CMSC 23700 **Introduction to Computer Graphics** **Project 5**
Autumn 2015 **November 15, 2015**

Terrain Rendering (Part 1)

Due: Tuesday November 24 at 10pm

1 Summary

The final project involves rendering large-scale outdoor scenes. It will be split into two parts (Projects 5 and 6). This assignment is the first part. The goal of this assignment is to get basic infrastructure for the final project up and running. There are a lot of pieces and you may not get all of them implemented, but do the best that you can. You will implement basic terrain rendering using the so-called chunked-level-of-detail algorithm. For the second part (Project 6), you will be responsible enhancing your implementation with various special effects of your choosing.

You may work in a group of two or three students for the final 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/26901`

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

You were introduced to the idea of heightfields in Project 3. Recall that they are a special case of mesh data structure, in which only one number, the height, is stored per vertex. 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.

As discussed in the Project 4 description, heightfields are trivial to triangulate, but the naïve rendering of a heightfield mesh requires excessive resources (a relatively small 2049×2049 grid has over eight-million triangles). To support using heightfields for large outdoor scenes, researchers have developed many algorithms 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

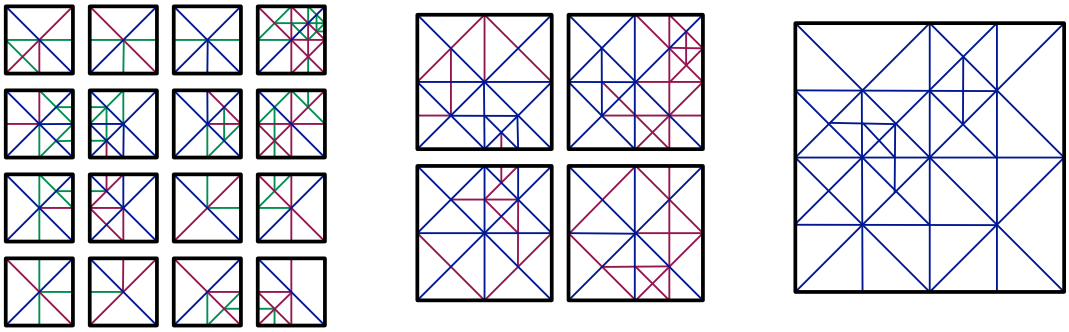


Figure 1: Chunking of a heightfield

the lower-detail geometry. For this project, you will implement the rendering part of a position-dependent algorithm called *Chunked LOD*.

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 1 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 2 shows how the chunks at different levels of detail are combined to render the heightfield. In this example, we have taken three chunks from Level 1 and four from Level 2. Because chunks at 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 2), which 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.

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 between it and the next higher level of detail. If δ is the precomputed error value, 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). When ρ exceeds some error tolerance, then we need to replace the chunk by the

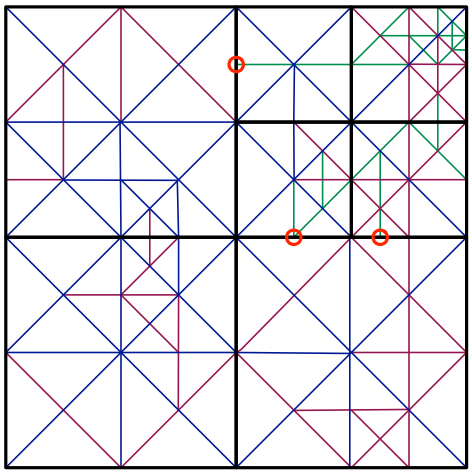


Figure 2: Rendering the heightfield from Figure 1

four chunks of the next higher level of detail.

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 3 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,
    "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

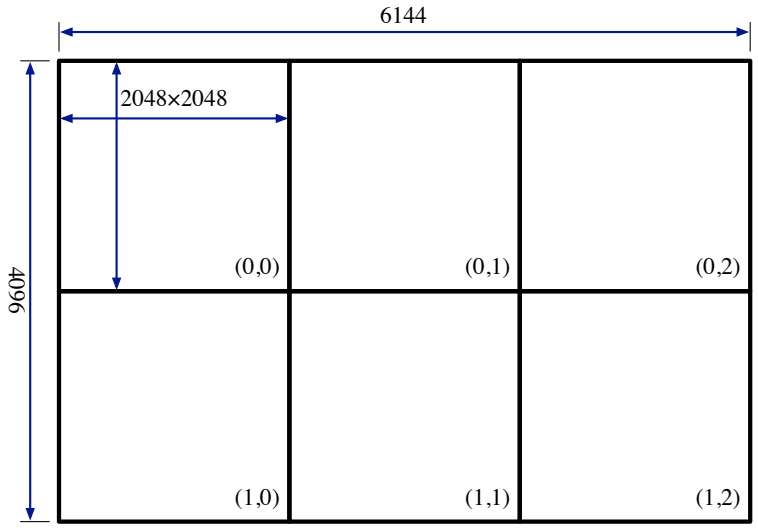


Figure 3: Map grid

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).

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, but please do **not** add them to your svn repository.

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

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 *tiles*. Each level of the quad tree defines a different level of detail and consists of $2^{lod} \times 2^{lod}$ tiles arranged in a grid (see Figure 1). The tiles are indexed by their level, row, and column. A tile covers a square region of the cell and contains various bits of information, including a *chunk*, which is the triangle mesh for that part of the cell's terrain at the tile's level of detail.

Chunks are 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.

For large terrains, using single-precision floating point values for world coordinates can cause loss of precision when the viewer is far from the world origin. One can avoid these problems by splitting out the translation from model (*i.e.*, cell) coordinates to camera-relative coordinates from the rest of the model-view-projection transform. The transformation of the `Vertex` structure to camera-relative coordinates in the vertex shader is straightforward. If we assume that $\langle x, y, z, m \rangle$ is the 4-element `Vertex`,[2] we compute

$$\mathbf{v} = \langle s_x x, s_y y + s_m m, s_z z \rangle + \mathbf{o}_{cell}$$

where $\mathbf{s} = \langle s_x, s_y, s_z, s_m \rangle$ is a uniform scaling vector and $\mathbf{o}_{cell}$ is the camera-relative origin of the cell. As described in Section 5.4 below, the morph delta (m) is being used to offset the altitude of the vertex (the adjusted altitude is called the *morph target*). Notice that $\mathbf{v}$ is in a space where the axes are aligned with the world-space axes, but the origin is at the camera.

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.

4.2 Texture Quad Trees

For each map cell, there are also two *texture quad trees* (TQTs) for the cell's color and normal maps. TQTs provide a parallel structure to the tile quad tree and use the same indexing scheme.

[2]The vertex fetch on the GPU will convert the 16-bit integers to floats.

Note, however, that some maps (*e.g.*, `gcanyon`) may have more levels of detail in the tile quad tree than in the TQTs. In this case, you will need to spread the texture over multiple chunks.

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 issues described in the remainder of this section, as well as the UI controls described in Section 6. Your fragment shader should use the color and normal textures if they are available in the scene.

The features are listed in order of importance. At a minimum, you should get the first two done for this project, but try to make as much progress as possible, since that will give you more time to work on Project 6.

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. We recommend that you structure the rendering of the terrain mesh as two passes over each cell's tile quadtree. The first pass should determine visibility information, level of detail, and manage the acquiring and releasing of resources (*i.e.*, VAOs and textures). The second pass can then render the visible chunks.

5.2 Lighting

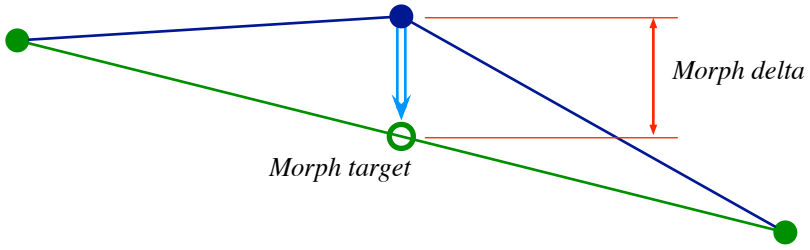
You should support a single directional light that represents the sun and diffuse shading of the heightfield. The map file format will include a specification of the direction of the light.

5.3 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.

5.4 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:



5.5 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 may optionally include a specification of the fog color C_{fog} and density ρ_{fog}. In your fragment shader, you should compute a distance-based *fog factor* as follows

$$f_{fog} = \frac{1}{e^{(\rho_{fog} d)^2}} \tag{1}$$

where d is the distance from the viewer to the fragment. One then uses the fog factor to compute the resulting color as follows:

$$C_{framebuffer} = (1 - f_{fog})C_{fog} + f_{fog}C_{fragment}$$

5.6 Details

Note that when the fog factor is close to one, the color will essentially be C_{fog}. Therefore, you should use the fog color as the background clear color so that the objects at a distance fade into the background. If you decide to add a skybox (in Project 6), you will want to add an altitude component to your fog computation so that the skybox blends into the fog at the horizon.

Calling the function `exp` for every fragment can be a performance hit. There are a couple of ways that you can make this faster, if performance is a problem. First, we can rewrite Equation 1 as

$$f_{fog} = 2^{k(\rho_{fog} d)^2} \tag{2}$$

where $k = \frac{-1}{\ln 2}$ (about -1.442695). This form of the fog equation allows us to take advantage of the fact that the GLSL function `exp2` is typically faster than `exp`. Second, we can compute the fog factor in the vertex shader and let the rasterizer interpolate it for us. The disadvantage of the second approach is that it may result in changes to the fog as the tessellation changes between levels of detail and it does not model work well when the terrain mesh has large triangles.

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 5 is due on Tuesday November 24. As part of your submission, you should include a text file named `PROJECT-5` that describes your plans for Project 6 (more information about Project 6 will be posted soon).

History

2015-11-24 Revised the discussion Section 4.1 and added some discussion about TQTs.

2015-11-22 Added discussion about transforming mesh vertices.

2015-11-21 Added more details about fog.

2015-11-17 Fixed due date.

2015-11-15 Original version.