

CS 365 Homework 6
Topographic Display
(April 28 2004 rev.)

You will write a program that displays topographic data. It reads the data from standard input and displays the topographic surface.

You can reuse your HW3 code for the model-view and projection matrices. You can reuse the keyboard arrow key functionality to adjust the model-view and the projection matrices via the keyboard arrow keys, so you can zoom around to obtain different views of the figure.

This program is written in OpenGL/GLUT. Change the size of your window to 600 by 600.

Initial Display

When the program starts, it reads the height data from standard input and displays the same as a wire mesh. It also puts a solid sphere in the middle of the scene, for debugging purposes. The scene is lit from above by simulated sunlight. User controls change the eye point, the view point, the sun position, and toggle the topographic rendering between wire frame and false color.

User Controls

- **HOME key** on the keyboard. Resets the model-view matrix so that the eye point (the center of projection in the camera) is $(0, 2.0, 8)$ and the at point (where the camera is focused) is $(0, 0, 0)$.

In this program the projection matrix is fixed by the `glFrustum()` parameters: $x_{min} = -1$, $x_{max} = 1$, $y_{min} = -1$, $y_{max} = 1$, $z_{near} = 1.5$, and $z_{far} = 20$.

- **Left and right arrow keys.** The left and right keyboard arrow keys adjust the x coordinate of the camera's eye point, decreasing or increasing respectively. Each push of an arrow adjusts the x value by 0.5.
- **Up and down arrow keys.** These adjust the y position of the camera's eye point in the same manner as the left and right arrow keys.

- **Page up and page down keys.** These adjust the z position of the camera's eye point in the same manner as the arrow keys.
- **CTRL+Arrow keys.** Pressing ctrl together with one of the arrow or page up/down keys causes the at point to be adjusted, instead of the eye point.
- **Q key.** Quits the program.
- **C and c keys.** The C key converts the display mode to false color, with redder or whiter colors representing higher elevations and bluer colors representing lower. The c key converts the display mode to wire mesh again.
- **Digits 1 – 9.** Single digits 1 through 9 set the granularity of the display, where 1 means every square in the original data is displayed, 2 means every 2×2 square in the data is displayed as a single square on the screen, and so on up to 9×9 .
- **S and s keys.** Each push of the s key moves the angle of the sun by 22.5° to the “east,” positive x direction, until it is due east, where it does not move any further. The S key moves the sun “west” in the negative x direction until it is due west. Initially the sun comes from straight up (along the positive y axis) above the origin.

Display Data

Height data comes a file with one integer number per line. The first two numbers are the number of rows and columns, the following lines contain the height values for each coordinate, stored a column at a time. (In other words, you read the height data with the first subscript varying most rapidly). For example, the file may start like this: (comments not in original file):

```

768  number of rows
1024 number of columns
1705 height for grid coordinates (0,0)
1705 height for grid coordinates (1,0)
1701 height for grid coordinates (2,0)
...
459  height for grid coordinates (767,0)
1699 height for grid coordinates (0,1)
...etc.
```

The wire mesh display of this data is scaled so that the height is displayed along the y axis from 0.0 to 2.0. Note that the heights in the file are all non-negative integers,

but the lowest height may be well above 0. The (i, j) data coordinates are mapped into a rectangle in the (x, z) plane, centered at $(0, 0)$, with the largest dimension being 20.0 units long. If the i coordinate maps into the x axis, and the j coordinates maps into the z axis, then in the above example each grid square would display as $20/1024$ units on a side. The j values map into the z interval ± 10.0 while the i values map into x interval ± 7.5 .

There are three data files you can copy from `/home/mglass/cs365/` on the GEM Sun cluster: `honolulu.txt` maps part of Honolulu, `nmtopo.txt` maps part of New Mexico, and `objects.txt` displays some simple geometric objects. The geometric objects are quite useful for debugging.

Example C code for reading the height data into a 2-dimensional `int` array is distributed in class.

Sphere

Please put a unit sphere in the middle of the scene by invoking:

```
glutSolidSphere(1.0, 20, 16);
```

You can color the sphere a dull yellow, `rgb={0.5, 0.5, 0.1}`. This sphere will clearly show the reflected light and the 3-dimensional effect. It is a valuable debugging aid. In addition, it looks surreal in the middle of the landscape of New Mexico or Honolulu.

Lighting

The topographical scene is initially lit from above by white sunlight at an infinite distance. In order to accomplish this you can use `LIGHT0`, set up with the following OpenGL commands:

- Use `glLightfv()` to set the light position to $\{0.0, 1.0, 0.0, 0.0\}$ initially. This corresponds to an infinite distance (the fourth coordinate is 0) along the y axis (straight overhead). When the sun moves in response to the keyboard commands, you can set the x and y coordinates of the light position according to $\cos \theta$ and $\sin \theta$, where θ is the position of the sun relative to the eastern horizon.
- Use `glLightfv()` to set the diffuse light to white, `rgb={1.0, 1.0, 1.0, 1.0}`.

- Use `glEnable()` to enable `GL_LIGHTING` and `GL_LIGHT0`.
- In order to see the lit scene in 3D correctly, you will need to enable the z buffer: `GL_DEPTH_TEST` with `glEnable()` and `GLUT_DEPTH` with `glutInitDisplayMode()`.
- You will also need to set the the material colors and make sure you set surface normals at each vertex, both are discussed below.

This lighting scheme provides a kind of stark effect due to the lack of ambient light. Real terrain has quite a bit of ambient light from atmospheric diffraction. If you want to make the display more realistic, you can play with providing both ambient light and ambient reflectivity to the surfaces. If you want to make the display more surreal you can set the clear color to black, which gives the effect of viewing a terrain without atmosphere.

Displaying a Mesh Surface

I recommend you use `glBegin(GL_QUADS)` to display each square. If you display points counter clockwise, traversing the grid coordinates in the following order, the front face of the quad will be oriented up:

```
( i, j )
( i, j+1 )
( i+1, j+1 )
( i+1, j )
```

Even though the map looks like a wire frame when viewed from above, it is really composed of square white-colored filled-in polygons with black borders. This is so you cannot see through the mesh. I recommend that you use a true see-through wire mesh for the reverse (underneath) surface by using `glPolygonMode` as you did in HW 3. It is helpful for orienting the viewer while panning around the scene.

Reliably drawing black-bordered white-filled square polygons can be tricky. The problem is to ensure that the border is always drawn, and not sometimes clobbered in the z buffer by pixels from the polygon interiors. The method is to draw *both* a `GL_QUAD` for the interior and a `GL_LINE_LOOP` for the border. Furthermore you use a special “polygon offset” feature that moves the interior of the polygon slightly down from the edge. This both emphasises the border and ensures that the interior never clobbers the border. The code looks like this:

```

glPolygonMode(GL_FRONT, GL_FILL);
glPolygonMode(GL_BACK, GL_LINE);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(1.0, 0.5);

for (i=0; i<nrows; i+=p) {
    for (j=0; j<ncols; j+=p) {
        glBegin(GL_QUADS);
        ... draw a quad ...
        glEnd();
    }
}
for (i=0; i<nrows; i+=p) {
    for (j=0; j<ncols; j+=p) {
        glBegin(GL_LINE_LOOP)
        ...draw a line loop...
        glEnd()
    }
}
}

```

In the above code, p is the granularity of the mesh display. This parameter is set from the keyboard, it defaults to $p = 2$ when your program starts. When you step through the height matrix, each display square is p units on a side. Adjust the limits of the for loop so that $i + p$ and $j + p$ do not overflow the bounds of the height matrix.

Coloring

When the C key is pressed the topological display switches to false color mode. You pick an rgb color for each quad depending on its height from the base. The peaks should be reddish or whitish and the valleys bluish. Note that the false color scheme suggested on p. 360 of your textbook is quite beautiful, but it suffers from the problem that very small elevations are black. Much of the topographic surface simply disappears. So if you use the book's coloring you must fix it somewhat.

There are two ways to specify color. When lighting is enabled you need to specify the material properties of the surfaces using `glMaterial()`. The most important property of a surface is its color. Therefore you generally change the color of a surface by using `glMaterial()` to set the diffuse color to the desired rgb value, and you never use `glColor()` again.

However there is an alternate method that allows you to set the material properties by using the simpler (and more familiar) `glColor()` calls. You simply put the following in your initialization section, and use `glColor()` as usual. This will cause every call to `glColor()` to modify the material properties.

```
glEnable(GL_COLOR_MATERIAL);  
glColorMaterial(GL_FRONT_AND_BACK, GL_DIFFUSE);
```

When you enter false color mode please disable the grid lines (the `LINE_LOOP` borders).

Surface Normals

In order for the sun to properly illuminate the surface in 3-D, and cast shadows which move when the sun moves, you need to specify surface normal vectors at the corners of the mesh squares. Traditionally this is calculated by finding the surface normals of all four adjacent squares and averaging them. However it works quite well to compute only one normal, for one of the incident squares, and use it.

For each square that you display, label the corners of the square counterclockwise $\vec{p}_1, \vec{p}_2, \vec{p}_3, \vec{p}_4$. Compute the normal at \vec{p}_1 , which is simply $(\vec{p}_4 - \vec{p}_1) \times (\vec{p}_2 - \vec{p}_1)$. For many purposes you would want to convert this normal to unit length by computing:

$$\begin{aligned}\vec{n}' &= (\vec{p}_4 - \vec{p}_1) \times (\vec{p}_2 - \vec{p}_1) \\ \vec{n} &= \frac{\vec{n}'}{\|\vec{n}'\|}\end{aligned}$$

Although you compute a normal for one corner of the square, normals for the other three corners will be computed when you render their own squares.

This leaves some normals along the edges of the diagram unassigned. The edge strips and may stand out in the display, but that is not a problem for this assignment. If you want to clean it up, you can assign these edge vertices a straight-up normal: $(0, 1, 0)$.

The code for displaying a quad now looks like this:

```
for (i=0; i<nrows; i+=p) {  
    for (j=0; j<ncols; j+=p) {
```

```

... compute points p1=(i,j) scaled,
        p2=(i,j+p) scaled, p4=(i+p,j) scaled
... compute normal from p1, p4, and p2
... compute the color
... set the color with glColor or glMaterial
glBegin(GL_QUADS);
    ... set the normal with glNormal()
    ... draw the square with four calls to glVertex()
glEnd();
}
}

```

You must also enable `GL_NORMALIZE` and `GL_RESCALE_NORMAL`. If the surface does not display correctly, or the shadows come out reversed, it may mean your normals are pointing in the opposite direction. Compute the cross product in the other direction.

Acknowledgment

Thanks to Edward Angel of the University of New Mexico for the Honolulu and New Mexico height data along with some suggestions for this assignment.