CMSC 23700 **Introduction to Computer Graphics** **Project 3**
Fall 2003 **November 20**

Terrain rendering

Due: Thursday, December 11

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. The texture data defines one texel for each grid cell. The input data also includes additional terrain features that you may choose to render for extra credit (see Section 7.1).

1.1 The controls

Navigation is controlled using the arrow keys. In addition, your implementation should support a wireframe view and dynamically changing the level of detail.

UP ARROW	accelerate
DOWN ARROW	brake
LEFT ARROW	turn left
RIGHT ARROW	turn right
`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);
```

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

1.2 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, and h_i is its heading in degrees (with north being 180 and south being 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 state at step $i+1$ as follows:

$$
\begin{aligned}
v &= v_i - f_0 - f_1 v_i - f_2(v_i^2) + (\mathbf{g} \cdot \mathbf{s}(\mathbf{p}_i, h_i)) \\
v_{i+1} &= \begin{cases} \max(0, v+a) & \text{if accelerating} \\ \max(0, v-b) & \text{if breaking} \\ v & \text{otherwise} \end{cases} \\
h_{i+1} &= \begin{cases} h_i + \frac{t}{(v_{i+1}^2)+l} & \text{if turning left} \\ h_i - \frac{t}{(v_{i+1}^2)+l} & \text{if turning right} \\ h_i & \text{otherwise} \end{cases} \\
p_{i+1} &= p_i + v_{i+1}\langle \sin(h_{i+1}), \cos(h_{i+1})\rangle
\end{aligned}
$$

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

a	$=$	5×10^{-3}	acceleration factor
b	$=$	6×10^{-3}	braking factor
f_0	$=$	6×10^{-5}	friction coefficient
f_1	$=$	2×10^{-4}	friction coefficient
f_2	$=$	4×10^{-4}	friction coefficient
$\mathbf{g}$	$=$	$\langle 0, -0.1, 0\rangle$	gravity
l	$=$	0.3	turn limit
t	$=$	0.2	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.

[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.

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 that 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. The ROAM algorithm that we use in this project produces

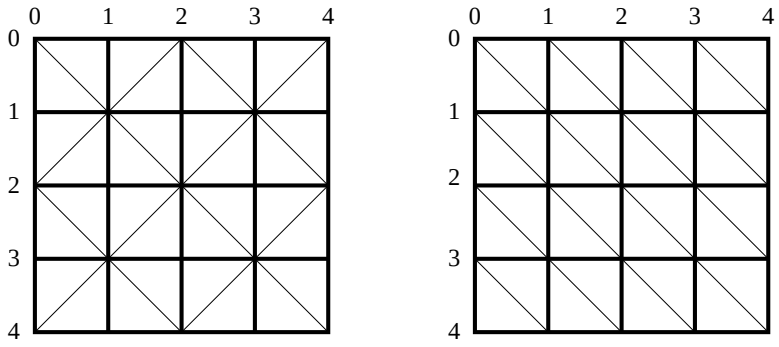

Figure 1: Heightfield triangulations

triangulations that follow the pattern on the left of Figure 1.

3 ROAM

The goal of a terrain rendering algorithm is to render a triangle mesh that models a given heightfield. The naive approach to draw the complete triangle mesh requires rendering very large numbers of triangles (*e.g.*, a 1024×1024 heightfield has 2 million triangles).

The basic idea of terrain rendering is that you are given a square *heightfield* of $(2^n+1)\times(2^n+1)$ points that defines a triangle mesh. In this project, the points are spaced one meter apart and the height values are 16-bit integers in units of 0.1 meters. Rendered as a triangle mesh, the heightfield requires $2(2^n)^2 = 2^{2n+1}$ triangles. Thus a one-kilometer square map has 2 million triangles. 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. For this project, we will use the *split-only* variant of the ROAM algorithm [DWS$^+$97].

The ROAM algorithm is organized around a dynamic representation of triangle meshes called *triangle binary trees*. 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

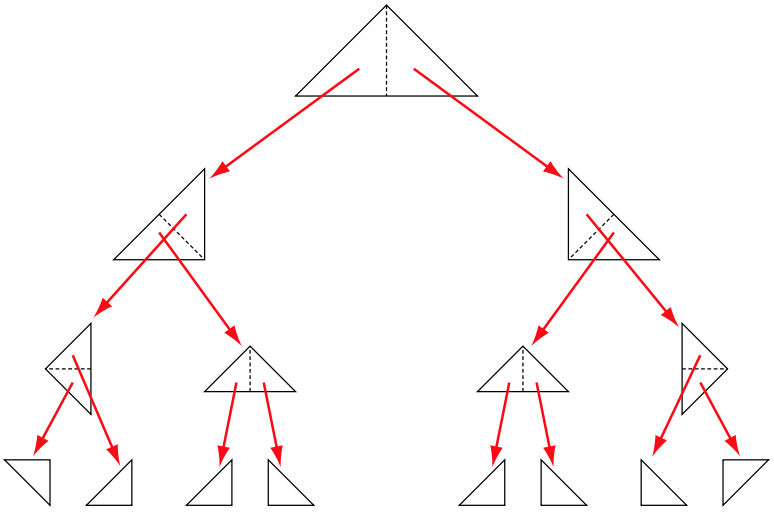

Figure 2: Binary triangle tree

then refining the mesh. Each triangle in the binary triangle tree has three neighbors (except for those triangles on the border) as is show in Figure 4.

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.

4 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.
- `hf.pgm` – this file contains the height-field data.
- `color.ppm`

We will provide code for loading the input data.

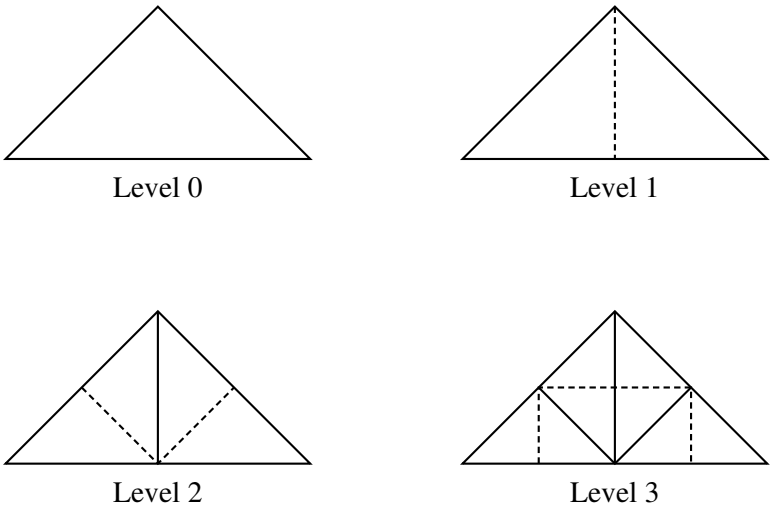

Figure 3: Binary triangle tree levels

4.1 Map file

The map file contains summary information about the terrain, plus the names and positions of the terrain objects (*i.e.*, geysers).

4.2 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.

4.3 Color

The color of the terrain is specified as a separate pixmap image, with one pixel per heightfield grid square (*e.g.*, if you have a 513×513 heightfield, then the corresponding color file will be 512×512).

4.4 Plant descriptions

The plant description files are given as L-Systems. The sample code includes the L-system loader and evaluator from Project 2, plus a simple scene-graph parser. You may use our parser or your own from Project 2. For this project you do not need to render plant shadows.

5 Hints

The key to this assignment is getting the data structures right. You will need a data structure to represent the trimesh; the triangles in this mesh are what your split operation will work on. You will also need a variance tree that maps bintree nodes to their variance. You should compute this tree at

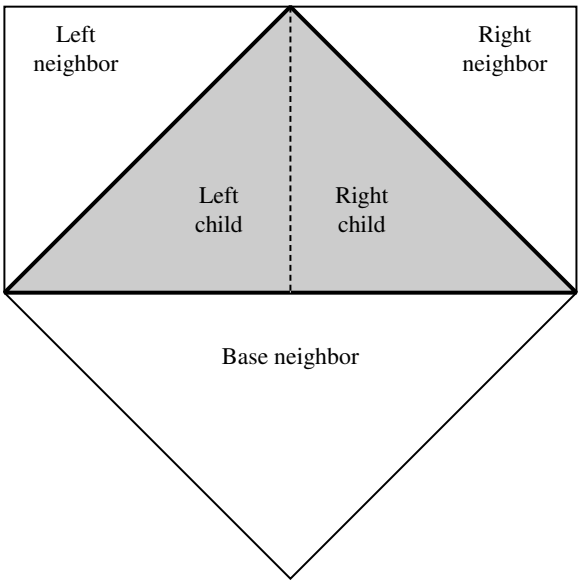

Figure 4: Triangle neighbors

heightfield load time. Since the variance of a leaf (*i.e.*, a triangle at the finest level of detail) is zero, your variance tree does not need nodes for the leaves. For a heightfield of dimension $2^N + 1$, there are 2^{2N+1} leaf triangles, which means $2^{2N+1} - 1$ nodes in the variance tree. Since the variance tree is a complete binary tree, you can use a linear representation of it.

6 Requirements

Your final version must be checked into your CVS repository by **Noon** on Thursday, December 11. You will be expected to demo your project in the MacLab on the 11th between 1:30 and 3:30pm.

The car's initial position should be in the center of the map facing east. The initial velocity should be zero and your initial triangle budget should be 10000.

7 Extra credit

If you have the time and ambition, there are a number of additional features that you may want to implement.

7.1 Geysers

Driving around an empty wasteland is boring, so we populate the terrain with geysers. Information about the position and size of geyser's is stored in the `map` file. Geysers are represented by particle systems, which will be discussed in another handout.

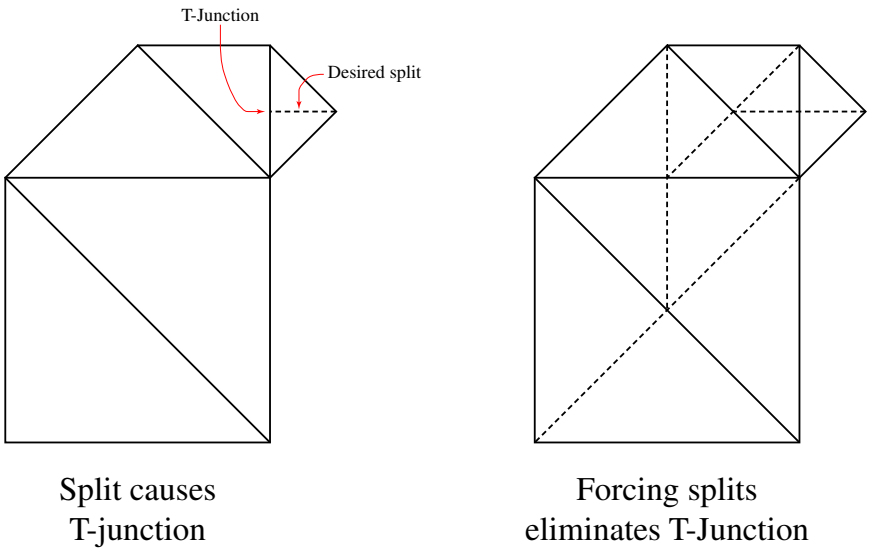

Figure 5: Forcing splits

7.2 Detail textures

Close up, the texture map we are using only has one texel per grid square. One approach to making the image more realistic is to blend in a *detail* texture on triangles that are close to the viewer. A detail texture can be something as simple as Perlin noise or it can be terrain features, such as rocks, grass, ...

7.3 Random terrain generation

There are many algorithms for generating random terrains. Implement one of them to produce the world.

7.4 Shadows

You can compute shadow volumes much the same way that you did in Project 2. Assuming that the position of the sun does not vary over the course of the simulation, you can precompute the shadow volumes at load time. Note that you should cull shadow volumes by the view frustum. You also need to mark those triangles that have silhouette edges with a high priority so that your shadow volumes are closed.

7.5 Full ROAM implementation

The full ROAM algorithm exploits frame-to-frame coherence by incrementally recomputing the tessellation from the previous frame. It does this by maintaining a pair of priority queues: one with triangles to split and one with diamonds (triangle pairs that share a common base) that can be merged.

8 Document history

Dec. 4 Sped up physics model and added initial conditions to Requirements section.

Dec. 1 Fixed equations for computing slope (no $i + 1$ subscript on h).

Nov. 20 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.