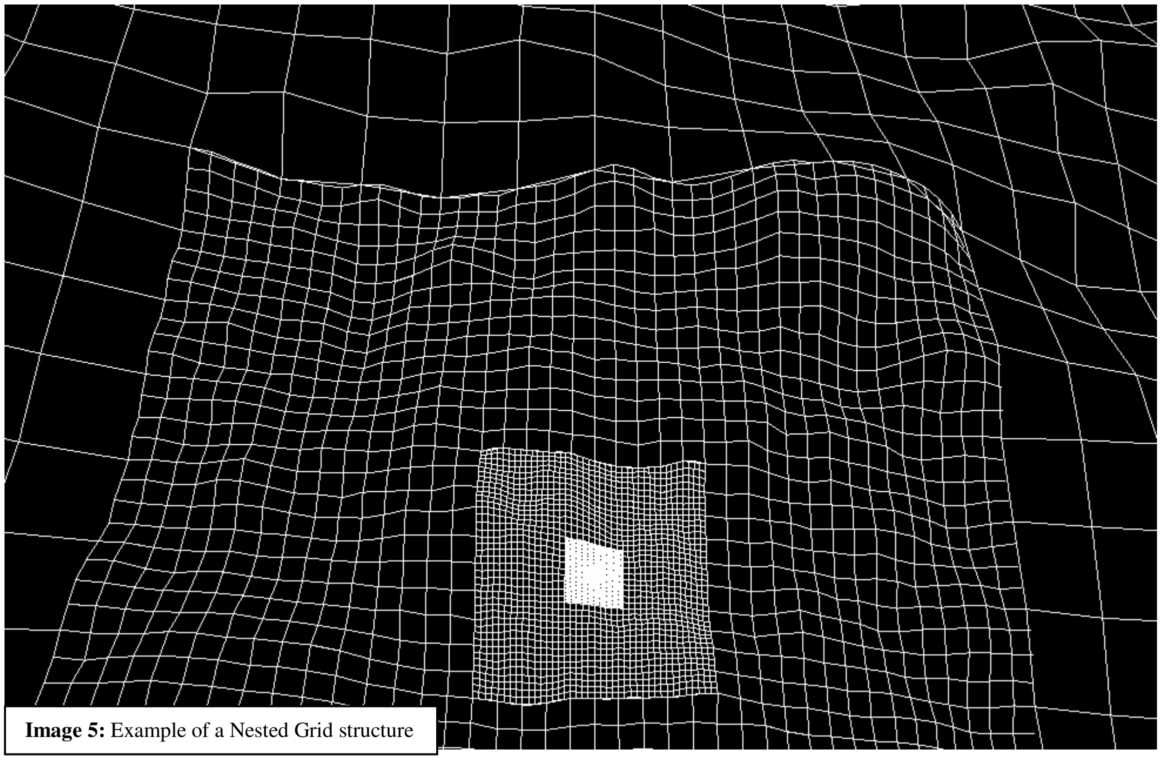


Image 5: Example of a Nested Grid structure

The nested grid structure is both simple to implement and suits the needs of our dynamic height generation. An example can be seen above in Image 5. Our implementation involves the use of overlapping 2D arrays to hold terrain points. One of the requirements is that we must be able to traverse the world that we create in real-time. The nested grid structure allows us to do so with a minimal amount of effort. Generating the terrain heights is the most costly operation of the entire process. As the camera moves through the world the worst case scenario would require only one row per grid layer to be updated, the rest can be shifted. The best case scenario would only require one row in one layer to be updated. This phenomenon is the result of the grid sizes of each layer. Let us assume that the smallest grid is 1.0 square meter and the second layer is 5.0 square meters. If the camera were to move North 2.0 meters then the values of the first layer would be shifted South and a new row would be generated, this action would occur twice. On the other hand the second layer would remain the same. Now, from this new position we move North another 3.0 meters. Now the first layer is shifted South and a new row is generated three times. This time the total movement is 5.0 meters, now the second layer will be shifted South once and one new row will be generated. In addition to facilitating our “generate on demand” design, another benefit of the nested grid structure is that the number of polygons is static. This allows for better performance measurements when evaluating the cost of render time and generation time and how they affect the frame rate.

The most important piece of information that we gathered from the prototype development was that in order to generate the terrain as it is traversed and maintain a reasonable average frame rate (25+) no more than 600 terrain points can be generated per camera move. This puts the first of many constraints on the sizing of each layer. The constraints are: No more than 600 terrain points can be generated per move. Each layer must fit exactly within a symmetric subset of the next largest layer. The largest layer must span 400 square miles or 640 square kilometers. Each layer must be large enough to prevent detail changes from occurring so close to the camera that "popping" becomes an issue.

The worst case scenario requires one row from each layer to be generated, this means that in order to meet the first constraint the sum of the number of terrain points making up a row for each layer must be less than 600. Our original implementation used 8 arrays of size 41 by 41, meaning at the very worst we would have to generate 328 terrain points. This way each array represented a 40 by 40 grid of squares. This decision forces the resolution of the largest layer to be 640000 meters/40 squares or 16000 meters. Now that we know that each grid space of the largest layer is 16000 square meters we can begin calculating the smaller internal layers. We know that we have 7 layers left, and that our goal is that the smallest layer's resolution be 1.0 square meter. This is where trial and error divisions and the second constraint come into play. The second constraint means that whatever we choose to be the next smaller resolution must divide evenly into the resolution of the previous larger layer. After multiple attempts and failures to get the final resolution to fall on 1.0 square meters we decided to pick a constant division of 4. This gave us the following resolutions in square meters: 16000, 4000, 1000, 250, 62.5, 15.625, 3.90625, 0.9765625. To make a long story short this implementation is visible in Image 5, the theory and layouts are sound, where this fails is in the final constraint with respect to "popping". The smallest resolution is roughly 1.0 square meter, with the camera in the center the edge of the layer is visible within 20 meters. This means that as the camera moves forward 4 square meter details are suddenly given 1.0 square meters of detail and this resulted in a negative visual effect. It was also pointed out by Dr. Chappell that the largest resolution of 16000 square meters or 10 miles is far too large at 200 miles away and that smaller details should be visible. These two points lead to the final implementation.

The new scheme now had a new constraint, but on the other hand we now had a benchmark which we can use to build the new scheme. The seam between the nested layers is where the changes in detail occur. The only way to move the seam farther away from the camera is to increase the size of the smallest layer. It had become apparent that the smallest layer needed to be larger than the rest in order to keep the detail changes as far from the camera as possible, the question is, "How large?" The previous implementation had the first seam located 20 meters from the camera and this was deemed too close, so we turned to real life for an answer. A football field is roughly 100 meters long, and if a small change occurred at the end of a football field would it be terribly noticeable? Our decision was "No." and we went with a 201 by 201 array to represent a 200 by 200 grid of 1.0 square meter blocks. Now when the camera is in the center of the layer the seam is 100 meters away. The downside of this decision is that we now only have 400 blocks to work with since we can only update 600 per move. We have also only covered 100 meters of what needs to be a 360000 meters view distance. We now have to determine three things, how many more layers will there be, what will their resolutions be, and what array size will we use to represent them. The smaller the array the more layers we can have, but the resolution of each layer will be very different from the adjacent layers, thus resulting in incredible "popping". The larger the array the less layers we can have, but the resolution between adjacent layers will be close enough to minimize “popping”, unfortunately we are still bound by the performance constraint that the no more than 600 points can be generated per move. After days of number crunching we came up with the following scheme. We used 5 arrays. One was 201 by 201 to represent a 200 by 200 grid and would represent the 1.0 square meter resolution. The remaining 4 were 101 by 101 to represent 100 by 100 grids. The resolutions chosen were 6250.0, 1250.0, 125.0, 5.0, and 1.0 square meters. This scheme satisfied the constraints set forth. There is one piece of information that we let slide, 6250 * 100 is not 640000 so the view distance may not seem to be 200 miles. In fact it is even more, the distance from the camera to the farthest terrain point directly North will be under 200 miles, but the distance to the farthest terrain point to the Northeast will be well above 200 miles, so we decided that this constraint was met. Dr. Chappell’s constraint was met by setting the largest grid resolution to 6250.0 meters or 4 miles. This more than doubles the resolution of the previous scheme at the largest layer. Finally the “popping” has been reduced by increasing the distance of the first seam from the camera.

As you may have already noticed each decision made when designing the representation scheme can have adverse effects on other portions of the scheme. If you play with the parameters of a grid layer in order to meet a certain constraint, that affects all other layers, it affects the data structure used to represent the layers, and in turn could possibly break one of the other constraints. A perfect marriage between all parameters is needed in order to satisfy all constraints, and our solution was generated through a trial and error approach.

Texture Generation

The terrain generation algorithm ends with a 1.0 square meter resolution. It is obvious that changes in terrain occur at a far smaller scale and those changes should be taken into account. In order to do so we had to look at techniques that would give the terrain a false sense of detail. These details are generated through the use of different texturing techniques.

Before we can do any texturing techniques we must first have the textures. The beauty of our noise algorithm is that it has a myriad of applications and is not limited to height generation. Instead of generating heights the noise function can be used to generate a color value for each pixel in a texture. As stated in the Putting it all Together section frequency, amplitude, and octave count all affect the characteristics of the output. There are three texturing techniques that were implemented in the final demo. The first is standard multi-texturing, the second is bump mapping, and last the color mapping.

The standard multi-texturing was used for terrain that was distant from the camera. The reasoning is that bump mapping in useless on objects that are far away since its purpose is to fake fine details. This texture was created by generating a grayscale image of light and dark areas. By using grayscale we can lay this texture over some other colored texture and get the effect of rocks or grass depending on the texture it is laid on. For this texture the noise function used a frequency of 1.0 meaning that there should be no interpolation between pixels, the octave count was set to 1, and the initial amplitude was set to 1.0. If you run this through your head what you should be imagining is a texture similar to the noise you see on your television when your signal goes bad. The over all texturing scheme uses 3 textures, one for the close terrain, one for the distant terrain, and one to color mountains, plains, and lakes. Image 7 shows the effect of the noise generated texture when applied to the distant terrain.

The most complicated of the three was the bump mapping. The bump mapping technique implemented takes three steps. First generate a height map. This is simple given the framework we already have available, treat each pixel as if it were a terrain point and map the value between 0 and 255. We chose an octave of 6 and a starting frequency of 0.125. This resulted in the output shown in Image 6. Now we use the height map to generate normals. Each pixel in the height map will have a normal based on the values of its neighbors. The normal map will consist of RGB values that represent the normal for the corresponding pixel in the height map. This is done by letting R=(normal.x + 1)/2, G=(normal.y + 1)/2, and B=(normal.z + 1)/2. This normal map texture is then used by OpenGL extensions to generate the bump map texture.

The final texture is the color map. This is the simplest of the three. A texture is procedurally generated that gradients from blue to green to brown to white. Terrain points then have their texture coordinates constructed based on their heights.

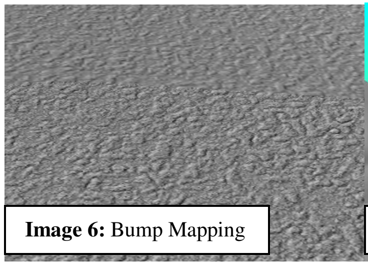


Image 6: Bump Mapping

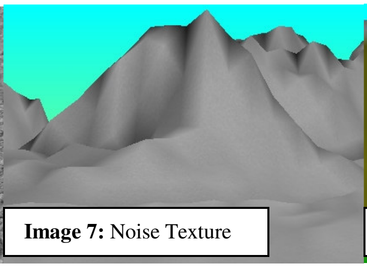


Image 7: Noise Texture

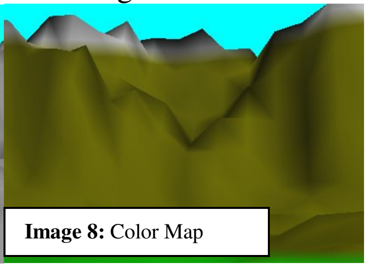


Image 8: Color Map

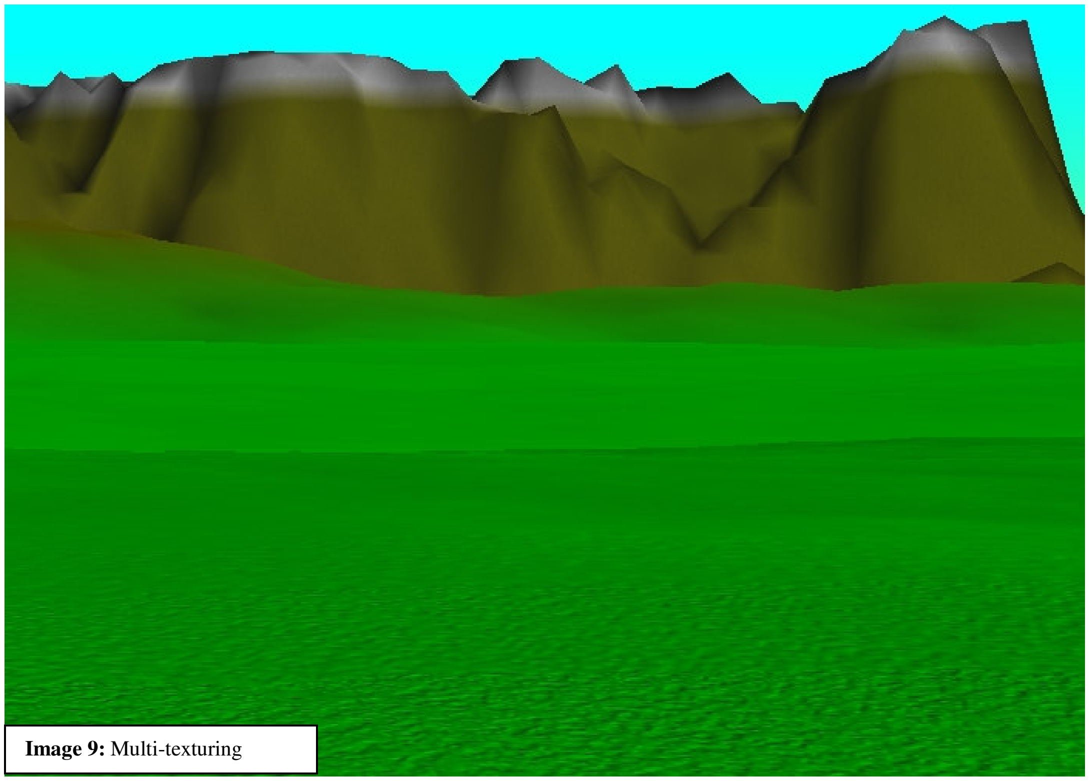


Image 9: Multi-texturing

Future

Terrain Representation

A new idea could be to have x and z be represented as unsigned integers. This would provide us with a world of 4 billion square meters. The problem with this implementation is that the nested grid structure used to represent the terrain would be more restricted than it already is since resolutions are limited to integer only values. We stand by our belief that the nested grid structure is the best way to handle the distance based level of detail, but using arrays may not be the best way to hold the data. A custom data structure that functions similarly to arrays but holds more information could allow for better culling and faster generation

Function Application

We used X^3. There are many other ways to alter the output of the height generator that may produce more Earth-like results. For example, the concept of thresholds could be used where heights within certain ranges are affected by different functions.

Pixel Shader

We used OpenGL extensions that facilitate multi-texturing and bump mapping. This is far slower and far more difficult that using pixel shaders to accomplish the same effects. We were unable to get OpenGL pixel shaders to properly compile so we were forced to use the slower method. Frame rate could be considerable increased if pixel shaders were used.

Culling

The only form of culling used is “Behind Me” culling. Geometry behind the camera isn’t processed. This is a brute force method and does not remove all the unneeded processing. Far more effective culling techniques exist and would increase frame rate dramatically. For example occlusion or frustum culling would be a beneficial addition.

Amplitude and Frequency

By changing the initial values and the rate that they change the characteristics of the terrain will be changed and it may be possible to find numbers that better represent terrain found on Earth.

Reduce Generation

Use the interpolated version of the previous layer as the starting value for the next layer which should reduce the number of octaves needed to generate new terrain points. This should allow dramatically more terrain data to be generated per frame.

Sphere

Drape the terrain over a spherical or ellipsoidal planet. Treating each height as a radius is one possibility.

Performance

System
AMD Athlon 64 X2
Dual Core 4800+
2.4 GHz, 1 GB RAM

Video Card
256 MB ATI Radeon X700 SE

Frames Per Second
Render terrain without movement. 32 FPS
Render terrain and generate data as needed. 21 FPS
Render terrain and generate world each frame. 2 FPS

A Look at the Result

Conclusion

Yes, it is possible to create a program such that no disk loads occur from a real-time renderable, traversable world larger than Earth with a view distance of 200 miles.

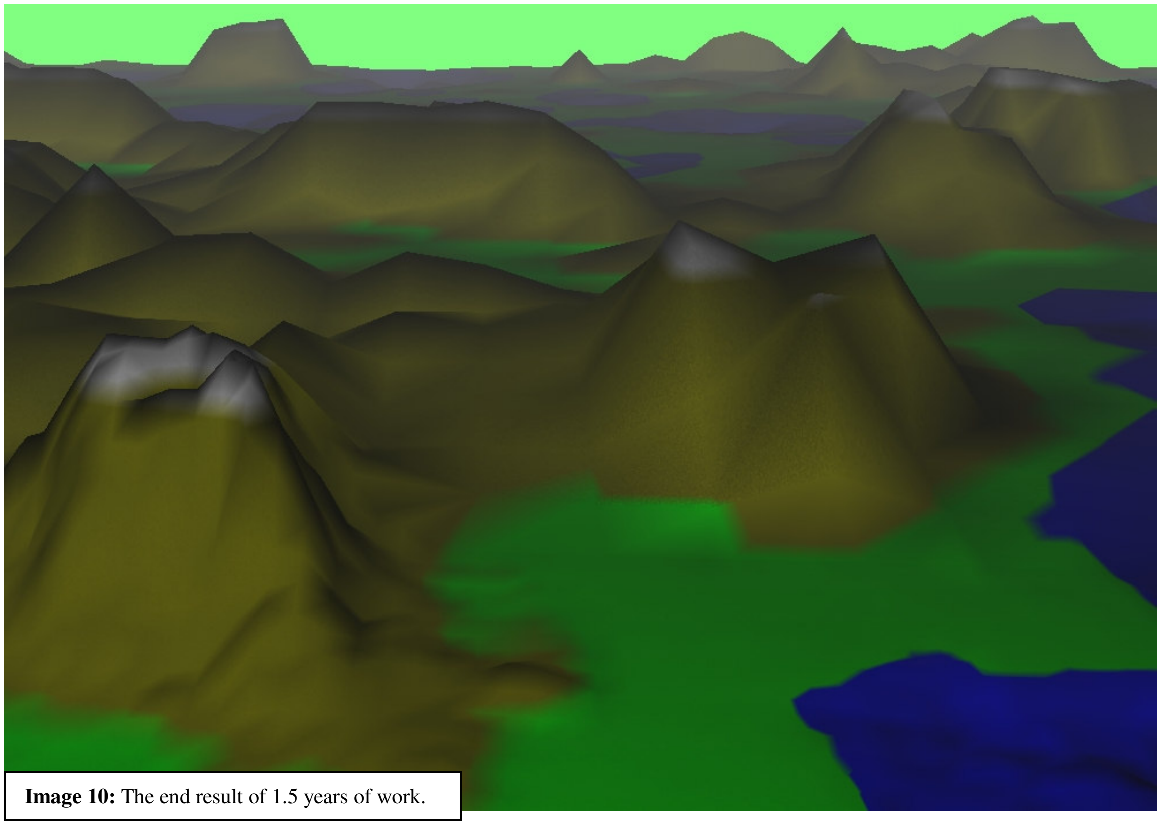


Image 10: The end result of 1.5 years of work.

Bibliography

Benoît Mandelbrot
Stochastic models for the Earth's relief, the shape and the fractal dimension of the coastlines, and the number-area rule for islands.
Proceedings of the National Academy of Sciences (USA): 72, 3825-3828.

Benoît Mandelbrot
The fractal geometry of trees and other natural phenomena. Geometrical Probability and Biological Structures:
Buffon's 100th Anniversary Conference (Paris, 1977).
Edited by Roger Miles & Jean Serra (Lecture Notes in Biomathematics 23).
New York: Springer, 235-249.

Ken Musgrave (MojoWorld)
Texturing and Modeling: A Procedural Approach
CA: MK Publishing, 1998

Peter Lindstrom and Valerio Pascucci.
Visualization of Large Terrains Made Easy.
IEEE Visualization 2001, October 2001, pp. 363-370, 574.

Arul Asirvatham and Hugues Hoppe
GPU Gems: Terrain Rendering Using GPU-Based Clipmaps
NJ: Addison-Wesley, 2005

Ken Perlin “Perlin Noise”
Improving Noise, Computer Graphics; Vol. 35 No. 3.
http://www.noisemachine.com/talk1

Oliver Deussen & Bernd Lintermann
Digital Design of Nature
NY: Springer; 1 edition (May 24, 2005)

Tomas Akenine-Moller & Eric Haines
Real-Time Rendering Second Edition
MA: A K Peters, 2002

hugo@shadow.org.uk “Implementing Noise”
http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

Paul Baker “Simple Bump Mapping”
http://www.paulsprojects.net/tutorials/simplebump/simplebump.html