Real Time Rendering of Complex Height Maps

Walking an infinite realistic landscape

By: Jeffrey Riaboy

Written 9/7/03

Table of Contents 1

I. Overview

The purpose of this paper is to report my discoveries over the last few months from working on my fractal landscape program. Source code is available at http://www.castledragmire.com/Projects/Fractal_Landscape.

To be able to understand this paper and the code, it is necessary to be fluent in 3D theory, C++, and DirectX or OpenGL. I use the left handed perspective system where **X** is to your left, **Z** is straight forward, and **Y** is up. This paper also contains some new theories that I have formulated on the topic.

I programmed implementations of the ray tracer in both the DirectX and OpenGL APIs, which are available at the above URL. After doing extensive testing in both, DirectX appears to be the most affective approach at this point in time. Section III.C.4 contains approximate time test results. All testing was done on a 1.8 GHz P4 laptop with a 7500 mobile Radeon, and 768MB of RAM.

II. Creation of the landscape using fractals

A. Diamond Square Algorithm

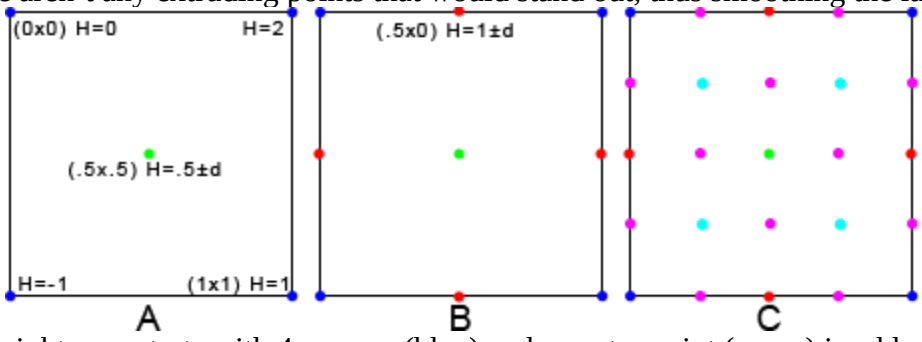
There are numerous approaches to creating fractal landscapes. The approach I use is the diamond square algorithm to generate a height map. I will quickly go over my implementation of this algorithm, but will not extensively be reviewing the diamond square approach as it has already been widely evaluated by others. More in depth explanations of this algorithm can be found online.

The **X** and **Y** coordinates in the 2D height map will eventually remap to **X** and **Z** positions in the 3D fractal landscape world. Height values determine the eventual **Y** position in the 3D world, and are referred to as **H** in the diagrams below.

This approach starts out [iteration level 0] with a square, defined by 4 corner points of varying height values. This square is then divided along the midpoints of its 2 axis to yield 4 square quadrants that are equal in size by adding a center point; See **Figure A** below. The height value of this center point is chosen by averaging the height values of its parent square's 4 corners and then adding a **random value between d and -d (**hereby denoted as **±d, d** is always different for every point). Each of the 4 line segments of the parent square are then halved along their midpoints; See **Figure B** below. These midpoints' (which appear as a diamond in relation to each other) heights are determined by an average of their surrounding points' heights with an added value of **±d**. This process is then repeated on each new area (there are now 4 total), again using **±d** as the added random value after surrounding points are averaged.

This completes the first iteration, leaving you with 16 areas (4 areas per row/column). This means there are 4+1 points on each row and column. Each iteration continues in the same manner by having each new area further divided into 16 new areas, but the range for added random values to heights are reduced by one half on each iteration (**±d/2**, **±d/4**, **±d/8**, …). Around the 4th iteration, it is best to decrease the random divide by a factor of 8 instead of 2.

After the height map is finished, it can also be advantageous to do a quick iteration through all the points, checking each point's surrounding heights and making sure there aren't any extruding points that would stand out, thus smoothing the landscape.



A) The height map starts with 4 corners (blue) and a center point (green) is added, splitting the single square into four quadrants. Example height values (**H**) are given for each corner, and when averaged, produce the *base* height value of **.5** used for the center point (before **±d** is added).

B) Four more points are added (red) in a diamond shape around the green point.

C) The process is repeated a second time, first adding center points to each new quadrant (cyan), then points (purple) in a diamond shape around each of these cyan points. This completes 1 iteration.

B. Coloring your landscape

Now that a height map has been created, there are 2 ways to map the colors onto the landscape that I provide that can make nice effects. The first is using a texture, and the second is making each vertex a specific color based on its height value.

Finding **u/v** texture coordinates for each vertex is relatively simple, as they are all in a standard grid. To get these numbers, you need to know how many vertices there are in each row. The formula for the number of areas in a row is $\mathbf{2^{Iterations*2}}$. Logically, there would be 1 more vertex in a row than the number of areas, so, the formula for the number of vertices in each row is $\mathbf{2^{Iterations*2}+1}$. To find the **u** or **v** coordinate along the **x** or **z** axis respectively, the formula would be the **X**/**Y** coordinate in the 2D height map divided by the number of vertices: **Coordinate/TotalVertices**.

The second approach is taking your height value and turning it into a color. The method I used is as follows.

Heights (0 to 1 scale)	Gradient Color	Shading Direction
0 to 1/8	blue	dark to light
1/8 to 4/8	green	light to dark
4/8 to 7/8	brown	light to dark
7/8 to 1	white	dark to light

C. Lighting

So far we still have a pretty poor landscape visually, as colors appear too uniform. To rectify this, we add lighting. I added a directional light source at (**X, 5, Z)** where **X** and **Z** are determined by the sine and cosine of a circle with a given radius. This can create the simple effect of the changing of time by rotating the light around the world. Adding a textured quad with a sun, or a sun flair, at that point can be a nice effect to let the user knows where it is.

Right now all the light does is use the average of each triangle's 3 vertices to shade the triangle. This is standard Gouraud shading. This means the normal for each vertex must be found beforehand. After the height map is created, the normal for each face (triangle) needs to be found, and from there, a normal is assigned to each vertex based on the average of the normals of the four faces that use each vertex.

The only problem left is some triangles are still fully black since they have a normal pointing in the same direction as the light. This doesn't happen in the real world when the sun is out due to reflection of light off of other surfaces. Radiosity is not a real possibility at this point for the engine, so ambient light on every triangle is the simple solution. Real time shadows are not very realistic at this time but will be discussed in a later section.

D. Other Additions

There are other simple additions one can add to their landscape to make it look better. For example, on mine I added in an alpha blended blue layer at 1/8 height, where the blue vertices stop, to create the water line. When the user's eye moves below this line, the rest of the world turns into shades of blue via fogging. Billboarded bubbles are also produced from the center of the screen when under water. I may also turn this layer into moving water at some point with sine/cosine functions.

I also added 5 cloud layers that move at different speeds that are generated through remapped values of the fractal height map.

Simple things such as bump mapping or billboarded (or even polygonal) trees could also be added for extra affect.

III. Culling approaches for real time speed

5 levels of iteration on a fractal contains 1024x1024 areas, which yields 1,048,576 areas total, or 2,097,152 total polygons. At this point in time, pumping 2 million polygons a second is not easy for a graphics card. For this reason, culling is needed.

A. Quad tree bounding boxes against viewing frustrum

This approach has, so far, proved the most affective. I group together the fractal into sets of 32x32 squares (when at 5 levels of iteration). For clarification, I will hereby be referring to these groupings as "areas", and the squares within these groups will be called sub-areas. This means each area contains 2048 total triangles (32x32 sub-areas * 2 triangles per sub-area) and there are 32x32 total areas on the level 5 fractal. The min and max points for the bounding box for these areas is then easily calculated, as the first and last **x** and **z** positions are already known, and **y** is found by a quick iteration of all the height values in the area from the height map (which I precalculate). Once all the min and max values are known, a simple box of 8 vertices and 6 planes can be made that permits quick and relatively inexpensive checking of the area against the viewing frustrum. The technical term for the bounding boxes of these areas is axis aligned bounding boxes (**AABB**). The viewing frustrum planes and vertices also need to be found every time the eye moves, but this is relatively inexpensive also. By putting each area in a separate vertex buffer and knowing that there are exactly 2048 triangle s in it, calling an area to draw is relatively fast, especially since the buffers are already in video card resident memory.

One way to save space is to use triangle stripping, which has only 1 minor drawback. Generally, triangle stripping is only done row by row in a rectangle mesh. It is not possible to do a whole rectangle mesh (rows and columns) due to the nature of triangle stripping ending on odd vertices. To get around this, I bent the rule, and make one extra triangle per row that is basically a straight line to get to the proper point to start the next row. Since the triangle is a line and has no real area, it will not be drawn. This makes it possible to triangle strip an entire rectangle mesh with no need to make multiple calls to *DrawPrimitives*. The drawback is 31 extra triangles are drawn per area, giving 2079 polygons as opposed to 2048 per area. The major advantage is only 2081 vertices are needed in each vertex buffer, as opposed to 6144 if a triangle list was used. The advantage(s) in this case heavily outweighed the drawback(s).
The formulas for number of vertices that would be in triangle strips and triangle lists are:
List: **Number of Vertices = Number of Triangles * 3**
Strip: **Number of Vertices = Number of Triangles + 2**

DirectX automatically switches around the vertex buffers from resident graphic card memory to RAM when they aren't being used, which causes a minor hit when turning around the first time and viewing the whole world. However, OpenGL, using triangle lists, keeps them all in GPU memory. In my case there wasn't enough memory to hold all 2.1 million vertices and this slowed the rendering down to a crawl (~3FPS). On the other hand, when the iterations were lowered by 1, OpenGL's more advanced native culling algorithms increased the FPS to ~200 while DX went from ~20 to ~60.

The main drawback to using this approach is that a good infinite version is nearly impossible to implement at this time, as new vertex buffers would have to be made and destroyed on the GPU quite often.

B. Occlusion culling using pixel shaders

DirectX has occlusion culling support via the *IDirect3DQuery9* interface and OpenGL has it via the *glFeedBackBuffer*. Unfortunately, neither of these are completely practical. The *glFeedBackBuffer* is too slow to be sensible, and *IDirect3DQuery9* is only supported on graphics cards that support pixel shading. These include nVIDIA's GF3 TI/FX, GF4 TI/FX, and ATI's Radeon 9000+. I did write a version with source code that uses the DirectX query interface, but again, only modern cards support it. It also is not completely finished because I could only develop it over pcAnywhere with friends who had pixel shader compatible cards, which was a little slow. This is used in addition to the frustrum culling, which is executed first.

This approach draws the solid bounding box for each area with writes to the color buffer and z-buffer turned off to count how many pixels would be drawn. Alpha blending could be used instead of turning off the color buffer to minimize GPU state changes. If no pixels would be drawn, the bounding box is not visible; therefore nothing inside it would be either. For areas that are hidden, only needing to draw the 12 polygons, instead of all 2079, and only needing to check the z-buffer, is a vast improvement. However, areas *do* need to be drawn front to back for this implementation. This could also just as easily be written in a pixel shader instead of using the query interface.

C. Ray-Tracing

When people normally think of ray tracing, they think of higher quality but more slowly rendered scenes. Ray tracing has always produced better quality graphics because it traces light and radiosity naturally and also naturally use complex geometric objects, as opposed to just triangle polygons that rasterisation specializes in, but these take a toll in the computational time needed. Normal renderings of single frames of a complex ray traced scene, depending on the number of polygons and resolution, can take hours or even days. Real time ray tracing is becoming a reality, however, as the speed of computers increases. This theory has already been proven by multiple people, including the PC 3D shooter demo *AntiPlanet* (http://www.virtualray.ru/eng/news.html) which uses ray tracing, and spheres as primitives instead of triangle polygons.

i. Explanation of real time ray tracing a height map

The normal sluggishness of ray tracing is due to finding which polygon belongs to any pixel (and subsequent light bounces, which are not relevant to this paper). Normally, a ray would have to be cast from every pixel on the screen in the proper direction and tested against every polygon using barycentric intersection routines. Having to cast 800x600 rays for that many pixels and testing each ray against each polygon is incredibly time consuming, so we need a way to increase the speed.

Since we already know the **X** and **Z** coordinates of every polygon in the whole map due to the grid like construction, if we look at the height map from above and only check the polygons in a 2D fashion against the ray projected into 2D, most of the polygons are already cut out as possible hits. It is relatively easy to move from sub-area to sub-area using floating point numbers and the *modf* function to quickly traverse the height map in this manner. From there, if you take your eye's starting point and see how far above the current sub-area it is, a simple quadratic equation using the **Y** slope of the ray and the maximum amount each sub-area can be above another could be applied that would allow you to extrapolate the first possible sub-area that would be high enough for the ray to intersect.

Furthermore, the program would only have to check each sub-area's intersection at 2 distinct points, where the ray enters and leaves the sub-area, to see if the ray would

intersect on either side. This is much less expensive than the normal ray hit polygon test because each test is just a weighted average of the ray between 2 vertices. Ray tracing also makes it easier to make the world infinite, as no data would need to be stored on the GPU, so only RAM updates are needed.

ii. Advantages of ray tracing including real time shadows

Now that we have the exact point along the ray that a polygon is intersected (**point A**), we can cast another ray from **point A** to every light source using the same height map based method as described above. If any other polygons are hit before the ray reaches the light source, **point A** is not illuminated by that light. This could be used in conjunction with normal dot product lighting calculations to create incredibly realistic effects. Radiosity is also a reality with ray tracing since rays can also be cast from light sources and bounced off of objects, effectively eliminating the need for ambient light. Rays would also be easy to shift to new angles when going through different states of matter to create glass and water effects.

iii. Why the theory doesn't work, but still has potential

While the theory is sound and does reduce time significantly from hours to mere seconds, mere seconds is still not in the realm of real time. The extra effects, like shadowing, would add a good 10-60% per light on the running time. With further optimizations, much more debugging, and more hardware support like there is for rasterization, this could be a reality.

iv. Time comparisons of approaches and APIs

All of these times are approximate, as exact results where hard to get due to the multithreading nature of windows. All of my testing was done on a 1.8 GHz P4 laptop with a 7500 mobile Radeon, 768MB of RAM, and using 5 levels of iteration unless otherwise specified.

Test	Speed (In FPS)
Rasterisation of All Polygons using DirectX	1-2 FPS
Rasterisation of All Polygons using OpenGL	11-15 FPS
Quad tree approach in DirectX	13-22 FPS
Quad tree approach in OpenGL	3-4 FPS
Quad tree approach in DirectX – 4 iterations	~60 FPS
Quad tree approach in OpenGL – 4 iterations	~200 FPS
Quad tree approach in DirectX GF4 FX	20-50 FPS
Occlusion + Quad tree in DirectX GF4 FX	~60 FPS

Test	Speed (In Seconds per Frame)
Ray Traced/Rasterisation Hybrid DirectX:	
Unoptimized ray-triangle intersection	~2.5
No skip area through height approach	~22
Optimized	~1.9
Ray Traced/Rasterisation Optimized OpenGL	~2.3

IV. Creating effects using vertex shaders on the height map

To save on memory and create some high quality effects, vertex shaders can be used after being given the vertex position and vertex normal.

A. Height based

By already having the height position (the **Y** coordinate) the color can be calculated every time a vertex is sent through processing the same way as it is during the preprocessing of the height map.

B. Distance Based

By taking the coordinate of the vertex and the coordinate of the eye (stored in a vertex shader constant) a distance color can be calculated using the distance formula and gradient scaling. $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2}$