Height map

A height map or a height field is described as a surjective function f, where

f: R^2 |→ R

is used in a mapping from 2 to 3 dimensions. Given a coordinate (x,y), f returns a height value z, such that f describes a field of height values. For every coordinate (x,y) we can obtain a 3-dimensional vector describing a point by a function h

h(x,y) := (x, y, f(x,y))

so that f is our height function, and h describes a plane in 3 dimensions.

In 3d graphics applications a height map is usually used to model a horizontally aligned surface mesh, such as a body of water or a piece of land, where f can e.g. be a parametric function (to model parametric surfaces), or a lookup into an array of predefined height values, where (x, y) is a discrete coordinate.

Introduction

We have looked at a variety of graphics techniques, including lighting models, shadow algorithms, model loading, etc. This tutorial will focus on putting it all together to create realistic 3d terrains. Several steps are required, however, to model a 3d terrain, the first of which is the definition of a height map. Height maps are used everywhere and are a really common solution for modeling landscapes in 3d applications. The concept is very simple – to model a landscape we represent it as an array of height values, and use a coordinate to lookup a height value.

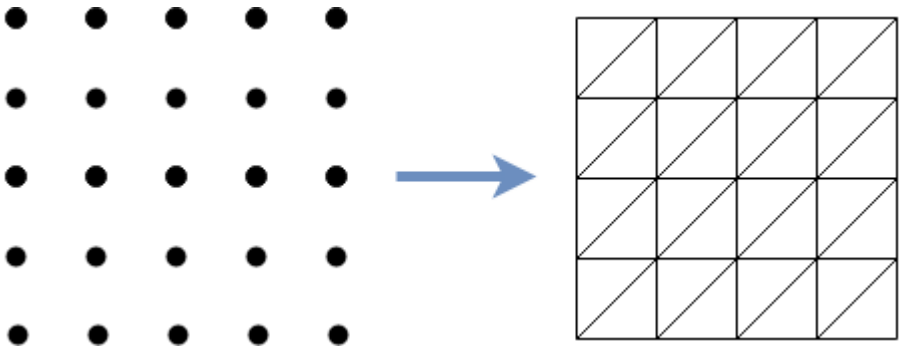
If we model our array in 1 dimension, which is most typical, we use

```
class HeightValue
{
public:
    Glfloat Height;
    enum TerrainType Terrain;
};

const int HEIGHT_MAP_SIZE = 32;
HeightValue heightMap[HEIGHT_MAP_SIZE];
```

Here we specify the data type for each data point in the height map. We could easily just define the height map as an array of floats, but by using a class as here we can use any set of values for each height point, such assigning the terrain type (water, grass, dirt, rock, etc.), biome information, information on how to do texture blending, data for generating foliage, or even create functions for extracting values based on external parameters. This will prove very useful later on when we wish to populate our height map with landscape textures, trees, foliage, animals, etc.

Now we just need to figure out how to pass this information on to OpenGL in a useful way. Since OpenGL is a rasterization pipeline based on triangles, we need to convert our height field into a triangle mesh such that we can render it, not as individual data points but as a contiguous segment of land. Thus, we need a method for constructing triangles from the height map as seen below:



To the left we see the abstract representation of our height field, where each point is a single height point. What we wish to obtain is the image on the right, where a triangle grid is formed as to connect each grid point into a mesh. We say about the grid that it is “discrete uniform grid”, meaning that the distance between each grid point is the same and since each grid point (x,y) is defined as non-negative integer coordinates.

There is a bit of math involved in this process, though it is not too complicated. First, we need to create a data structure for containing the triangles. As demonstrated in earlier tutorials, indexed rendering can be used for 3d objects where many triangles share corners, such as to minimize GPU memory usage. It’s quite easy to tell OpenGL to read vertices from a buffer 3 at a time and connect them together as a triangle (this step is called the ‘primitive assembly’), so we need to create an array and then add the indices for each triangle corner, in a sequential order.

Given a height field of size N * M (where N can be the same as M), the number of triangles T in the mesh is computed as

$T = (N - 1) * (M - 1) * 2$

And the number of indices I is

$I = T * 3$

The code for generating the triangle indices looks, then, as following:

```
// TODO: Fill vertex array with height positions
std::vector<glm::vec3> vertices;
vertices.reserve(N * M);
for (int y = 0; y < N; y++) {
    for (int x = 0; x < M; x++) {
        Gluint index = y*N + x;
        vertices.push_back(glm::vec3(x, y,
                           heightMap[index].Height);
    }
}

Gluint T = (N-1) * (M-1) * 2;
Gluint I = T * 3;
std::vector<GLuint> indices;
indices.reserve(I);
for (int y = 0; y < N-1; y++) {
    for (int x = 0; x < M-1; x++) {
        // upper-left triangle
        Gluint topLeft = y*N + x;
        indices.push_back(topLeft);
        indices.push_back(topLeft + 1);
        indices.push_back(topLeft + M);

        // lower-right triangle
        indices.push_back(topLeft + 1);
        indices.push_back(topLeft + 1 + M);
        indices.push_back(topLeft + M);
    }
}
```

We now have the triangle indices, in correct order, for our mesh. The only thing remaining is to set the attribute pointers, and send them to the GPU:

```
Glint VAO, VBO, EBO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);

glBindVertexArray(VAO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices) * N * M,
             vertices.data(), GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) * I,
             indices.data(), GL_STATIC_DRAW);

// attribute pointer - vertex position
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3),
                      (GLvoid*)0);
glEnableVertexAttribArray(0);

glBindVertexArray(0);
```

And when we wish to render the surface mesh:

```
glBindVertexArray(VAO);
glDrawElements(GL_TRIANGLES, I, GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
```