CMSC 23700 **Introduction to Computer Graphics** **Project 2(a)**
Fall 2005 **November 15**

Terrain rendering

Due: Wednesday, November 30

1 The problem

For this project your task is to implement a simulator for the next generation of off-road vehicles: the *sports utility hovercraft* (SUH). This program will read in terrain height and texture information and render the view from a simulated vehicle. The terrain height data is given as a square array of 16-bit height samples, which define a grid (or heightfield). This project is divided into two stages. In the first, you will implement a terrain renderer and vehicle simulator. The terrain renderer uses a level-of-detail optimization for the heightfield based on the ROAM algorithm. In the second stage, you will add texture mapping and two or more special effects.

In this document, we give an overview of the problem and describe the first part of the assignment.

2 Heightfields

Heightfields 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 Y 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. The heightfield is typically represented as a linear array of height samples, with the i, j element at index $iw + j$, where w is the width of the heightfield. Because of their regular structure, heightfields are trivial to triangulate; for example, Figure 1 gives two possible triangulations of a 5×5 grid. For this project, we will use the ROAM algorithm, which uses the triangulation shown on the left of Figure 1.

3 The vehicle

The user interface provides simple controls to navigate in the simulated SUH. This section describes the controls and the simulated physics of the vehicle.

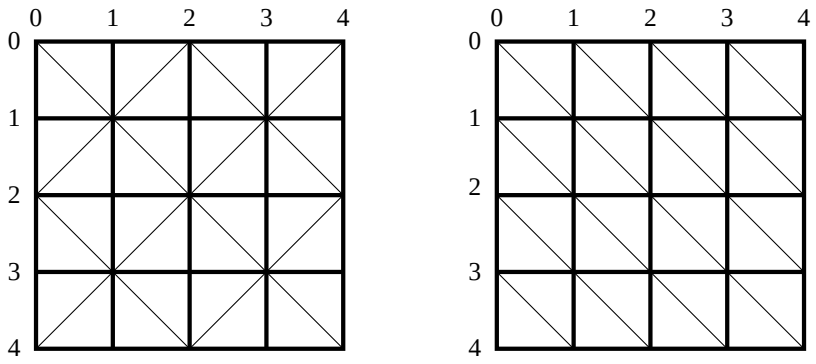

Figure 1: Heightfield triangulations

3.1 The controls

Navigation is controlled using the arrow keys. In addition, your implementation should support controls for the view as described in the following table:

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.[1] 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);
```

3.2 The physics model

We simulate the SUH with a very simple physics model based on discreet sampling. The state of the vehicle at a step i is given as a triple $(\mathbf{p}_i, v_i, h_i)$, where $\mathbf{p}_i$ is the vehicle's position in the $X-Z$ plane,[2] v_i is its velocity in $\frac{m}{s}$, and h_i is its heading in degrees (with north being 180 and south being

[1] Note that you should guarantee that any transient keystoke gets sampled at least once in the physics model.

[2] Note that the position $\mathbf{p}$ of the vehicle is given in X-Z coordinates; the altitude of the vehicle (the Y coordinate) will always be 2 meters above the terrain at the vehicle's position.

0). We recompute the state of the vehicle one hundred times per second (*i.e.*, every 10 milliseconds). Given the vehicle's state at step i, we can compute its new velocity at at step $i+1$ as follows:

$$\begin{aligned} a &= \max\left(0.0, 8.0 - \frac{v_i^2}{16.0}\right) \\ f &= f_0 + f_1 v_i + f_2(v_i^2) \\ g &= \mathbf{g} \cdot \mathbf{s}(\mathbf{p}_i, h_i) \\ v &= v_i - 0.01 \times (f + g) \\ v_{i+1} &= \begin{cases} \max(0, v + 0.01 \times a) & \text{if accelerating} \\ \max(0, v - 0.01 \times b) & \text{if breaking} \\ \max(0, v) & \text{otherwise} \end{cases} \end{aligned}$$

In these equations, a is the acceleration, which is speed limited, f is the force of friction, and g is the force of gravity. The new heading of the vehicle is determined by the following equation:

$$h_{i+1} = \begin{cases} h_i + \frac{t}{(v_{i+1}^2)+l} & \text{if turning left} \\ h_i - \frac{t}{(v_{i+1}^2)+l} & \text{if turning right} \\ h_i & \text{otherwise} \end{cases}$$

and the new position is given by

$$p_{i+1} = p_i + 0.01 \times v_{i+1} \langle \sin(h_{i+1}), \cos(h_{i+1}) \rangle$$

This computation depends on a number of factors, which are defined as follows:

b	$=$	0.6	braking factor
f_0	$=$	6×10^{-4}	friction coefficient
f_1	$=$	2×10^{-4}	friction coefficient
f_2	$=$	4×10^{-4}	friction coefficient
$\mathbf{g}$	$=$	$\langle 0, 9.8, 0 \rangle$	gravity
l	$=$	32	turn limit
t	$=$	24	turning factor
		$\mathbf{s}(\mathbf{p}, h)$	unit slope vector with direction h at position $\mathbf{p}$

If the vehicle is traveling at velocity v_i, we first compute a new velocity v that represents the effects of friction and gravity. We then apply acceleration and/or breaking to compute v_{i+1} (note that we do not let the velocity fall below zero). We use the new velocity in computing the new heading. Lastly, we compute the new position.

Computing the slope function $\mathbf{s}(\mathbf{p}, h)$ can be done in one of a couple ways.

- Project a 2D unit vector $\mathbf{d}$ in the direction h; *i.e.*, $\mathbf{d} = \langle \sin(h), \cos(h) \rangle$ and let $\mathbf{p}' = \mathbf{p} + \mathbf{d}$. Then let $H(\mathbf{p})$ be the height at position $\mathbf{p}$. The unnormalized slope vector is $\langle \mathbf{d}_x, H(\mathbf{p}') - H(\mathbf{p}), \mathbf{d}_z \rangle$. Divide this vector by its length to get $\mathbf{s}(\mathbf{p}, h)$.
- The other approach is to let $\mathbf{n}$ be the normal vector of the triangle containing $\mathbf{p}$ and let $\mathbf{d} = \langle \sin(h), y, \cos(h) \rangle$, for some unknown y. Then solve $\mathbf{n} \cdot \mathbf{d} = 0$ for y and set $\mathbf{s}(\mathbf{p}, h) = \frac{\mathbf{d}}{||\mathbf{d}||}$.

These methods will produce different results, but either is sufficient for the simulation.

Your simulator should take care that the vehicle does not go off the edge of the map. If that happens, you could teleport it back to the center of the map, or bounce it off the edge. It is also important to make sure that the vehicle stays above the surface of the heightfield.

The camera position is determined by the position and heading of the vehicle. One tricky problem is the vertical orientation. The simplest approach is to look parallel to the XZ plane, but this approach can leave you staring at steep slopes. Another approach is to pick a point some distance ahead of the vehicle to use as a "look-at" point. The disadvantage of this approach is that it can be quite jerky as your vehicle moves across the terrain. You can refine this approach by damping the change in view direction.

4 Rendering

The goal of a terrain rendering algorithm is to render a triangle mesh that models a given heightfield. The naïve approach of is to draw the complete triangle mesh, which requires rendering very large numbers of triangles (*e.g.*, a 1024×1024 heightfield has 2 million triangles). Even with view-frustum culling, the number of triangles is substantial. Since rendering such large numbers of triangles in real-time is impractical, we will use a *continuous level-of-detail* (CLOD) scheme to reduce the number of triangles rendered per frame. Many techniques have been developed for managing level-of-detail when rendering terrain; we will use the *split-only* variant of ROAM algorithm.

4.1 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.

4.2 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. We will use the `GL_EXP` fog mode for this project, but feel free to experiment with other settings Note that to make fog work effectively, you will need to render a *skybox* around the border of the heightfield.

5 ROAM

The ROAM algorithm is organized around a dynamic representation of triangle meshes called *triangle binary trees* [DWS+97]. Figure 2 gives an example of a tree and Figure 3 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 4.

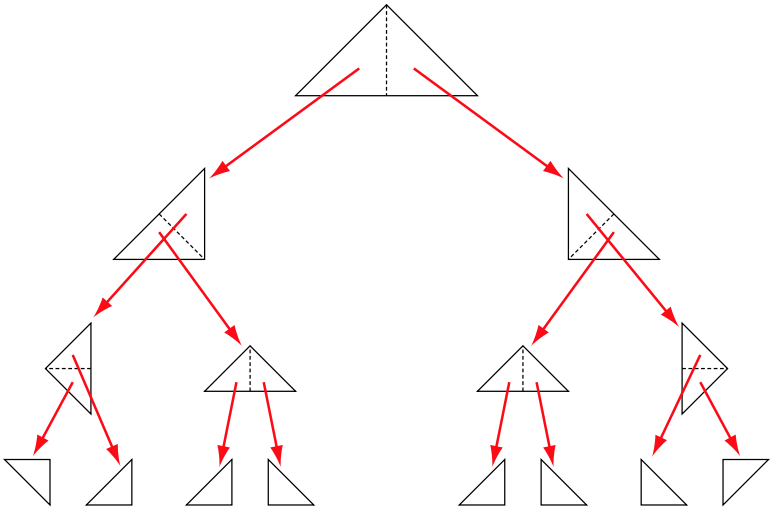

Figure 2: Binary triangle tree

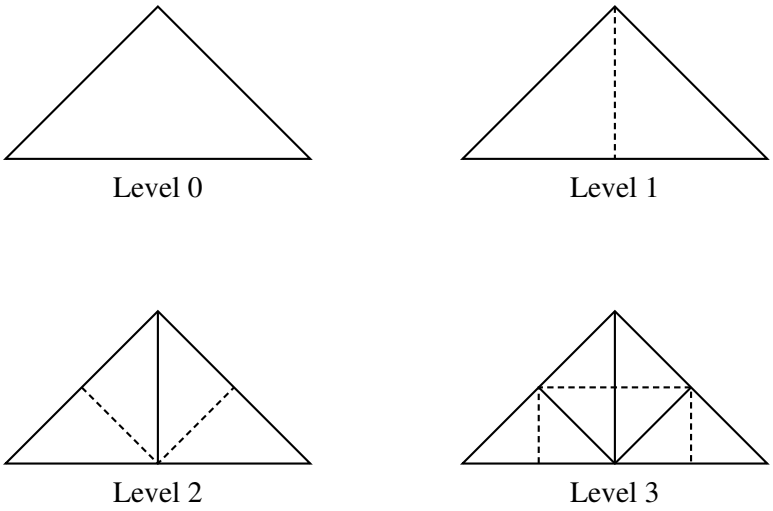

Figure 3: Binary triangle tree levels

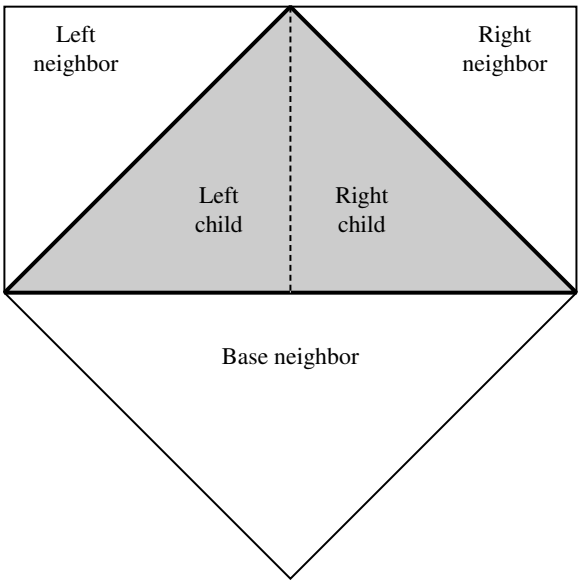

Figure 4: Triangle neighbors

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 5 shows this situation.

5.1 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 vehicle 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 6 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.

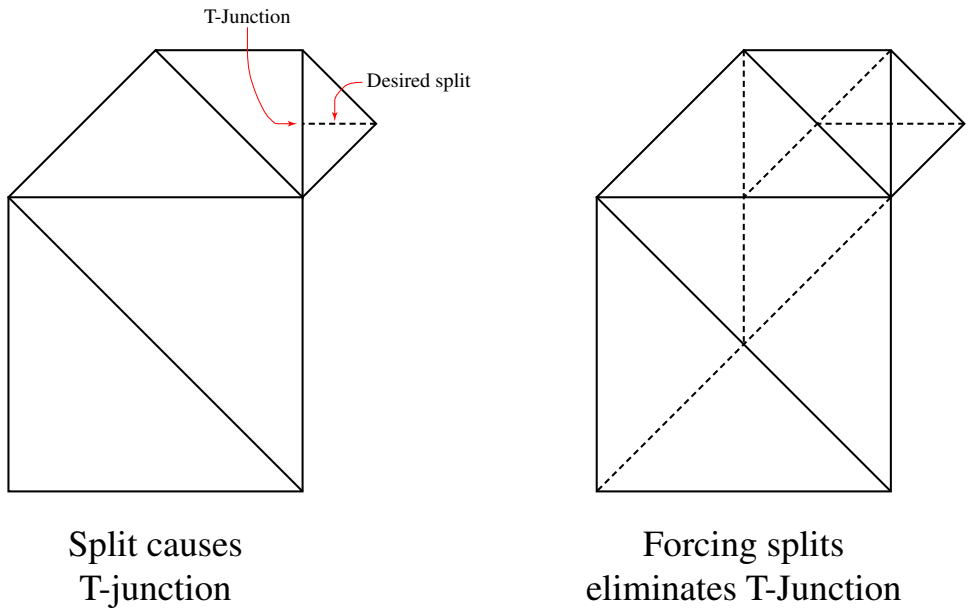

Figure 5: Forcing splits

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.

6 Input format

A terrain data set is represented as a directory containing the following files:

- `map` — this file contains information about the terrain data set, such as scale and feature locations, and about the initial position of the vehicle.
- `hf.pgm` – this file contains the height-field data.
- `color.ppm` — this file contains the vertex color information.

Later stages of the project will add more files to the data set. We will provide code for loading the input data.

6.1 Map file

The map file contains summary information about the terrain, plus the names and positions of various terrain objects.

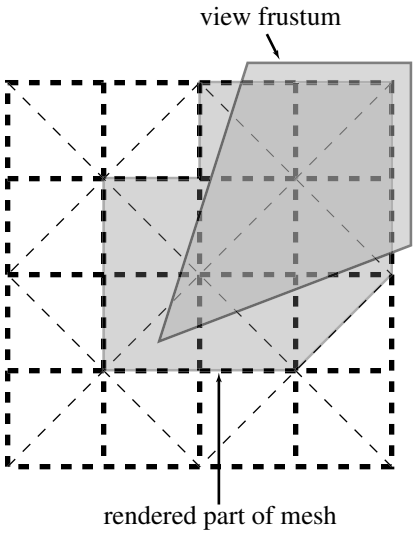

Figure 6: View-frustum culling

6.2 Height-field data

The heightfield data is stored as a *Portable Grey Map* (PGM) file with 16-bit samples. Its dimension will be $2^N + 1$ samples on a side (*i.e.*, $2^N \times 2^N$ grid cells). The horizontal scale (*i.e.*, distance between grid points) and vertical scale are given as part of the map file.

6.3 Color

The color of the terrain is specified as a separate pixmap image in *Portable PixMap* (PPM) format. There is one pixel per heightfield grid square (*e.g.*, if you have a 513×513 heightfield, then the corresponding color file will be 512×512). Use this color for the two triangles that represent the grid cell.

6.4 Data structures

We will provide code for importing the terrain description. The main entry-point to this API is the function

```
Map_t *LoadTerrain (const char *terrain, float vehiclePos[3]);
```

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, and three floats that represent the initial state of the vehicle (the XZ position and heading). The `Map_t` type, which is defined in `include/map.h`, is given in Figure 7. The `elev` and `color` arrays have `sideLen`^{2} elements and are stored in row-major order.

```
typedef unsigned short Elev_t;
typedef unsigned char RGB_t[3];

typedef struct {
    char        *path;      /* the pathname of the terrain */
    int         sideLen;    /* number of rows and columns in */
                            /* the data file */
    float       vScale;     /* vertical scale (default 0.1) */
    float       hScale;     /* horizontal scale (default 1.0) */
                          /* lighting information */
    Vec3_t      sunDir;     /* direction of sun light */
    RGB_t       sunColor;   /* color of sun */
    float       fogDensity; /* fog density parameter */
    RGB_t       fogColor;   /* fog color */
                          /* Height-field data */
    Elev_t      *elev;      /* elevation data */
    RGB_t       *color;     /* color data */
} Map_t;
```

Figure 7: The `Map_t` data structure

7 Requirements

Part-a of the project is due on Wednesday, November 30 at 9pm. You implementation should contain the following features:

1. User interface controls.
2. Vehicle simulator.
3. ROAM-based heightfield renderer.
4. Fog and lighting.

You should be prepared to demo your project in Lab on the 30th.

8 Document history

Nov. 15 Original version.

References

[DWS+97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.