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: