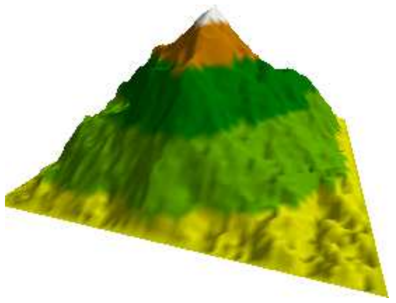
CSci 490, Spring 2005, Assignment 4

This assignment, worth 40 points, is due at 3pm, Friday, February 18. *Submit it by attaching your modified files to an e-mail to* `cburch@cburch.com`*.*

Your job in this assignment is to write a program using OpenGL that generates and draws a random terrain for a volcanic island that is, improbably, square.



From the Web page you can download a set of files that are useful for structuring your code.

`terrain.c` (and its header file `terrain.h`) contains the `terrainInit` and `terrainDraw` functions that you will write for this assignment. This is the only file you should modify.

`main.c` uses `terrain.c`'s functions to show a window allowing the user to navigate up and down and around the island using the arrow keys. This code also sets up the lighting for the island.

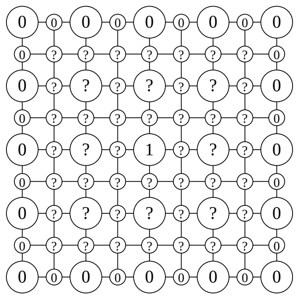
`random.c` (and its header file `random.h`) contain the `randomGaussian` function, which generates a random number drawn from the normal (Gaussian) distribution. As mentioned below, drawing random numbers from the normal distribution is important for the terrain generation process.

I recommend that you complete this assignment in three steps. First, make sure that the program correctly generates and draws the terrain's surface. (Generating the terrain is the hardest part.) Then compute normals so that shading occurs correctly. And finally, add color. The following three sections explain each of these steps.

Step 1: Random terrain generation

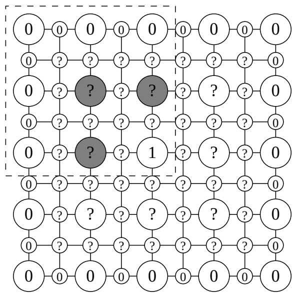
Generating realistic-looking terrain is important in generating artificial environments for movies and computer games set in fictional locations. Designers have created a variety of techniques for this. We'll use a simple and reasonably good approach called **midpoint displacement**.

We will represent the terrain using an evenly spaced grid, with each height being a number between 0 and 1. We want our end grid to have 0 along the borders (where the island meets the sea) and 1 in the center (at the mountain's peak), so we'll go ahead and nail that down now.



(There is no real distinction between large and small circles, although as we shall see, they do indicate the different levels of the algorithm.) The example in this handout uses an 8×8 grid, with $8 + 1 = 9$ points along each side. Your program, however, should scale to finer grids by simply altering the `TERRAIN_WIDTH` constant in `terrain.h`. You can assume that `TERRAIN_WIDTH` is a power of 2.

We first work on the upper left quadrant of the grid. We're particularly interested in three points of this quadrant: The midpoint of the right side, the midpoint of the bottom side, and the quadrant's center.

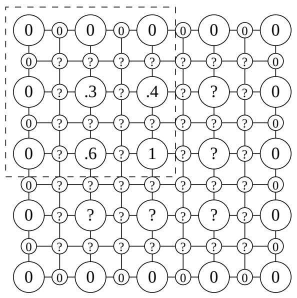


For the right side's midpoint, we assign a height to be the midpoint of the quadrant's adjacent corners, skewed by a random value:

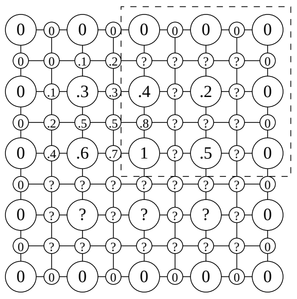
$$\frac{0+1}{2} + roughness \cdot width \cdot random$$

The two corners surrounding this midpoint have heights of 0 and 1, hence the first term is the average. The second term is the product of three variables, $roughness$, $width$, and $random$. The $roughness$ variable, a constant used throughout the algorithm, configures how variable the surface should be. You should use `terrain.h`'s `ROUGHNESS` constant for this; you can play around with different values, but I found that 0.25 worked relatively well. The $width$ variable represents the width of this quadrant; this should be measured in world coordinates. (The grid will be mapped into a 1×1 quadrant.) And the $random$ variable should be a random number chosen from the Gaussian distribution (a bell curve) centered at 0 with a variance of 1.

We choose the value for the quadrant's bottom similarly. For the quadrant's center, we use the average of the four midpoints, again skewed by a random amount. After completing this process, our grid is now the following.



Once we have the top left quadrant, we can repeat the process for the top right quadrant. This time, since the right midpoint lies on the far right side, we end up not changing that value because we want it to be nailed down to 0. But we will determine values for that quadrant's bottom midpoint and center.

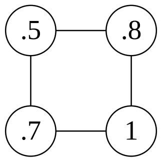


Then we handle the bottom left quadrant, and finally the bottom right quadrant. The ordering among quadrants is significant: Computing the top right quadrant's center, for example, requires the midpoint of that quadrant's left side, which is determined in the process of computing the top left quadrant.

With the four quadrants' midpoints committed, we can proceed to computing midpoints for squares half that size. One could do this either by handling all four quadrants at once, then proceeding to the quadrants' quadrants, and their quadrants, and so forth. Or one could do this by recursing as we compute each quadrant. (Personally, I used the latter approach, which is why the small circles in the top left quadrant of the diagram above are completely filled in before filling the top right quadrant.)

Step 2: Normal calculation

Of course, a good picture will have some shading, which OpenGL will determine if you can provide a good normal for each grid point. A simple way to do this is to consider each point along with its neighbors to the south and to the east. (The following is taken from the lower right corner of the upper left quadrant above.)



For the point at height 0.5, we can represent the change in the x-direction using the vector $(\frac{1}{8}, 0, 0.3)$ (The $\frac{1}{8}$ coordinate comes from the fact that the 8×8 grid is mapped to a 1×1 area, so each grid square is $\frac{1}{8} \times \frac{1}{8}$. The 0.3 z-value comes from the difference of 0.8 and 0.5.) Likewise, the change in the y-direction can be understood to be $(0, \frac{1}{8}, 0.2)$. The normal at the point with height 0.5, then, would be the cross-product of these two vectors, normalized (i.e., scaled so that the cross-product's length is 1).

A really nice program would support shadows, too. But don't worry about this: OpenGL doesn't really support shadows well.

Step 3: Coloring

A black-and-white image does not look like a real island. A simple way to add a hint of realism is to color each vertex according to its height.

- If the height is 0.9 or greater, we draw the vertex white (for the snow cap).
- If the height is between 0.7 and 0.9, we use brown (for the rock above the tree line).
- If the height is between 0.4 and 0.7, we use dark green (for a coniferous forest).
- If the height is between 0.1 and 0.4, we use light green (for a deciduous forest).
- If the height is below 0.1, we use yellow (for the beach).