Computer Science 591B, Graphics

http://www.cs.umass.edu/∼elm/graphics/

Assignment 3: Primitive 3-D Rendering

You will be using the SDL environment again for this assignment. You will define the geometry and coloring of a 3-D object and render it using orthographic projection in 3-D. You will then be asked to make the object "tumble" by continuously rotating it in a semi-random fashion.

Following the steps laid out here is designed to make the assignment easier. I recommend that you try to do the assignment in the order presented here, confirming along the way that each step works as expected before going to the next part.

1. Define a structure called `vector3` which will hold the points that define your object. It should have 3 components each of which is a `double`. You may find it easier later on if you define the components as a fixed size array of size 3. For the purposes of this assignment, I will refer to the first coordinate of this triple as the "x" coordinates, the second as the "y" coordinate, and the third as the "z" coordinate. Define a default constructor for `vector3` which does nothing, and another constructor that takes 3 doubles as initial values. Write a member function `mag()` which returns the Euclidean magnitude (length) of the vector. Write two more functions THAT ARE NOT MEMBER FUNCTIONS OF `vector3`. The first function should subtract two vectors and be defined as

   ```
   vector3 minus(vector3 v1, vector3 v2);
   ```

   The second function should evaluate the cross product of two 3-D vectors and should be defined as

   ```
   vector3 cross(vector3 v1,vector3 v2);
   ```

2. The next task is to define the fundamental rendering primitive, the triangle. Each of the vertices of the triangle should be a point in 3 dimensions. Define a C++ structure called `triangle`. It should consist of 3 `vector3` objects and an unsigned integer called `color` which will represent the surface color of the triangle. Define a default constructor which does nothing, and another constructor which takes 3 `vector3` objects and an unsigned int for the color to define the triangle in 3-D. Define a member function

   ```
   double triangle::signedAreaXY();
   ```

   This function should NOT compute the signed area of the 3-D triangle. What it SHOULD compute is the signed area of the triangle defined by the `x` and `y` coordinates of each point. In other words, it should compute the signed area of the triangle which results from projecting this triangle onto the $x-y$ plane. This can be found simply by using the formula for computing the signed area of a triangle in two dimensions that we discussed in class. Just ignore the z-coordinate and you should get the right result. Furthermore, note that a signed area can be negative.

   Test your area function with some different 3-D triangles. If the x-coordinate of all 3 vertices of the triangle are the same, the area should be 0. The same is true for the y coordinates. Note that if you reverse the ordering of the vertices of a defined triangle, your signed area should be negated. Make sure all of these properties are correct before you move on, as they will be very hard to debug later.

3. You will now use the area of the two-dimensional projection of a triangle to find the two-dimensional barycentric coordinates of the projected triangle. Define a member function

   ```
   triangle::baryXY(vector3 v,double &alpha,double &beta,double &gamma);
   ```

   which computes the two-dimensional barycentric coordinates of a point in the given triangle and returns them in the variables `alpha`, `beta`, and `gamma`. Just as you ignored the *z*-coordinate of the triangle vertices in computing the signed area of the triangle, you should ignore the *z*-coordinate of the argument `v` in computing the barycentric coordinates. Thus, you are actually computing the barycentric coordinates of *the projection* of `v` in the projection of the original 3-D triangle. To make sure this routine works, try out various points within a triangle and make sure the barycentric coordinates are correct. Again, it will pay off if you make sure this routine works before continuing. It will be very hard to debug later. Try putting in points on the edge of the triangle for the argument `v`. What should the barycentric coordinates be? What about a point at a vertex of the triangle? What about a point in the middle of an equilateral triangle? Finally, what about a point outside the triangle? If you don't remember the answer to these questions, read about barycentric coordinates in the book or look at your notes from class. (*What notes?*)

4. The next member function you will write will be

   ```
   triangle::rasterize();
   ```

   As its name suggests, this will take in a 3-D triangle, and, ignoring the *z*-coordinate, will paint each pixel of the triangle according to the color of the triangle. You should rasterize the triangle as we discussed in class. Rather than cycling over the ENTIRE frame buffer for each triangle, you should find the bounding box for the triangle (determined by the min and max of the *x* and *y* coordinates) and loop over the bounding box. For each screen coordinate in the bounding box, see whether the barycentric coordinates are non-negative, and if so, paint the point into the frame buffer. Try rasterizing a few simple triangles before moving on, including triangles of 0 width or 0 size.

5. Next, you should add a z-buffer to your triangle rasterization routine. Probably the easiest way to do this is to use a global variable for the z-buffer. Each time you are going to plot a point in the frame buffer (when its barycentric coordinates are all non-negative), you should store the *z* coordinate of the current point in the z-buffer if it is closer to the viewer than the last point. Alternatively, if the point in the z-buffer is already closer to the viewer than the point you are rendering, you should not render the point. You will need to compute the *z* coordinate of the current point by interpolating it from the *z* coordinates of the vertices of the current triangle. This can be done using interpolation using the barycentric coordinates, as we showed with color interpolation. Plot some triangles using the z-buffer code and make sure that they come out in the proper visual order. I don't care whether the viewer is looking in the *z* positive or *z* negative direction.

6. Define a structure called `mat3h` which is a homogeneous transformation matrix for 3-D coordinates. It should consist of a 4 by 4 array of doubles. Define a default constructor which does nothing. Define a member function `void mat3d::zero()` which sets all elements of the array to 0. Define another member function `void identity()` that sets the elements of the matrix so that it is an identity matrix.

   Write functions (NOT MEMBER FUNCTIONS) that multiply two mat3h's:

   ```
   mat3h mat3hMult(mat3h a,mat3h b);
   ```

   and that multiply a matrix (`mat3h`) times a vector (`vector3`):

   ```
   vector3 matMult(mat3h a,vector3 v);
   ```

   Define additional functions which generate rotation matrices for each directions of rotation:

   ```
   mat3h rot3hX(double theta);
   mat3h rot3hY(double theta);
   mat3h rot3hZ(double theta);
   ```

   given the angle of rotation theta about the particular axis. A good check for your rotation matrices and matrix vector multiplication is that each rotation matrix should leave one coordinate of a vector unchanged.

7. Now it's time to define a 3-D object. I suggest a cube, since the geometry is very simple, but you may use other objects if you like. The first step is to define the 3-D vertices of the shape. After that, you should define triangles that use these vertices to define your 3-D shape. The order of the vertices in each triangle definition should not matter, and you shouldn't need to worry about it.

8. Finally, put it all together. Define three variables, `xrot`, `yrot`, and `zrot` for storing the current "rotation" of your shape. At each rendering step, you should apply rotation matrices created (by using `xrot`, `yrot`, and `zrot` as the angles) to your shape to change its orientation in 3-D. You may need to translate your object to the origin before rotating it, and translate it back after rotating it. After rotating your object, render it by rasterizing all of the triangles within it using the z-buffer rasterization. Finally, change the rotation variables a little bit after each rendering. This will cause your object to "tumble" during the animation.

Turn in your code as usual to Alex.