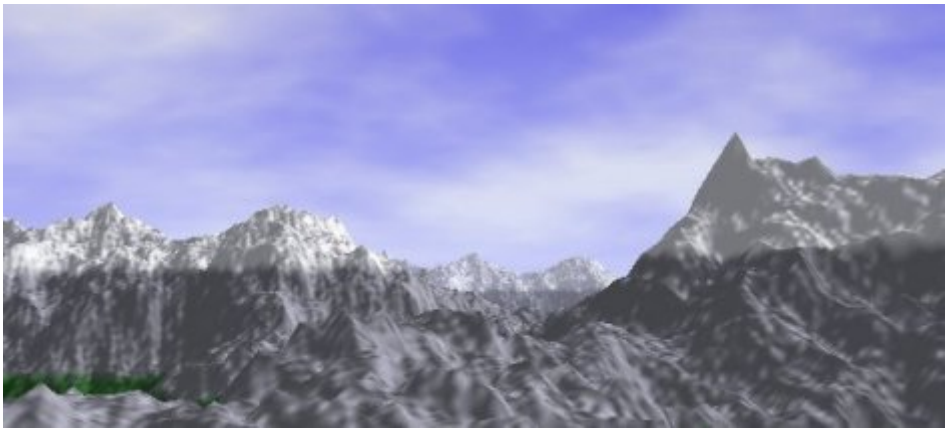
Generating Random Fractal Terrain

Paul Martz
martz@frii.com



Contents

Part I: Generating Random Fractal Terrain

Introduction

Ten years ago, I stumbled across the 1986 SIGGRAPH Proceedings and was awestruck by one paper in particular, entitled *The Definition and Rendering of Terrain Maps* by Gavin S. P. Miller [1] . It described a handful of algorithms for generating fractal terrain, and the authors also introduce a new method which they considered to be an improvement.

Initially I was impressed that these algorithms (even the algorithms considered "flawed" by the authors) could create such incredible landscape images! Then, upon reading the paper, I was floored by the simplicity of these algorithms.

I've been a fractal terrain addict ever since.

The math behind the algorithm can get quite complex. However, completely understanding the math is not a prerequisite for grasping the algorithm. And that's good. Because if I had to explain all the math to you before explaining the algorithm, we'd never get to the algorithm. Besides, there is literally tons of material out there on the mathematical concepts involved in fractals. See the references at the end of this article for a good start.

For the same reasons that I won't go into the math details, I can't include a broad overview of fractals and everything they can be used for. Instead, I'm going describe the concepts behind fractal terrain generation, and give a focused and detailed description of my personal favorite algorithm: the "diamond-square" algorithm. I'll demonstrate how to use this algorithm to statically tessellate an array of height data that can be used for geometric terrain data, terrain texture maps, and cloud texture maps.

What can you do with a fractal terrain? I assume you already know that; that's why you're reading this. Random terrain maps are great for flight simulators or making texture maps to use as a background (showing a distant mountain range, for example). The same algorithm that makes terrain can also be used to generate texture maps for partly cloudy skies.

Before I go further, a disclaimer: I am not a game programmer. If you are reading this because you want algorithms for rendering terrain quickly, you've come to the wrong place. I'll only describe the process of generating the terrain model. How you render it is up to you.

Self-Similarity

The key concept behind any fractal is *self-similarity*. An object is said to be self-similar when magnified subsets of the object look like (or identical to) the whole and to each other.[2]

Consider the human circulatory system. This is a fine example of self-similarity in nature. The same branching pattern is exhibited from the largest arteries and veins all the way down to the smallest capillaries. If you didn't know you were using a microscope, you wouldn't be able to tell the difference between capillaries and arteries.

Now consider a simple sphere. Is it self-similar? No. At significantly large magnification, it stops looking like a sphere altogether and starts looking like a flat plane. If you don't believe me, just take a look outside. Unless you happen to be in orbit while reading this, you'll see no indication that the Earth is a sphere. A sphere is not self-similar. It is best described using traditional Euclidean geometry, rather than a fractal algorithm.

Terrain falls into the "self-similar" category. The jagged edge of a broken rock in the palm of your hand has the same irregularities as a ridgeline on a distant horizon. This allows us to use fractals to generate terrain which still looks like terrain, regardless of the scale in which it is displayed.

A side note on self-similarity: In its strictest sense, it means *self-identical*, that is, exact miniature copies of itself are visible at increasingly small or large scales. I actually don't know of any self-identical fractals that exist in nature. But the Mandelbrot set is self-identical. I won't even go into describing the Mandelbrot set. Go dig up any of the references for more info.

Midpoint Displacement in One Dimension

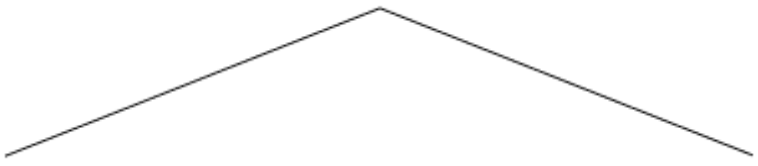
The diamond-square algorithm, which I will describe later, uses a kind of midpoint-displacement algorithm in two dimensions. To help you get a grip on it, we'll look at it first in one dimension.

One-dimensional midpoint displacement is a great algorithm for drawing a ridgeline, as mountains might appear on a distant horizon. Here's how it works:

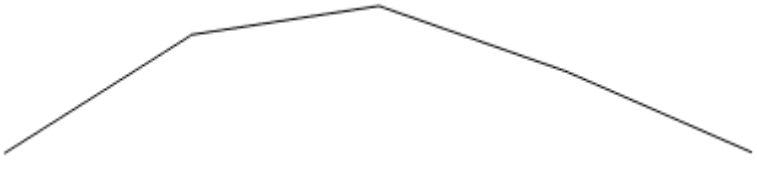
```
Start with a single horizontal line segment.
Repeat for a sufficiently large number of times {
 Repeat over each line segment in the scene {
  Find the midpoint of the line segment.
  Displace the midpoint in Y by a random amount.
  Reduce the range for random numbers.
 }
}
```

How much do you reduce the random number range? That depends on how rough you want your fractal. The more you reduce it each pass through the loop, the smoother the resulting ridgeline will be. If you don't reduce the range very much, the resulting ridgeline will be very jagged. It turns out you can tie roughness to a constant; I'll explain how to do this later on.

Let's look at an example. Here, we start with a line from -1.0 to 1.0 in X, with the Y value at each endpoint being zero. Initially we'll set the random number range to be from -1.0 to 1.0 (arbitrary). So we generate a random number in that range, and displace the midpoint by that amount. After doing this, we have:



Now the second time through the outer loop, we have two segments, each half the length of the original segment. Our random number range is reduced by half, so it is now -0.5 to 0.5. We generate a random number in this range for each of the two midpoints. Here's the result:



We shrink the range again; it is now -0.25 to 0.25. After displacing the four midpoints with random numbers in this range, we have:

Two things you should note about this.

First, it's recursive. Actually, it can be implemented quite naturally as an iterative routine. For this case, either recursive or iterative would do. It turns out that for the surface generation code, there are some advantages to using an iterative implementation over a recursive one. So for consistency, the accompanying sample code implements both the line and surface code as iterative.

Second, it's a very simple algorithm, yet it creates a very complex result. That is the beauty of fractal algorithms. A few simple instructions can create a very rich and detailed image.

Here I go off on a tangent: The realization that a small, simple set of instructions can create a complex image has lead to research in a new field known as *fractal image compression*. The idea is to store the simple, recursive instructions for creating the image rather than storing the image itself. This works great for images which are truly fractal in nature, since the instructions take up much less space than the image itself. *Chaos and Fractals, New Frontiers of Science* [3] has a chapter and an appendix devoted to this topic and is a great read for any fractal nut in general.

Back to reality.

Without much effort, you can read the output of this function into a paint program and come up with something like this:

This could be used as scenery outside a window, for example. The nice thing about it is that it wraps, so you can keep around one relatively small image and completely wrap a scene with it. That is, if you don't mind seeing the same mountain in every direction.

OK, before we go into 2D fractal surfaces, you need to know about the roughness constant. This is the value which will determine how much the random number range is reduced each time through the loop and, therefore, will determine the roughness of the resulting fractal. The sample code uses a

floating-point number in the range 0.0 to 1.0 and calls it **H**. $2^{(-H)}$ is therefore a number in the range 1.0 (for small **H**) to 0.5 (for large **H**). The random number range can be multiplied by this amount each time through the loop. With **H** set to 1.0, the random number range will be halved each time through the loop, resulting in a very smooth fractal. With **H** set to 0.0, the range will not be reduced at all, resulting in something quite jagged.

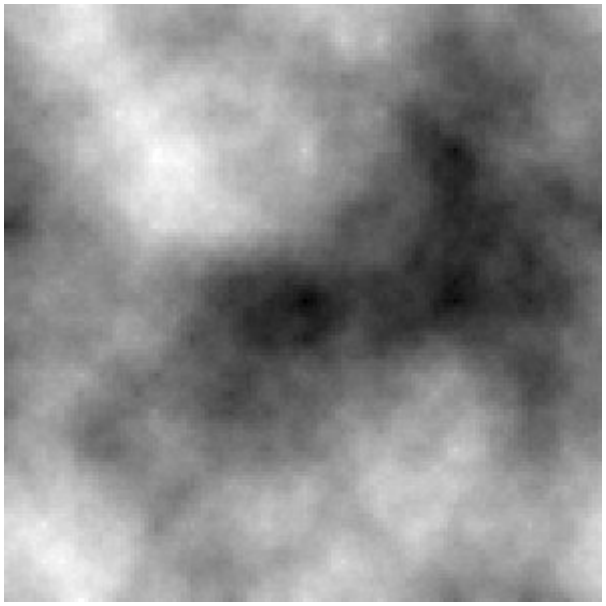
Here are three ridgelines, each rendered with varying **H** values:

Height Maps

The midpoint displacement algorithm described above can be implemented using a one-dimensional array of height values which indicate the vertical location of each line segment vertex. This array is a one-dimensional *height map*. It maps its indices (X values) to height values (Y values).

To simulate random terrain, we want to extrapolate this algorithm into 3D space, and to do so we need a two-dimensional array of height values which will map indices (X and Z values) into height values (Y values). Note that although our end goal here is to generate three-dimensional coordinates, the array only needs to store the height (Y) values; the horizontal (X and Z) values can be generated on the fly as we parse through the array.

By assigning a color to each height value, you could display a height map as an image. Here, high points in the terrain (large values) are represented by white, and low points (small values) are represented by black:



Rendering a height map this way is useful for generating cloud texture maps, which I'll discuss later. Such a representation could also be used to seed a height map.

Now I'll describe how to tessellate our two-dimensional height map array.

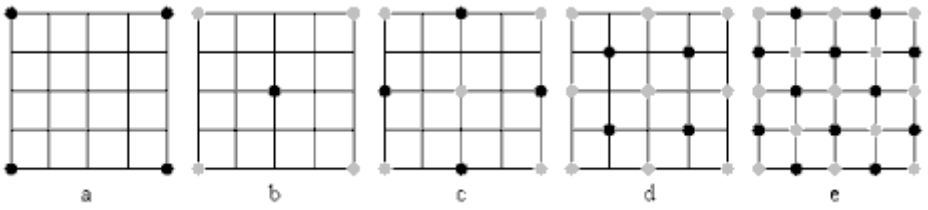
The Diamond-Square Algorithm

As I mentioned at the start of this article, I was first introduced to the concept of generating random terrain in Gavin S. P. Miller's paper [1] . Ironically, Miller describes the diamond-square algorithm as flawed in this paper, and he then goes on to describe a different algorithm based on weighted averaging and control points.

Miller's complaints with the diamond-square algorithm stem from his attempt to force the algorithm into creating a mountain, that is, with a peak, by artificially increasing the height of the grid center-point. He lets all other points in the array generate randomly. If Miller had simply generated the center-point randomly, then even he would've had to admit that the algorithm works pretty decently as a terrain generator. The Diamond-Square algorithm can be used to force a mountain with a peak, by "seeding" the array with values. More than just the center point of the array must be seeded to achieve acceptable results. He complains of some inherent creasing problems as well. But you judge for yourself. The algorithm is originally described by Fournier, Fussell, and Carpenter [4].

Here's the idea: You start with a large empty 2D array of points. How big? To make it easy, it should be square, and the dimension should be a power of two, plus one (e.g. 33x33, 65x65, 129x129, etc.). Set the four corner points to the same height value. If you look at what you've got, it's a square.

As a simple example, let's use a 5x5 array. (We'll be referring to this image later on in the article, so don't forget about it.) In figure *a* the four corner "seed" values are highlighted in black:



This is the starting-point for the iterative subdivision routine, which is in two steps:

> *The diamond step*: Taking a square of four points, generate a random value at the square midpoint, where the two diagonals meet. The midpoint value is calculated by averaging the four corner values, plus a random amount. This gives you diamonds when you have multiple squares arranged in a grid.
>
> *The square step*: Taking each diamond of four points, generate a random value at the center of the diamond. Calculate the midpoint value by averaging the corner values, plus a random amount generated in the same range as used for the diamond step. This gives you squares again.

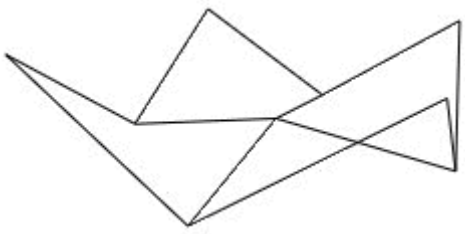
So if you were to seed a square and make a single pass through the subdivision routine, you would end up with four squares. Running it twice would yield 16 squares. A third pass would result in 64 squares. It gets big fast. The number of squares generated is equal to $\mathbf{2^{(I+2)}}$, where **I** is the number of iterations through the recursive subdivision routine.

Referring to the previous five figures, Here's what happens as we make two passes over the array with our diamond-square algorithm.

For the diamond step of the first pass, we generate a value at the center of the array based on the height of the four corner values. We average the four corner values (really not necessary if they were all seeded to the same value), and add a random value from the range -1.0 to 1.0. In figure *b*, the new value is shown in black, and the existing corner values are shown in gray.

For the square step, we use the same range for generating the random values. There are four diamonds at this stage; they all meet in the center of the array, so we calculate four diamond centers. The corners of the diamonds are averaged to find the base for the new values. Figure *c* shows the new values in black and existing values in gray.

That's the first pass. If you were to connect these 9 points with lines, you might get a wireframe surface which looks like this:

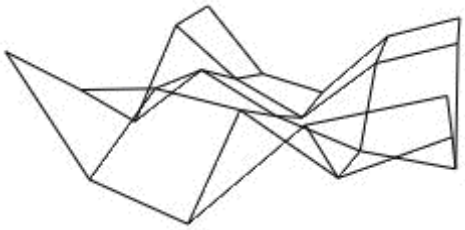


Now we perform the second pass. Again start with the diamond step. The second pass is different from the first pass in two ways. First, we now have four squares instead of one, so we need to calculate four square centers. Second, and this is key,the range for generating random numbers has been

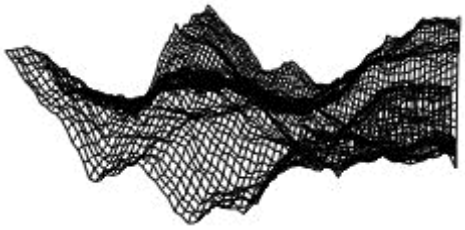
reduced. For the sake of example, let's say we are using an **H** value of 1.0. This will reduce our random number range from (-1.0, 1.0) to (-0.5, 0.5). In figure *d*, the four square center values we calculate at this step are shown in black.

Finally, we perform the square step for this second pass. With 12 diamond centers, we now need to calculate 12 new values. Figure *e* shows them in black.

Now, all 25 elements of the array have been generated. We might now have a wireframe surface which looks like this:



Had a larger array been allocated, we could have continued to make more passes, adding more detail in each pass. For example, after five passes, our surface might look something like this:

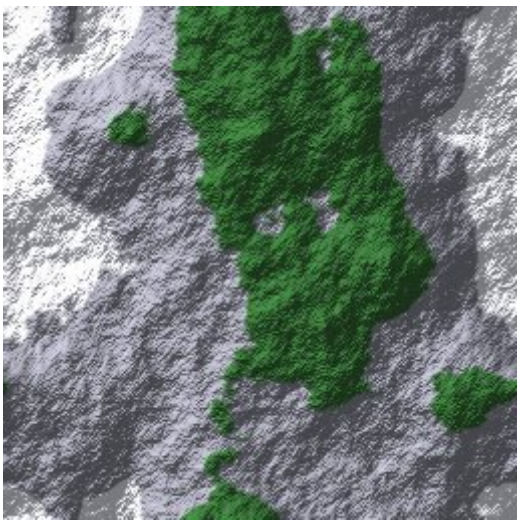


I previously mentioned that the array dimensions need to be a power of two plus one. This is because the number of floats needed in the 2D array is equal to $\mathbf{(2^l+1)^2}$. Eight iterations would require a 257x257 array of floats, more than 256 Kbytes of memory for standard 32-bit IEEE floating-point numbers.

OK, so it's big. Using chars instead of floats would help. The sample code uses floats; if memory is really a concern and you must use chars, it should be easy to modify the sample code to use a range of -128 to 128. But don't forget to clamp those values as you generate them. Even if you limit your first pass to generating values between -128 and 128, subsequent passes could result in values outside this range, resulting in an overflow condition. This is especially likely for small values of **H**.

The sample code demonstrates another way to deal with the size problem. A large array is allocated and tessellated with the diamond-square algorithm. It is then rendered from a top-down orthographic view. This image is read back and used as a texture map on a second array tessellated to a lesser extent. Although the sample code doesn't do this, once the image is read back from the framebuffer, the first array can be freed.

Here's an example of one such texture map:



The map is artificially colored with white at the peaks, green in the valleys, and gray in between. Feel free to experiment with your own color scheme using the example source code provided.

Earlier I had mentioned that there are advantages to implementing this routine iterative rather than recursive. Here's why: A recursive implementation might take the form:

```
Do diamond step.
Do square step.
Reduce random number range.
Call myself four times.
```

That's a nice simple implementation, and I have no doubt that it would work. But it requires that some points be generated with insufficient data. Why? After the first pass, you'll be called upon to perform the square step without having all four corners of a diamond.

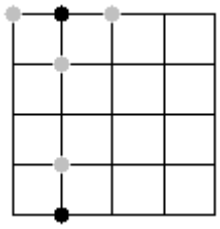
Instead, I've implemented this iteratively, and the basic pseudocode looks like this:

```
While the length of the side of the squares
is greater than zero {
Pass through the array and perform the diamond
step for each square present.
Pass through the array and perform the square
step for each diamond present.
Reduce the random number range.
}
```

This eliminates the problem of missing diamond corners found in the recursive implementation. But you'll run into this problem again anytime you generate a point on the edge of the array. It turns out this is only a concern in the square step. You can easily overcome this and simultaneously make the surface wrappable by taking into account that one of the four diamond corner points lies on the other side of the array. (Another key to making the surface wrappable is to remember to seed the four corners with the same value.)

Here's an example of taking a diamond corner from the other side of the array. In the following figure, we generate a point in the square step, and it just happens to fall right on the edge. The four locations in the array which

comprise the diamond corners are highlighted in gray. They need to be averaged to find the base for the new value, shown here in black.



Note that two values are highlighted in black. They are actually the same value. Every time you calculate a value on the edge during the square step, make sure to also store it on the opposite side of the array. These points need to be exactly the same in order for seamless wrapping to occur.

This means that in figure *e* earlier, we really didn't need to calculate 12 separate values, since four of them are repeats of other values on the opposite side of the array. So actually, only 8 values needed to be calculated.

I'll leave this as an exercise to the interested reader: Take the source code and make it work without having repeated elements on the edges. It's really not necessary for the algorithm to work; it just happens to be the way I wrote it.

If you haven't played with the sample program yet, now might be a good time to open it up and have a look. It starts up with a surface generated with two iterations. It is rendered in wireframe, simply by connecting the values of the array with line segments. The array values are treated as Y values, while the X and Z coordinates of each vertex are generated on the fly as the array is parsed. This could easily be rendered as triangles by breaking each square up into two triangles. Triangles are generally good primitives to use since they are always convex and always planar.

Go into the *View Options* dialog box and tweak the *Random seed* value. This should cause a different surface to be generated. Tweak the *Iterations* value a little higher to add more detail to the surface. The code limits this value to 10, which is a little much for my 32 Meg RAM Pentium Pro system and just looks black anyway. (Five years from now, people will run this code on new processors and higher resolution screens and wonder why on Earth I limited that to 10...)

The first *H* value controls the roughness of the surface. By default it is set to 0.7. Try setting it higher and lower and note the results.

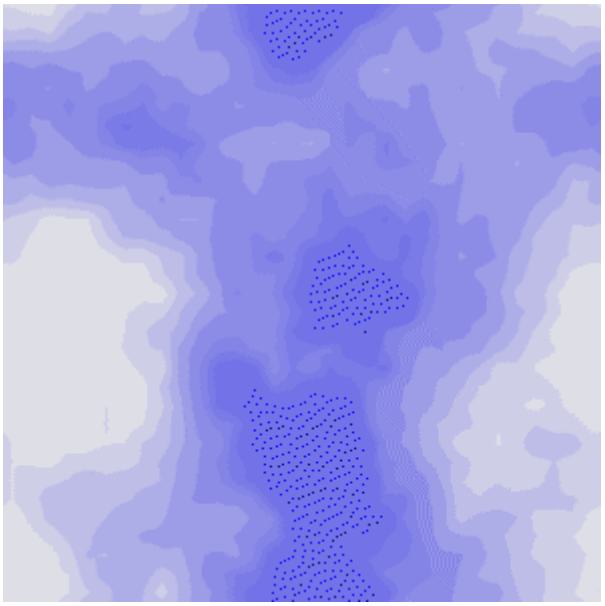
Yes, this algorithm does occasionally produce localized spikes and some creasing. But I tend to like the surreal nature of the spikes, and the creasing is not obvious depending on what angle you are viewing it from, or how fast you're flying over it.

Cloudy Skies

Now we know how to generate the surface. We can either generate and render tons of triangles, or we can generate a high-res texture map and apply it to a low-res surface. Either way, it looks pretty cool. Now how do we generate some sky overhead? It's easier than you think.

The array, after being completely tessellated by the diamond-square algorithm, is ideally suited for representing a texture map of a cloudy sky. Instead of thinking of the array as a collection of Y values in a height map, think of it as cloud opacity data. The smallest array values represent the bluest, clearest parts of the sky, and the largest array values represent the whitest, cloudiest part of the sky.

It's trivial to parse through the array and generate a texture map like so:



This is very similar to the height map image earlier in this article, but I have clamped the low and high values to create patches of clear and clouded sky.

You can produce an image like this with the sample code. Set the *Select rendering type* pulldown menu to *2D mesh / clouds*. (By default it will look pixelated. Try setting the *Cloud iterations* value to eight or higher to fix this.) Try different settings for the *H* value, immediately preceding the *Cloud iterations* value, to get different cloud effects.

If you go back up to the top of this article, the very first image puts together much of what I've discussed here. The sky is made with a texture map as shown above, tiled multiple times over an eight-sided pyramid. The surface geometry is rendered with a high-res texture map. This texture map was generated by rendering a highly-tessellated lit surface from a top-down orthographic view. The image was then read back and used as the texture map.

The accompanying example program was used to generate nearly all of the images that appear in this article.

Other Methods

You'll probably want to have a little more control over the surface generation than what the sample code has provided. You might, for example, want to initially seed the first few passes of the array with your own values, so that mountains, valleys, etc., can be placed according to your own design. Then, fill out the rest of the detail using the diamond-square algorithm.

Don't just kick the central array point upwards like Gavin Miller did, to create a mountain. To get reasonable results, you'll need to seed at least two or three

passes of the array.

This is easily accomplished by altering the code to skip over assigning values to elements of the array that already have values. Initialize your array to, say, -10.0, seed the first few iterations with your own values, and enhance the fractal generation code to only assign values where the current value is -10.0. The first few iterations will not generate any values, since your seed values are already there. Subsequent passes will generate new values based on your seed values.

How to get the seed values? Well, if the shape you want follows a known mathematical form, such as a sine curve, then simply use that function to generate the values. Otherwise, you'll need to find some other creative way to do it. One method I have seen used is to paint your own height map with gray values. Map the gray values to height values and store them in your array. Then use the diamond-square algorithm to add more detail.

Besides the diamond-square algorithm, there are plenty of other methods for tessellating surfaces.

With successive random addition, a random region of the 2D array is incremented by a small amount. Repeat this process over and over, adding a small amount into each randomly chosen region of the array. This generates good results but is not computationally linear. If compute time is not a concern, I encourage you to try this algorithm out.

Another similar method involves making a "fracture" across the array and incrementing one side of it, as if an earthquake had occurred. Again, repeat several times. This is also not a linear algorithm and takes several passes to get acceptable results.

Refer to the references for many other different approaches.

Part II: About the Example Source Code

Installation

The example source code comes in a zip file. Use your favorite zip software to unzip the archive. If you don't have a zip utility, try PKware.

The source code uses the OpenGL API for rendering. If you do not already have OpenGL on your machine, you'll need to get it. Both Microsoft and SGI have OpenGL available for Windows 95, however I encourage you to get SGI's (from here). It is superior to Microsoft's in terms of performance and robustness. Also, the sample code is linked with SGI's implementation, and since SGI and Microsoft chose to give different names to their DLL files, the code will expect SGI's DLL names.

Here's a "how do I make a cool picture without knowing what I'm doing" guide:

> Double click the *Fractal Example* icon.
>
> Open the *View Options* dialog box.

From the *Select render type* pulldown menu, select *2D mesh / rendered*.

Enter an *Iterations* value of 4.

Enter a *Tile* value of 3.

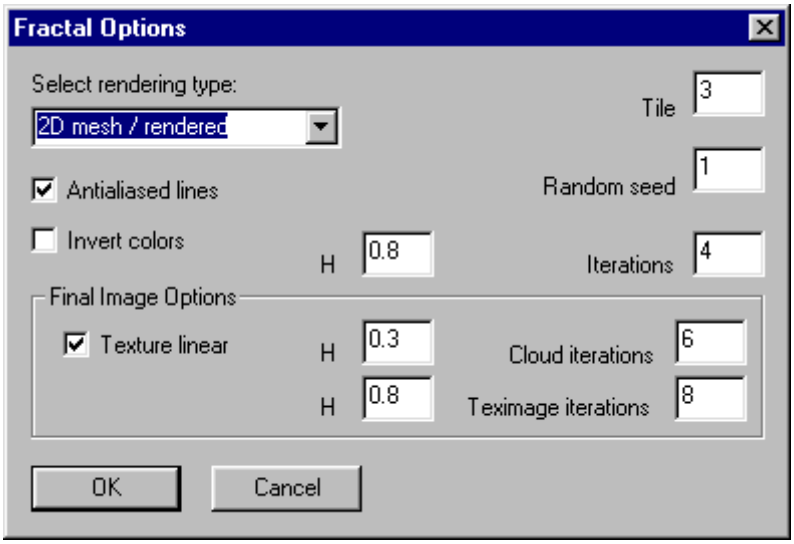
Click *OK*.

Using the Program

By default, the code starts up displaying a 2D mesh in line mode. It has been tessellated with the diamond-square algorithm. Two passes were made over the surface resulting in eight squares.

You can change your viewpoint with the arrow keys. Rotate the surface around using the left and right arrow keys, move the surface up or down by shifting the up and down arrow keys, and move forwards or backwards over the surface using the (unshifted) up and down arrow keys.

Now bring up the *View Options* dialog box. Use the *View* menu, or Ctrl-O. This is where you change what you are viewing and how it is generated. The dialog will look something like this:



The *Select rendering type* pulldown menu determines what is displayed. *1D midpoint displacement* renders a line segment tessellated using the midpoint displacement algorithm.

All render types starting with *2D mesh* use the diamond square algorithm to generate the image. *2D mesh / lines* renders a surface in line mode. *2D mesh / rendered* displays the 2D surface, texture mapped with the current surface texture map (which the program refers to as the "teximage"), under a sky texture mapped with clouds. *2D mesh / clouds* allows you to display only the cloud texture map. This is simply a 2D height map, using blue for small values and white for large values. *2D mesh teximage* allows you to display only the teximage that covers the surface when in *rendered* mode. This is a lit surface with a top-down orthographic projection, using different colors to represent different heights on the surface.

You can control the generation of all these images using the parameters on the right half of the dialog box.

The *Tile* parameter determines how many surfaces or lines are tiled. By default this value is one. For lines, the value determines how many lines to tile. For surfaces, setting this value to two creates a 2x2 tiling of surfaces; three creates a 3x3 tiling, etc.

The *Random seed* parameter sets a new random seed, thus creating a different surface.

The *Iterations* parameter determines how many tessellation passes are made over the surface or line being displayed. Bigger numbers make more detail. By default it is set to two. The value can be in the range 1 through 10.

You can set the level of detail of the cloud texture map using the *Cloud iterations* parameter. Likewise, the level of detail of the texture map used on the surface is set with the *Teximage iterations* parameter.

Note there are three *H* values. The first is used to generate the surface, the second is used to generate the cloud texture map, and the third is used to generate the surface's teximage texture map.

When *Rendering type* is set to *1D midpoint displacement* or *2D mesh / lines*, *Antialiased lines* toggles antialiased line mode.

Invert colors swaps the background and line colors.

When *Rendering type* is set to *2D mesh / rendered*, *Texture linear* toggles bilinear texturing on and off. Having this on in general results in a slower but higher quality image.

Code Structure

fractmod.c and fractmod.h are the C code heart of this example program. They comprise the fractal generation module.

The CFractalExampleView class is derived from the COpenGLView class as found in the November 1996 Microsoft Journal: http://www.microsoft.com/msj. The COpenGLView class was written by Ron Fosner, who describes it as a hacked version of his fully-blown COpenGLView class. To get the real thing, buy his book *OpenGL Programming for Windows 95 and Windows NT* published by Addison-Wesley.

The COpenGLView class has a RenderScene virtual member function which we override in CFractalExampleView. Here we do most of the rendering work. The function first examines the setting for *Rendering type*. When set to *2D mesh / lines* or *1D midpoint displacement*, the work is handled here in RenderScene. Otherwise, another function is called.

CFractalExampleView::OnViewDialog generates the *View Options* dialog box and handles setting and retrieving data between the dialog box class and CFractalExampleView.

CFractalExampleView::OnInitialUpdate handles setting all CFractalExampleView member variables to their default values (including dialog box values).

There's really not much point in explaining further how the code works. I assume you are a competent programmer, and I've done my best to comment

the code fully. If you are not familiar with OpenGL, you might like to know that all functions starting with "gl" are OpenGL API calls. Microsoft Visual C++ has some limited documentation on this API.

There is one feature that is just begging to be added to this code. In the file Fractal ExampleView.cpp, there is a preprocessor constant called DEF_HEIGHT_VALUE. This is passed in to the fractal generation functions in fractmod.c where it is used to scale the height values. This should really be a variable, controllable by the *View Options* dialog box. Feel free to add this feature.

References

[1]Miller, Gavin S. P., *The Definition and Rendering of Terrain Maps*. SIGGRAPH 1986 Conference Proceedings (Computer Graphics, Volume 20, Number 4, August 1986).

[2]Voss, Richard D., *FRACTALS in NATURE: characterization, measurement, and simulation*. SIGGRAPH 1987 course notes #15.

[3]Peitgen, Jurgens, and Saupe, *Chaos and Fractals, New Frontiers of Science*. Springer-Verlag, 1992.

[4]Fournier, A., Fussell, D., Carpenter, L., *Computer Rendering of Stochastic Models*, Communications of the ACM, June 1982.
