Lab 5 - Terrain

Support code: /course/cs123/src/labs/lab05
Demo: /course/cs123/bin/cs123_lab05_demo

Intro
In this lab we will learn one way to generate mountainous terrain like in the image above. There are many terrain generation algorithms out there. We will be making ours from a noise function, a technique used for a wide variety of applications beyond procedural terrain.

The support code already draws a flat 100x100 grid of vertices. To make our terrain we will be adjusting the height of each vertex in this grid. Note that this strategy does not allow for certain geological features like caves, but it can nonetheless produce some compelling scenes.

Noise Functions
A noise function is a function that generates pseudo-random values. The simplest type of noise is white noise, in which each point has a random value independent of its neighbors. However, for applications like generating terrain, we want there to be some amount of coherence between neighboring values, so that there are continuous features like mountains.

In 1983 Ken Perlin developed a noise function called Perlin noise, which enabled a wide variety of natural-looking effects to be produced with computer graphics. (The algorithm even won him an Academy Award for Technical Achievement in 1997!)

For this lab we will be implementing a noise function called value noise, which is slightly simpler than Perlin noise and produces similar results.
Getting Started
Copy the support code into a personal directory and run it. You should see a flat, wireframe grid. You can drag and scroll with the mouse to adjust the viewpoint.

We will be working entirely in terrain.cpp. Take a look at the method `getPosition`, which should return the object-space position of a vertex in the terrain. The inputs `row` and `col` are values between 0 and 99 representing where the vertex is on the grid.

We have provided a helper function called `randValue`, which returns a pseudo-random value between -1.0 and 1.0. Note that passing the same arguments to this function will always return the same value.

Task 1:
In `Terrain::getPosition`, set the y-value to a random value using the row and column. This is equivalent to using white noise. Run the program – you should see an extremely jagged surface. Time to improve it!

Smoothing It Out
Instead of generating a different random height for each vertex, we'll use a smaller number of random heights and linearly interpolate between them. We can think of this as scaling up the grid of random values.
For each terrain vertex, you will linearly interpolate between four grid values. This is called **bilinear interpolation**, because you are interpolating in two dimensions. Here’s a diagram to show a sample interpolation calculation between four values A, B, C, and D:

![Bilinear Interpolation Diagram](image)

For a given terrain position, we will use the following steps to calculate its height:

1. Divide `row` and `col` by 20 to find the vertex’s position on the upper-left 5x5 grid.
2. Use the position and the `randValue` function to get the four nearest height values.
3. Use bilinear interpolation between the four heights to compute the vertex’s height.

Some GLM functions that you may find useful to do bilinear interpolation are:

- `glm::mix(float a, float b, float x)`: Linearly interpolates between `a` and `b`, where `x` in the range `[0, 1]`. Returns `(1 - x)*a + x*b`.
- `glm::fract(float x)`: Returns the “fractional” part of `x`. `glm::fract(3.14)` returns 0.14.

**Task 2:**

In `Terrain::getPosition()` follow the steps above to compute the height using bilinear interpolation. Once you’ve done that, you should see a grid that looks something like this:
We’re getting closer! You’ll notice that right now, the peaks and valleys look really pointy. To fix this, we’ll switch from using bilinear interpolation to **bicubic interpolation**. Bicubic interpolation uses cubic functions instead of lines to transition from one value to the next.

Switching from linear to cubic interpolation is surprisingly easy! If you are using `glm::mix`, all you have to do is remap the interpolation value $x$ to the expression $(3x^2 - 2x^3)$, as shown below.


glm::mix(A, B, x)  
glm::mix(A, B, x*x*(3-2*x))

Note that the expression $(3x^2 - 2x^3)$ leaves the values 0 and 1 unchanged. Also note that its slope is 0 at both $x=0$ and $x=1$. This is why it is a good function for smooth interpolation. All the peaks and valleys will be flat, eliminating sharp edges from slope discontinuities (note that this is how `glm::smoothstep` works).

**Task 3:**
Switch to bicubic interpolation by remapping the interpolation values. You should now see a curved surface that looks something like this:
Computing Normals

Before adding more improvements to the terrain shape, let's make sure the terrain is shaded properly. The support code shaders already implement Phong lighting (without the specular component). Your job will be to compute the normal vectors for each vertex.

Let's start by turning off the wireframe.

**Task 4:**
In `Terrain::init`, change the last argument of `glPolygonMode` from `GL_LINE` to `GL_FILL`. This tells OpenGL to rasterize the triangles as filled-in shapes rather than lines. Notice that the terrain is currently just a solid color because the normals are all set to `(0, 1, 0)` in `getNormal`.

We want the surface edges to appear smooth instead of faceted. (Think sphere in Shapes.) This means that each normal vector should take into account the normals of all neighboring triangles.

Consider a vertex $P$ surrounded by eight neighboring vertices, $n_0$ through $n_7$. To estimate the normal vector at $P$, average the normal vectors of the eight neighboring triangles.
The normal vector of a triangle is equal to the normalized cross product between two of its edges. For example, the normal of triangle \((P_{n_0} n_1)\) above would be \((\vec{P_{n_0}} \times \vec{P_{n_1}})\) normalized. Note that crossing in the other order will make the normal go in the opposite direction. Remember the right hand rule!

**Task 5:**
*In Terrain::getNormal, compute the normal vector by taking the average of the normals of the eight neighboring triangles. You should use getPosition to get the position of the vertex and all its neighbors. Feel free to ignore edge cases. And don’t forget to normalize when appropriate!*

Your terrain show now be shaded like so:

![Terrain image](image)

**Noise Octaves**
Now let’s return to the terrain shape. It’d be nice if it had some smaller bumps on top of the larger ones that we have.

How can we make smaller bumps? If we think of the noise function as a wave, the “smaller” bumps will have a higher frequency and a lower amplitude. Think about how you can control frequency and amplitude given the technique we used in getPosition. We already adjusted the frequency once (the “white noise” version from Task 1 had a really high frequency).

Once smaller bumps are created, you can simply add them to the larger ones! Adding levels of noise like this creates **fractal noise**.
How much should we increase the frequency between each level? Typically noise functions sequentially increase the frequency by a factor of 2. In this case, the noise levels are called octaves. (In music, an octave separates two notes with frequencies that differ by a factor of 2.)

How much should we decrease the amplitude between each level? Some possible choices include:
- dividing the amplitude by a constant factor at each level, or
- dividing the amplitude by sequential powers of 2 at each level.

Feel free to choose any function you like, as long as the end result looks like terrain.

Task 6:
In Terrain::getPosition, add together at least three octaves of noise to create a fractal noise pattern.

Once you’ve created fractal terrain, get checked off by a TA!