CMSC 23700 **Introduction to Computer Graphics** **Project 4**
Winter 2014 **February 25**

Terrain rendering (part 1)

Due: Monday, March 10, 10pm

1 Summary

The final two projects involves rendering large-scale outdoor scenes. It will be split into two parts. For the first part (Project 4) you will implement basic terrain rendering using the so-called chunked-level-of-detail algorithm. For the second part (Project 5), you will be responsible enhancing your implementation various special effects of your choosing.

You will also be allowed to work in a group of two or three students for this project (note that CMSC 33700 students must work individually). There is a web interface for creating groups at

`https://work-groups.cs.uchicago.edu/pick_assignment/15334`

It works by one person logging in and then sending invites to the other group members. After login, the user will see a page where they can choose the assignment (there is only one), and then their partner(s). All invitations to group must be acknowledged on the site before the group's repository is created. Any issues should be reported to `techstaff@cs.uchicago.edu`.

2 Heightfields

Heightfields are a special case of mesh data structure, in which only one number, the height, is stored per vertex (heightfield data is often stored as a greyscale image). The other two coordinates are implicit in the grid position. If s_h is the horizontal scale, s_v the vertical scale, and $\mathbf{H}$ a height field, then the 3D coordinate of the vertex in row i and column j is $\langle s_h j, s_v \mathbf{H}_{i,j}, s_h i \rangle$ (assuming that the upper-left corner of the heightfield has X and Z coordinates of 0). By convention, the top of the heightfield is north; thus, assuming a right-handed coordinate system, the positive X-axis points east, the positive Y axis points up, and the positive Z-axis points south.

Because of their regular structure, heightfields are trivial to triangulate; for example, Figure 4 gives two possible triangulations of a 5×5 grid. Since naïve rendering of a heightfield mesh requires excessive resources (a relatively small 2049×2049 grid has over eight-million triangles), many algorithms have been developed to reduce the number of triangles that are rendered in any given frame. The goal is to pick a set of triangles that both minimizes rendering requirements and minimizes the screen-space error. Screen-space error is a metric that approximates the difference between the screen-space projection of the higher-detail geometry and the screen-space projection of the lower-detail geometry. For this project, you will implement the rendering part of a position-dependent algorithm called *Chunked LOD*.

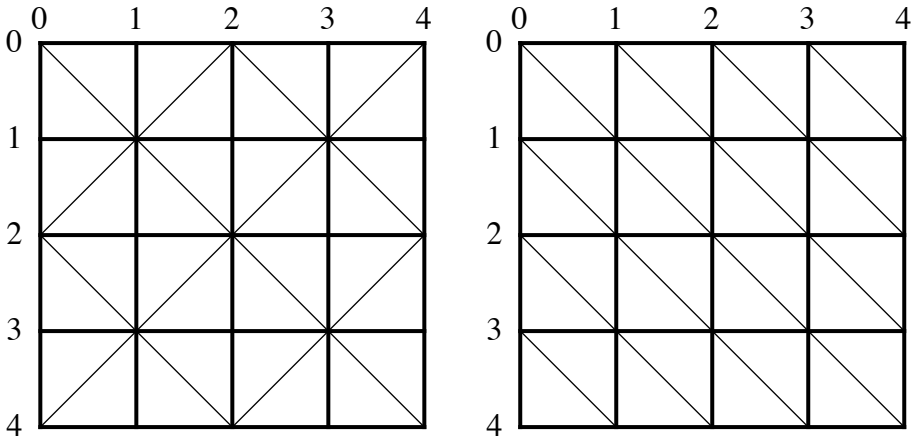


Figure 1: Possible heightfield triangulations

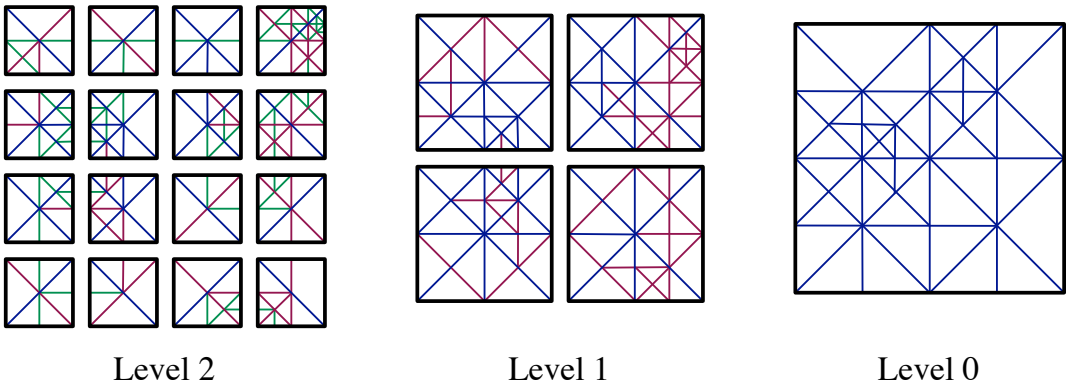


Figure 2: Chunking of a heightfield

3 Chunked LOD

The *Chunked Level-of-Detail* (or *Chunked LOD*) algorithm was developed by Thatcher Ulrich. This algorithm uses precomputed meshes (called *chunks*) at varying levels of detail. For example, Figure 2 shows a 33×33 heightfield with three levels of detail (note that level 0 is the *least detailed* level). The chunks of a heightmap form a complete quad tree Each chunk has an associated geometric error, which is used to determine the screen-space error. Figure 3 shows how the chunks at different levels of detail are combined to render the heightfield. Because chunks are different levels of detail are not guaranteed to match where they meet, the resulting rendering is likely to suffer from T-junctions (these are circled in red in Figure 3) that can cause visible cracks. To avoid rendering artifacts that might be caused because of T-junctions, each chunk has a *skirt*, which is a strip of triangles around the edge of the chunk extending down below the surface. The skirts are part of the precomputed triangle meshes.

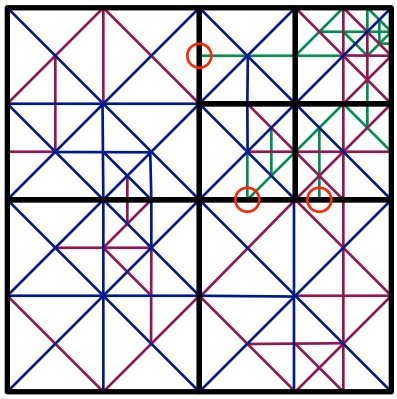


Figure 3: Rendering the heightfield from Figure 2

As mentioned above, which level of detail to render for a chunk is determined by the screen-space error of the chunk. For each chunk, we have precomputed the maximum geometric error (call it δ). Then, the screen-space error ρ can be conservatively approximated by the equation

$$\rho = \left(\frac{\textit{viewport-wid}}{2 \tan\left(\frac{\textit{fov}}{2}\right)} \right) \frac{\delta}{D}$$

where D is the distance from the viewer to the bounding box of the chunk and *fov* is the horizontal field of view. Better approximations can be had by taking into account the viewing angle (for example, if you are looking down on the terrain from above, then the projection of the vertical error will be smaller).

4 Map format

The maps for this project are not restricted to being square. A map covers a $w2^m \times h2^n$ unit rectangular area, which is represented by a $w2^m + 1 \times h2^n + 1$ array of height samples. This array is divided into a grid of square *cells*, each of which covers $2^k + 1 \times 2^k + 1$ height samples. For example, Figure 4 shows a map that is $3 \cdot 2^{11} \times 2^{12}$ units in area and is divided into 3×2 square cells, each of which is 2^{11} units wide. The cells of the grid are indexed by $(row, column)$ pairs, with the north-west cell having index $(0, 0)$.

A map is represented in the file system as a directory. In the directory is a JSON file `map.json` that documents various properties of the map. For example, here is the `map.json` file from a small example map:

```
{
    "name" : "Grand Canyon",
    "h-scale" : 60,
    "v-scale" : 10.004,
    "base-elev" : 284,
```

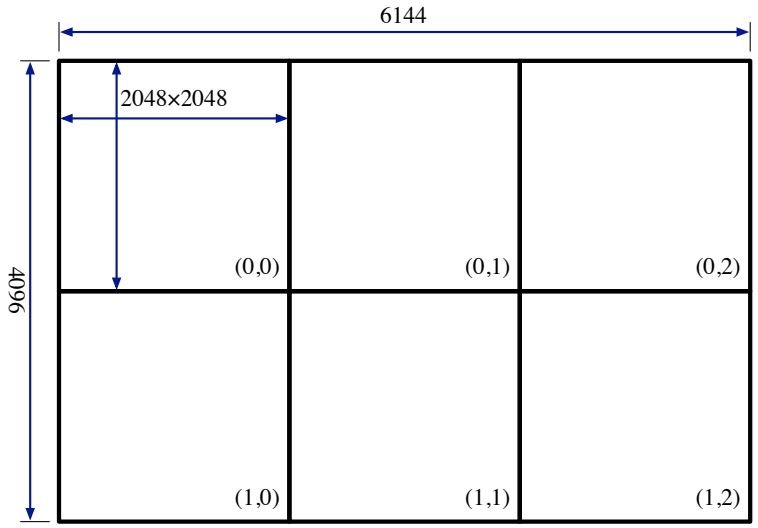


Figure 4: Map grid

```
    "min-elev" : 414.052,
    "max-elev" : 2154.75,
    "width" : 4096,
    "height" : 2048,
    "cell-size" : 2048,
    "color-map" : true,
    "normal-map" : true,
    "water-map" : false,
    "grid" : [ "00_00", "00_01" ]
}
```

The map information includes the horizontal and vertical scales (*i.e.*, meters per unit); the base, minimum, and maximum elevation in meters; the total width and height of the map in height-field samples;[1] the width of a cell; information about what additional data is available (color-map texture, normal-map texture, and water mask); and an array of the cell-cell names in row-major order. Each cell has its own subdirectory, which contains the data files for that cell. These include:

- `hf.png` — the cell's raw heightfield data represented as a 16-bit grayscale PNG file.
- `hf.cell` — the cell's heightfield organized as a level-of-detail chunks.
- `color.tqt` — a texture quadtree for the cell's color-map texture.
- `norm.tqt` — a texture quadtree for the cell's normal-map texture.
- `water.png` — a 8-bit black and white PNG file that marks which of the cell's vertices are water (black) and which are land (white).

[1]Note that the width and height are one less than the number of vertices; *i.e.*, in this example, the number of vertices are 4097×2049.

Of these files, only the first two are guaranteed to be present.

The sample code includes support for reading the `map.json` file, as well as the other data formats (`.tqt` and `.cell` files).

Because the map datasets are quite large (in the 100's of megabytes), we have arranged for them to be available on the CSIL Macs in the `/data` directory. You can also download the datasets from the course webpage to use on your own machine.

4.1 Map cell files

The `.cell` files provide the key geometric information about the terrain being rendered. Each file consists of a complete quadtree of level-of-detail chunks, where each chunk is contains a triangle mesh for that part of the cell's terrain, which is represented as a vertex array and an array of indices. The triangle meshes have been organized so that they can be rendered as *triangle strips* (`GL_TRIANGLE_STRIP`). The mesh is actually five separate meshes, one for the terrain and four skirts, and we use OpenGL's *primitive restart* mechanism to split them (the primitive restart value is `0xffff`). Vertices in the mesh are represented as

```
struct Vertex {
    int16_t     _x;           // x coordinate relative to Cell's NW
                              // corner (in hScale units)
    int16_t     _y;           // y coordinate relative to Cell's base
                              // elevation (in vScale units)
    int16_t     _z;           // z coordinate relative to Cell's NW
                              // corner (in hScale units)
    int16_t     _morphDelta;  // y morph target relative to _y (in
                              // vScale units)
};
```

Note that the vertices' x and z coordinates are relative the the *cell's* coordinate system, not the world coordinate system.[2]

The fourth component of the vertex is used to compute the vertex's *morph target*, which is the position of the vertex in the next-lower level of detail. Morphing is discussed in more detail in the next section.

The chunk data structure also contains the chunk's geometric error in the Y direction, which is used to compute screen-space error, and the chunk's minimum and maximum Y values, which can be used to determine the chunk's AABB.

5 Rendering

The first part of this project is to implement a basic renderer on top of the Chunked-LOD implementation. You solution should address the remainder of this section, as well as in Section 6. Your fragment shader should use the color and normal textures when they are available.

[2]For large terrains, using single-precision floating point values for world coordinates could cause loss of precision when the viewer is far from the world origin. Once can avoid these problems by splitting out the translation from model (*i.e.*, cell) coordinates to eye coordinates from the rest of the model-view-projection transform.

5.1 Basic rendering

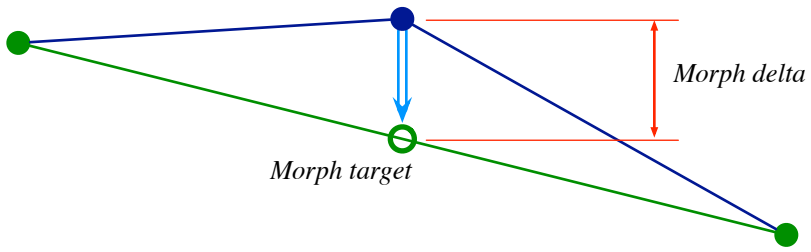
Your implementation should support basic rendering of the heightfield in both wireframe and shaded modes. You will need to compute the screen-space error of chunks and to choose which chunks to render based on screen-space error.

5.2 View frustum culling

Your implementation should support view-frustum culling. You can implement this feature by testing for intersection between the view frustum with the chunk's axis-aligned bounding box.

5.3 Morphing between levels of detail

To avoid popping when transitioning between levels of detail, your implementation should morph vertex positions over several frames (the morphing can be done in the vertex shader). Each vertex in a chunk is represented by four 16-bit integers. The first three represent the coordinates of the vertex (before scaling) and the fourth hold the difference between the vertex's Y position and its *morph target*. If the vertex is present in the next-lower level of detail, then the morph target is just its Y value, but if the vertex is omitted, then it is the Y-value of the projection of the vertex onto the edge in the next LOD as is shown in the following diagram:



5.4 Lighting

You should support a single directional light that represents the sun and diffuse shading of the heightfield. The map file format may optionally include a specification of the direction of the light.

5.5 Fog

Fog adds realism to outdoor scenes. It can also provide a way to reduce the amount of rendering by allowing the far plane to be set closer to the view. The map file format may optionally include a specification of the fog density and color.

6 User interface

We leave the details of your camera control unspecified. The main requirements is that it be possible using either the keyboard or mouse to change the direction and position of the viewpoint.

Furthermore, you should support the following keyboard commands:

`f`	toggle fog
`l`	toggle lighting
`w`	toggle wireframe mode
`+`	increase screen-space error tolerance (by $\sqrt{2}$)
`-`	decrease screen-space error tolerance (by $\sqrt{2}$)
`q`	quit the viewer

6.1 Submission

Project 4 is due on Monday, March 10. As part of your submission, you should include a text file named `PROJECT-5` that describes your plans for Project 5 (more information about Project 5 will be handed out soon).

History

2014-03-04 Added more details about the `.cell` file format and corrected how morph targets are represented.

2014-02-25 Original version.