CPSC 314, Project 2: Terrain Navigation

Out: Mon 7 Feb 2005
Due: Thu 24 Feb 2005 6pm PST
Value: 10% of final grade
Points: 100

In this assignment you will implement two different ways of flying around above bumpy terrain.

Terrain [40 pts]

You need to make a bumpy ground plane, where the color at each vertex is random and the height of each vertex in the plane varies randomly. Specifically, the amount of variance in the height should be +/- 20% of the width of the face. (If your faces are not equal in length and width, use whichever is bigger.) You should support switching between using per-face normals and per-vertex normals, using the 'v' key to switch between these modes. Your plane should lie in the xz plane, with the bumps in the y direction. The plane should be a grid of 100x100 faces, for a total of 10,000 faces in the terrain, with the corners of the terrain at (0,0,0), (100,0,0), (100,0,-100), and (0,0,-100).
You should support actually drawing the normals that you're using for the lighting calculations, as short lines perpendicular to the center of the face in per-face mode, or coming from each vertex in per-vertex mode. Use the 'n' key to switch normal drawing on and off.
Update: Use the 'u' key to replace your terrain with a new set of randomly generated geometry.
Update 2: Use the 't' key to toggle between grey and the random colors.
Extra credit: [1 pt] Control the granularity of the plane, subdividing it into more or fewer faces (i.e. change the number 100 into something smaller or larger, while keeping the size of the plane the same).

Terrain Hints:

- The easiest way to make a bumpy plane is to compute vertex locations on a flat plane in the x and z directions, then perturb the height of your vertices in the y direction.
- A rectangular region where each vertex has a different height will not be planar. For each rectangular region you should send two triangular faces to the OpenGL pipeline, and these two faces will be planar with a crease in between them on the diagonal of the rectangle. The easiest way to send geometry to OpenGL is with the GL_TRIANGLE mode, even though that requires you to resend information about an individual vertex multiple times. If you're feeling ambitious, you could gain efficiency by using the GL_TRIANGLE_STRIP mode.
- Since you will need to interpolate per-face normals to create per-vertex normals, you will need an easy way to find all the faces around a particular vertex. I recommend storing your vertices in a 2D array. Then you can traverse this data structure to do your vertex position calculations, then your per-face normal calculations, then your per-vertex normal calculations, and finally to send the geometric data to OpenGL.
- To compute the normal vector for a face: construct two vectors from the points that determine the face, take their cross product, and normalize that vector to be unit length. Remember to traverse the vertices for a face counter-clockwise, according to the OpenGL convention, both for computing these normals and when sending the geometric data to OpenGL.
- Consider what kind of a data structure you need to store the normals for each face.
- First compute per-face normals, then use those to compute your per-vertex normals. To compute the normal vector for each vertex: interpolate between the normal vectors for all the faces that share a vertex.
- When using per-face normals, send just one normal for each face, and OpenGL will use it for all the vertices. For per-vertex normals, send a new normal before each vertex.
- Note that normal drawing can help you debug your normal calculations, so you will want to implement that sooner rather than later.

Lighting and Shading [15 pts]

Lighting [10 pts]: You should have at least one light that is in the world coordinate system (you can add others if you want), and one headlamp in the camera coordinate system that moves along with your point of view. Your headlamp should be able to be turned on and off using 'h' as a keyboard toggle switch.

Shading [5 pts]: You should support switching between a flat shading model, where the color calculation is done once per face, and the smooth shading model, where colors are calculated at each vertex, using the `glShadeModel` command. Use the 'f' key as a toggle. **Update: changed from 's' to 'f'.** Notice the interplay between the shading model and the normal calculations.

Navigation [45 pts]

You will implement two different ways of flying around your scene: absolute camera movement, and relative camera movement. You'll probably find that relative camera movement is a lot easier to use!
Start with a camera that has an eye point of (50, 10, 10), looking at the point (50,0,-75), and the y axis is up. Hitting 'r' should reset the camera to this default view. **Update 3: Hitting 'm' should switch between the two modes of movement and also reset to the default view.**

Absolute Camera Movement [10 pts]: You should implement absolute camera movement where you control the x/y/z coordinates of the eye point, lookat point, and up vector with keyboard keys. You'll increment a value with a lowercase key, and decrement it with a uppercase key as follows: eye x: 'q'/'Q', eye y: 'w'/'W', eye z: 'e'/'E', lookat x: 'a'/'A', lookat y: 's'/'S', lookat z: 'd'/'D', up x: 'z'/'Z', up y: 'x'/'X', up z: 'c'/'C'. (Those are the keys on the top, middle, and bottom rows of the keyboard on the left side, so that you can hit them all without moving your hand.) You can play with the amount to increment/decrement according to the speed of the computer you're using, try starting with 0.5 for the value.

Relative Camera Movement [35 points]: You should implement camera movement where you fly around the scene by incrementally controlling forward/backward speed, roll, pitch, and yaw with mouse drags. All of this motion should be calculated with respect to the current camera coordinate system. Pitch is rotation around the horizontal axis, like nodding your head up and down for yes. Yaw is rotation around the vertical axis, like shaking your head from side to side for no or steering a car. Roll is rotation around the front-to-back axis, like tilting your head so your ear touches your shoulder (like the up vector in the lookat model). There's a good animated illustration at http://www.nasm.si.edu/galleries/gal109/NEWHTF/PITCH.HTM
When a mouse button is held down you move, and you stop moving when the button is lifted up. The vertical component of the vector created by a left mouse drag controls your forward or backward speed (up is forward and down is backward), and the horizontal component controls your yaw angle (left is yaw to the left, and right is a yaw to the right). **Update: changed left horizontal drag to yaw not roll.** You should keep track of the position at the start of a mouse drag, and consider the size of the vector from that start position to the current mouse position. Dragging further away from the starting point makes you go faster or turn more, and dragging it back towards that point slows you down and makes you turn less. Releasing the mouse button stops the motion. The right mouse similarly controls roll with horizontal drags and pitch with vertical drags. . **Update: changed right horizontal drag to roll not yaw.** You'll need to experiment with how to set the weights when converting drag vectors into motions, in order to make flying feel natural instead of jerky.

Extra credit [up to 9 pts]: Implement a third way to navigate. You'll get a few points for simple changes to the required navigation, and more points for doing something more elaborate or drastically different.
Update: Extra credit [.5 pts]: Use the '+/-' keys to increment and decrement your speed in absolute camera coordinate keyboard motion.

Camera Hints:

- Remember that you'll have to flip the y coordinate, since the window system will be sending you coordinates that start at the upper left instead of the lower left.

- The relative motion specification requires incremental changes of roll/pitch/yaw angles and forward/backward motion with respect to the current camera coordinate system. You could imagine keeping track of cumulative roll/pitch/yaw values with respect to some set of basis vectors kept in world coordinates, but that would require a lot of calculation. And transforming roll/pitch/yaw angles into the eye/lookat/up vector format required by gluLookat would be even more work.

  In contrast, if you assume that you know the current camera coordinate system (let's call it Current), it's easy to calculate the simple new incremental motion, where a drag means a simple motion with respect to the current x, y, or z axis of Current. This new incremental transformation (let's call it Incremental) needs to be applied with respect to the current transformation; that is, p' = Incremental * Current * p. The good news is that Current is exactly the modelview matrix used by OpenGL to draw the previous frame. If you don't wipe that out with glIdentity, that matrix is still intact and contains the information you need. However, OpenGL only allows you to postmultiply a matrix, which would result in the incorrect operation p' = Current * Incremental * p. The trick is to first explicitly store the Current matrix, which you can do with the `glGetDoublev` command that dumps out the contents of the top of the OpenGL matrix stack. Then you can get the desired effect by wiping the stack with glIdentity, first applying the incremental transformation, and finally multiply by the stored Current. This trick saves you a lot of work by using the OpenGL matrix stack as both a calculator and storage device!

Handin

We will again be grading through face-to-face demos. Do the same thing as with project 1, except for three changes. First, use the command 'handin cs314 proj2'. Second, you only need bring a printout of the README file to your demo slot (that is, no need to bring code printouts). Third, there is no need to submit image files since there will not be a Hall of Fame for this project. (Stay tuned for the ultimate Hall of Fame competition with Project 3!)