CMSC 23700 **Introduction to Computer Graphics** **Project 5**
Winter 2008 **March 4**

Terrain rendering

Due: Tuesday, March 18 at 1pm

1 Description

This project is a continuation of Project 5. Your task will be to take an implementation of the ROAM mesh-simplification algorithm and build an interactive terrain engine. In addition to implementing the basic rendering engine, you will be responsible for implementing several special effects. The project is designed to be more open ended than previous projects; we will provide the heart of the rendering engine, but it will be up to you to make it look interesting.

As in the Project 4, your program will load in map data, including the heightfield. Since naïve rendering of a heightfield mesh requires excessive resources (a $1025x1025$ grid has 2^21 triangles) we will use a *continuous level-of-detail* algorithm to reduce the mesh to a reasonable size. Specifically, we will use the *split-only* version of the ROAM algorithm. This algorithm works by taking an initial course-grain triangulation of the heightfield (*e.g.*, two triangles) and then iteratively refining it until a predefined triangle budget is reached.

2 Input format

A terrain data set is represented as a directory containing various files that define the scene to be rendered. These files include

- `map` — this file contains information about the terrain data set, such as scale, feature locations, and the direction of the sun.
- `hf.pgm` – this file contains the height-field data.
- `color.ppm` – this file contains the vertex color information (see Section 3.1).
- `shadow.pgm` – the precomputed shadow texture (see Section 3.3).
- `water.pgm` – this file contains a depth map that gives the water depth for each grid location (see Section 6.3).
- `detail.ppm` – detail texture (see Section 3.3).
- `trees` – location of trees (see Section 6.1).
- `geysers` – location of geysers (see Section 6.2).

As in Project 4, the `LoadTerrain` function will be used to load the map data.

```
Map_t *LoadTerrain (const char *terrain, Viewer_t *view);
```

This function takes the name of the *directory* containing the terrain data set and returns a *map* object, which contains in-memory versions of the data. It also initializes the initial camera position and direction.

3 Rendering

The first part of this project is to implement a basic renderer on top of the ROAM CLOD implementation. You should probably use a fragment shader for this renderer, although you may use a fixed-pipeline solution too.

3.1 Lighting

For this part, we will add a single directional light (the sun) to the scene, which is specified in the map file. Adding lighting means that you will need to specify normals for your triangles as you render them. The colors assigned to a given vertex are defined by the `color.ppm` file.

3.2 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 has been extended to include a specification of the fog density and color.

3.3 Texture mapping

To make the surface of the terrain look more realistic, your implementation should blend in a *detail texture*. Each map has a `detail.ppm` file that contains the texture; you can use the function

```
RGB_t *LoadTexture (Map_t *map, const char *name, int *wid, int *ht);
```

to load this texture.

You use the detail texture, which is essentially a noise texture, to modulate the surface color close to the camera (say out to 80 to 100 meters). The texture is allocated as an RGB image, you may want to copy it to an RGBA representation with a 40-50% alpha channel, which will mute its effect somewhat. You can also use alpha blending to get a smooth transition from textured polygons to untextured ones.

In Project 4, you computed a shadow texture for the map; since this computation can be lengthy, we will provide a precomputed shadow texture as part of the map. This texture should also be blended into the rendered color.

4 ROAM

The ROAM algorithm is organized around a dynamic representation of triangle meshes called *triangle binary trees*. Figure 1 gives an example of a tree and Figure 2 shows the corresponding levels of triangulation. In the split-only version of this algorithm, we compute a new tesselation of the heightfield each frame by starting with the two triangles that cover the whole heightfield and then refining the mesh. We assign triangles a priority based on the benefit of refining them (*e.g.*, error metrics). Each triangle in the mesh has three neighbors (except for those triangles on the border) as is show in Figure 3.

As can be seen from these figures, constructing a binary triangle tree can be done as a recursive splitting procedure. The trick is that we only want to split a triangle if the resulting mesh provides a visibly more accurate approximation of the height field. Thus, we modify the recursive splitting procedure to split the triangle with the highest priority, where priorities are a measure of the visual effect of not splitting. We use a limit of the number of triangles in the mesh to control the amount of rendering work we do. Thus, the psuedocode for the tesselation phase is

```
initialize the mesh to top two triangles
while (size of mesh < limit) {
  split highest priority triangle
}
```

Splitting a triangle requires splitting the triangle's base neighbor (otherwise a T-junction results), but it may also presplitting the neighbor, when it is at a higher-level in the binary triangle tree. Figure 4 shows this situation.

4.1 Hints

You can adjust the priority of triangles to eliminate detail where it is not needed and to enhance detail where it is needed. For example, triangles that lie wholly outside the view frustum should have minimum priority, while the triangle containing the camera should have maximum priority. Since the view is mostly horizontal, you can approximate the view frustum as a triangle in the XZ plane. Given the vehicle's heading (a vector in the XZ plane) and the horizontal field of view, one computes the line equations for the sides of the frustum and then uses these equations to assign low priorities to triangles outside the view. Figure 5 illustrates this optimization for a small mesh.

Your program will need several distinct, but related data structures. You start with the heightfield that is the input data. You will need to compute a *variance tree* that contains the world-space variance information, a representation of the triangle mesh, and a priority queue for ordering splits. A strict priority queue is both not necessary and not efficient enough. Instead, use some number of priority buckets (think radix sort) to get constant-time insertions and deletions. It is also useful to have the bintree triangles down to the mesh level. The sample code includes C definitions for these structures.

5 Camera controls

Your implementation should support controls for the view as described in the following table:

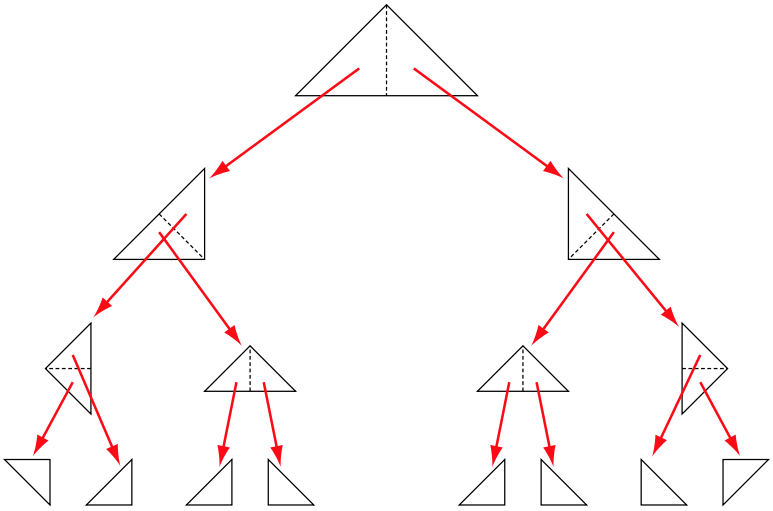

Figure 1: Binary triangle tree

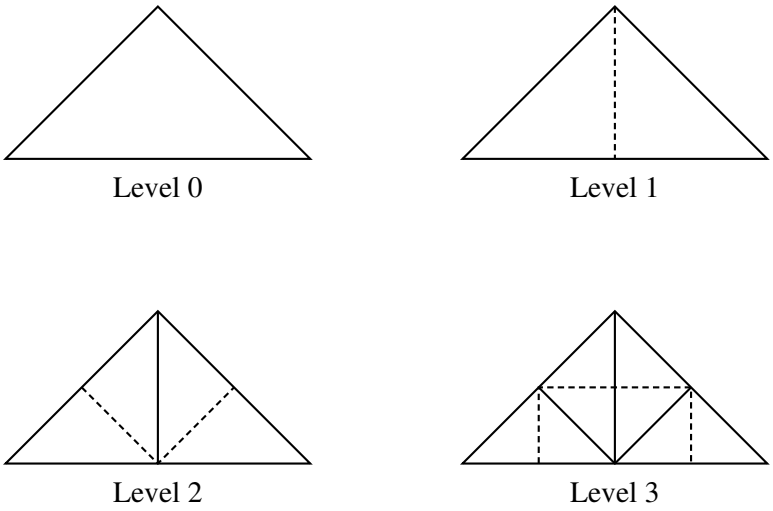

Figure 2: Binary triangle tree levels

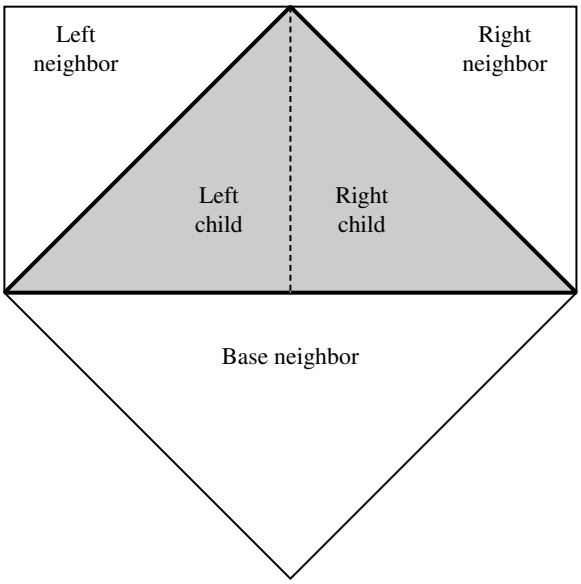

Figure 3: Triangle neighbors

UP ARROW	accelerate
DOWN ARROW	brake
LEFT ARROW	turn left
RIGHT ARROW	turn right
`f`	toggle fog
`l`	toggle lighting
`w`	toggle wireframe mode
`+`	increase level of detail (by $\sqrt{2}$)
`-`	decrease level of detail (by $\sqrt{2}$)
`q`	quit the viewer

For the navigation controls, you will need to sample the state of the arrow keys (instead of just reacting to keyboard events). GLUT provides a function for registering a callback that gets called when a special key is released:

```
void glutSpecialUpFunc (void (*func)(unsigned int key, int x, int y));
```

Thus you will need two callbacks to keep track of the state of the arrow keys. Since holding down a key generated a repeated sequence of key events, which take time to service, you can disable key repeats using the following GLUT call:

```
glutIgnoreKeyRepeat (1);
```

6 Visual effects

As part of your project, you should implement visual effects to make the terrain more interesting. We list a number of possible visual effects below, but you should feel free to be creative. Every group should implement at least a couple of effects.

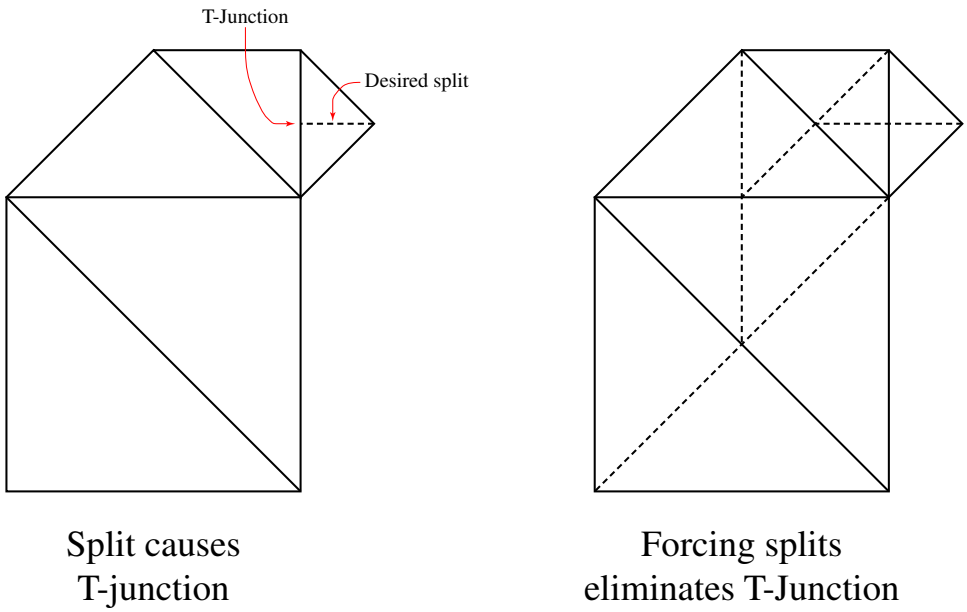

Figure 4: Forcing splits

6.1 Trees

A barren landscape in uninteresting, so to make it more attractive we can add some vegetation. A map can contain a `tree` file, which is a text file containing the locations of trees in the terrain. Use the function

```
Tree_t *LoadTrees (Map_t *map);
```

to load the array of tree information.

For purposes of this project, we use a simple representation of trees based on four concentric tapered cylinders (see Figure 6). We can use `gluCylinder` to render each of the tree components; to keep the polygon count down, you should use multiple levels of detail in your rendering. You may also want to consider various culling techniques.

6.2 Geysers

Driving around a static wasteland is boring, so we populate the terrain with geysers. Information about the position and size of geyser's is stored in the `geyser` file. Use the function

```
Geyser_t *LoadGeysers (Map_t *map);
```

to load the array of geyser information.

The behavior of a geyser is controlled by three parameters: the frequency f, the duration d, and the average height h. A geyser erupts on average once every $\frac{1}{f}$ seconds with an average duration of d seconds. The average height of an eruption is s.

We model geysers using particle systems. The initial velocity vector of the particles should have a cone-shaped distribution. You will want to set the average velocity to get the particles to the average height. You should also taper-off the system (both in particle velocity and number of

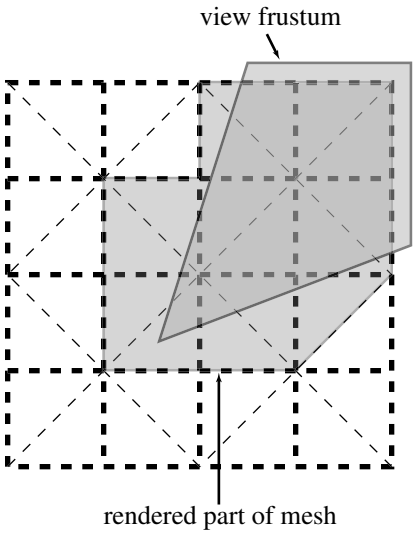

Figure 5: View-frustum culling

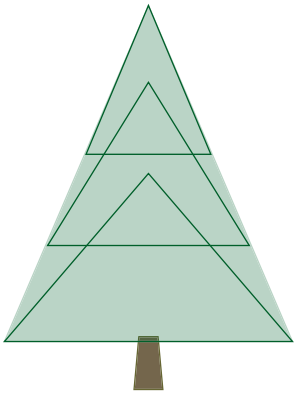

Figure 6: Tree constructed from tapered cylinders

particles) to get a gradual ending of the eruption. For example, let d be the duration of the geyser's eruption, then we can define the "size" of the geyser as a function of time t (assuming $t = 0$ is the start of the eruption):

$$s(t) = \begin{cases} he^{-(\frac{2t}{d})^2} & 0 \leq t \leq d \\ 0 & t > d \end{cases}$$

For example, $s(d) \approx 0.02h$. You can use this size function to influence the initial particle velocity, number of particles, and particle lifetimes. Feel free to play with your parameters and other decay functions.

6.3 Water animation

The surface of the lakes is flat and static. We can make it more interesting by adding waves to the water surface. The `water.pgm` file contains a depth map that gives the water depth for each grid

location (a value of zero means dry land). Use the function

```
Elev_t *LoadWaterMap (Map_t *map);
```

to load the water map. This map can be used to construct meshes that represent the lake surfaces, which can then be animated.

6.4 Sky box

Outdoor rendering engines use skyboxes (or skydomes) to provide a backdrop for the terrain. The basic idea is define a cube with the edges of the terrain mesh as sides and to render it with a texture of distant mountains, clouds, etc. Fancier skyboxes will animate the clouds.

6.5 Grass

In addition to using detail textures to make nearby terrain look more realistic, it is possible to add grass and small bushes to the scene. There are a number of shader-based techniques for rendering realistic looking grass.

7 Submission

We will set up a **gforge** repository for each group on the Computer Science server (see the *Lab notes* (Handout 2) for more information on **gforge**). We will collect the projects at 1pm on Tuesday, March 18th from the repositories, so make sure that you have committed your final version before then. You submission should include a `README` file in the top directory that describes what effects you have implemented and what methods you used. Please be careful to cite any papers, books, or online resources that you used.

You will be expected to demonstrate your render twice. During lab on March 12th, you should be ready to demo the basic rendering features of your project (*i.e.*, the features described in Section 3). We will also have a special demo session in the Mac Lab at 1:30 on March 18th (this is the regularly scheduled exam slot for the course).