CMSC 23700 **Introduction to Computer Graphics** **Project 2(a)**
Fall 2004 **November 4**

Terrain rendering (Part a)

Due: Monday, November 15

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 three stages. In the first, you will implement a basic terrain renderer and vehicle simulator. In the second stage, you will add a level-of-detail optimization for the heightfield based on the ROAM algorithm. In the last stage, you will add texture mapping and one or more of a list of 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 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 Part-b of the project produces triangulations that follow the pattern on the left of Figure 1.

2.1 View frustum culling

One of the easiest ways to improve rendering performance is to cull objects that are outside the view. For this project, where the view is horizontal, it is particularly easy. 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 cull triangles outside the view. Figure 2 illustrates this optimization for a small mesh.

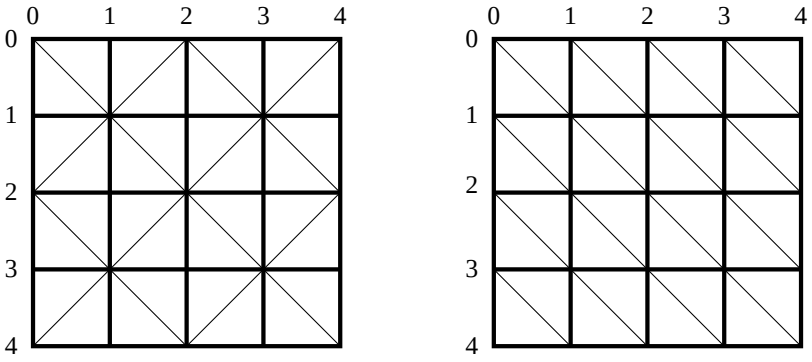

Figure 1: Heightfield triangulations

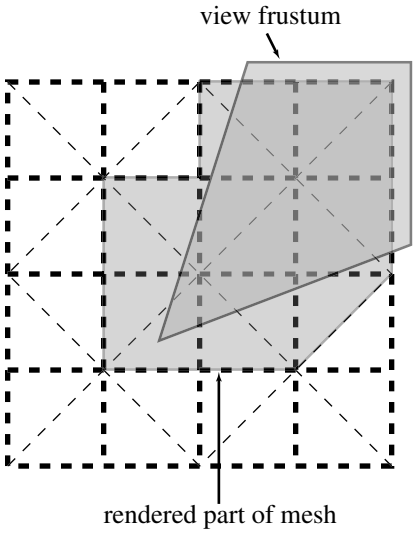

Figure 2: View-frustum culling

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.

3.1 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
`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, 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:

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

[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.

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. It is also important to make sure that the vehicle stays above the surface of the heightfield.

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` — 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.

4.1 Map file

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

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 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.

4.4 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);
```

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. The `Map_t` type is defined in `include/map.h` as follows:

```
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) */
                          /* Height-field data */
    Elev_t      *elev;      /* elevation data */
    RGB_t       *color;     /* color data */
 Map_t;
```

The `elev` and `color` arrays have `sideLen`^{2} elements and are stored in row-major order.

5 Requirements

Part-a of the project is due on Monday, November 15. For this part, you should implement the controls and vehicle simulator as described in Section 3 and the heightfield renderer with view frustum culling. Your final version must be checked into your CVS repository by 9pm on Monday, November 15. 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 and the initial velocity should be zero.

6 Document history

Nov. 15 Changed the equation for v_{i+1} to ensure that velocity does not go negative.

Nov. 4 Original version.