CS1566 Assignment 2: Stitcher

Out: Thu 09/13. Due: Tue 09/25 11:59pm

1. Introduction

Computer graphics often deals with images of three-dimensional scenes. The catch, however, is that our screens can only display two-dimensional images. Therefore, there needs to be some way to convert a three-dimensional scene to something that can be viewed in two dimensions. A common method, which we will use in this and later assignments, is by composing a scene using only triangles, and then projecting those triangles to the screen, drawing each one sequentially. Triangles behind other triangles simply aren't drawn. (Triangles are the simplest geometric surface unit, so all other surfaces can be reduced to triangles. Another commonly used surface unit is the quad, a 4-vertex flat polygon.) In this assignment, you will be writing the portion of this process that pertains to tessellating objects. That is, you will be breaking up a few "standard" 3D shapes — or primitives — into a lot of triangles or quads that, when stitched together, look as much like the desired shapes as possible. Flat-faced objects are pretty simple, and will come out looking just like the actual shape even when using a reduced number of triangles or quads. On the other hand, curved surfaces won't look exactly like the real thing. It is possible to get a better approximation of a curved surface by using more triangles or quads, but keep in mind that more-to-draw means more-to-compute, and a major motivation behind tessellating objects is to simplify the process of displaying them.

One of the most exciting aspects of this assignment is its "building-block" nature. In other words, you won't just hand-in your code and expect never to use it again, but rather you can expect to use some of your code even in your final project (or so we hope!). This, of course, makes paramount careful-planning and good design (but you already knew that).

2. The Assignment

For this assignment you will write an interactive program that constructs and displays 3D shapes. Depending on keyboard input, the program will display the model of a 3D house, a cube, a cylinder, a torus, a cone, or a sphere. Mouse input will control how the models are rendered – as wireframe or solid shapes. Pressing 'x' or 'y' will rotate the 3D shapes in space. Pressing '+'/'-' and '<'/'>' will control the tessellation level of the cylinder and the sphere, making them look more or less realistic.

Relax - most of the requirements are already taken care of. The example code for this assignment (see Assignments on the course webpage) displays a multi-colored house shape. You can rotate the shape around the X and Y axes by pressing 'x' and 'y' on the keyboard. You can see the model in wireframe mode or solid mode by clicking the mouse anywhere in the window. You can switch to another shape (a hollow cube) by pressing 'b' on the keyboard; you can switch back to the house model by pressing 'h'. As always, the support code is nothing but a courtesy. We highly encourage you to move away from the awkward sphere_verts etc. and define your own generic data structure that can accommodate both a sphere and let's say a cone, a cylinder etc.; perhaps store face information etc. Keep in mind that later assignments will ask you to iterate over a scenegraph that will contain many instances of such objects.

What **you** need to do is expand the program by adding at least a tessellated cylinder, a tessellated sphere, a tessellated cone and a tessellated torus to the gallery of possible shapes. In addition, you will need to be able to control the tessellation level of these curved shapes.

To encourage you to start working early, we broke the assignment into two parts. The first part is algorithmic and is due in five days, in paper form, with the grad TA; it is worth 40% of your grade for this assignment, no late handins accepted. The second part is the actual program and is due in ten days, via electronic submission; the coding component is worth 60% of your grade for this assignment.

Begin by reading and solving the 'Assignment 2 (part 1): Stitcher Algorithm' handout (available under Assignments on the course website). When done, please move on to coding.

Download the Stitcher example code package (available under Assignments on the course website). It contains a stitcher.c and a stitcher.h file, a Readme.txt file, and an optional Makefile. Make sure you can compile and run the support code. Test the stitcher program (keyboard and mouse controls). Read through the source code once, paying attention to comments. For the purpose of this assignment, ignore the code related to viewing and manipulating the shape; we've already taken care of these steps for you. Concentrate instead on the make_* and draw_* functions (labeled TODO). These are the main functions you will have to modify. Again, we strongly encourage you to move away from the example code and think of more general and elegant data structures to store objects. C++ and OOP concepts should help here; but you can still code in plain C if you wish.

Make sure you understand how shape vertices are defined and stored (look into stitcher.h), and how they are stitched together into faces (look into stitcher.c). A pen and a piece of paper may come in helpful at this point: draw the shape of the cube and number the vertices according to stitcher.h. To make sure you understood the cube representation and how to access the cube geometry, locate in stitcher.c the function that helps draw the hollow cube. Modify this function, by adding a bottom face and a top face to the hollow cube shape (either triangles or quads). When adding the faces, make sure you specify face-vertices in counter-clockwise order. Done? Good.

OK. At this point you *really* need to stop and complete the 'Assignment 2 (part 1): Stitcher Algorithm' handout, if you haven't already. Because next, you need to write the routines that, given a ray, a height (optionally) and two tessellation values, will compute the vertices and normals, then draw the actual three-dimensional triangles or quads needed to "simulate" the object. You will be using the draw_triangle and draw_quad support code functions (or, likely, you'll write your own version of draw_quad) to draw your shapes. You are NOT allowed to use any other OpenGL or GLU commands but those already called in draw_triangle and draw_quad.

In the draw_* functions, note the use of glVertex4f instead of glVertex3f. When we call glVertex3f, OpenGL adds a w=1 to the vertex, then calls glVertex4f. It's time you start being aware of "why" it adds the w. So, from now on, stick with glVertex4f, set the w yourself, and avoid glVertex3f.

You only need to tessellate four objects: a cylinder, a sphere, a cone, and a torus (aka a doughnut). The tessellation values have the following meaning: the first parameter should represent the number of "stacks," and the second should be the number of "slices." If you have trouble visualizing how these parameters affect the shapes, think of slice lines as longitude, and of stack lines as latitude. Use the pseudocode in the Stitcher Algorithm as a guide when writing the actual code. For the exact shape specs,

see the next section (they are slightly different from the algorithm specs). As in the algorithm handout, you may assume the tessellation values will never go above 50 slices x 50 stacks.

For each shape we require you to also compute and display the object normals. Your normal computation needs to be done and displayed per vertex (see the Stitcher algo notes). Surface normals are vectors that are "normal," or perpendicular, to a surface. They are used for lighting calculations and shading, as you will see for yourself later in the semester. Please be careful when computing cross products; make sure the face vertices you use in the cross product are accessed in counter-clockwise order; your normals need to point outwards from the object, not inwards.

Hint: To draw the vertex normal, you need to tell OpenGL to draw a line segment running from the vertex V to some set point V' along the vertex normal N (where N has been normalized, i.e. has length 1). How do we compute V'?
Magic: V' = V + 0.2 * N (we chose 0.2 here arbitrarily; to draw a longer normal, use a higher value).

Because we don't use shading, in this assignment you will find Wireframe renderings more useful than Solid. Still, please be careful when passing the face vertices to draw_quad: you need to pass them in counter-clockwise order, or OpenGL will draw the faces backwards (and they will be invisible in later assignments). This is a common cause of much agony, so be careful!

Note: The tessellation parameters only make sense for values greater than a certain minimum value, depending on the shape and which parameter it is. As you might imagine, if you allow the parameter for the "slices" of a cylinder to go below 3, your cylinder is going to be flat.

Expected Behavior
Pressing 'H' or 'h' should draw the support-code house. Pressing 'B' or 'b' should draw the cube specified below. Pressing 'C' or 'c' should draw the cylinder specified below. Pressing 'S' or 's' should draw the sphere specified below. Pressing 'N' or 'n' should draw the cone specified below.

Mouse clicks should toggle between Wireframe and Solid rendering.

Pressing '+' or '-' should increase, respectively decrease the 'slice' tessellation value.

Pressing '<' or '>' should increase, respectively decrease the 'stack' tessellation value.

Pressing 'V' should display the vertex normals.

Cube — the hard-coded cube in the example code.

Cylinder — the cylinder has a height of one unit, and is one unit in diameter. The Y axis passes vertically through the center; the ends are parallel to the XZ plane. The cylinder is centered at the origin. So the extents are -0.5 to 0.5 along **each** axis. Note: this is different from the in class example, where the Y extent was from 0 to H. The default number of slices and stacks is 10 by 10.

Sphere — the sphere is centered at the origin, and has a radius of 1.0. The default number of slices and stacks is 10 by 10.

Cone — the cone is centered at the origin, has a height of 1.0 and a base radius of 1.0. So the extents are -0.5 to 0.5 along **each** axis. Note: this is different from the algorithm handout. The default number of slices and stacks is 10 by 10.

Torus — the torus is centered at the origin. You can use the following values as default, but you are allowed the change them. The torus must wrap around the Y axis.

1. The distance from the center of the tube to the center of the torus - 1.2
2. Radius of the tube - 0.2
3. Number of circles along the tube - 15
4. Number of circles across the tube - 40

The actual details of the tessellation are left up to you. An important consideration when tessellating shapes is that whenever the user modifies one of the drawing parameters (i.e., the orientation, tessellation values, drawing style, etc.), you will need to redraw all the triangles that compose an object. Some of these adjustments change how the object is tessellated, but when they don't, you should not recompute all the triangles, just redraw them. For this assignment, you are allowed to 'stitch' faces together in the drawing stage. However, keep in mind that in a real application you should also precompute the faces. In other words, you would need to keep track of all the triangles of a particular shape. We also expect to see in your code functions for multiplying 4x4 matrices and column vectors, and to see you making use of these functions.

EC Hint: this can be done in a simple, *object-oriented* and *extensible* (ooh boy, buzzwords!) way. You will want to take some time to think about organizing your code. Remember that this code will likely be used in upcoming assignments other than Stitcher.

Note 2: There are many ways to design your internal data structures for shapes. Here are some options – none is particularly more correct than any other, so you should stick to what you're most comfortable with.

• Construct a gigantic 1-dimensional array of all your point data, then carefully index into it when drawing your shapes (ouch).

• Construct a 2-dimensional array of all your point data: if you have N points, there are N elements in the first dimension. The second dimension would always be of magnitude 4 (since the points have 4 coordinates in our homogenous coordinate system, though the fourth coordinate will remain at "1" in this assignment)

• Construct an M x N x 4 array of all your point data for the N stacks and M slices. The third dimension would always be of magnitude 4. You may allocate it statically if your C/C++ programming is shaky, or you can go dynamically.

• Use an advanced data structure to organize your points. Lists or queues would be the most obvious choices.

• There are many other options! It may help to build a simple wrapper class or struct for your point data.

If you have questions about your initial designs, please do come to a TA on hours and ask for advice. You may be living with this code for an entire semester: you don't want to have to struggle with bad initial choices a few months down the line.

Note 3: Be careful how you handle boundary cases. We may not be very particular when grading your pseudocode, but a missing North Pole on the sphere or a shortened cylinder in the actual implementation will cost you points.
In the support code program we have given you several extra "special" shape slots which you can use in whichever way you want (pawns, fractals, geodesic etc). Particularly interesting "special" shapes will earn you extra credit. There is also an extra credit "mesh" slot for those of you interested in adding mesh support. See the Extra Credit section for details.

Note 4: Two common mistakes that can affect programs
1) When you update your scene, you should not call the draw function directly. Before you return, you should call glutPostRedisplay(), to tell GLUT to call your function. Your draw function should only be for drawing/rendering (and visibility determination if you do it).

2) Some of us call redraw in the timer function. This can cause a delay in input if your timer function does not occur frequently enough. So, if your keyboard function is called, it should somehow store the state so that your timer function can process the update. In some applications, where redraws are infrequent, you may invoke a redraw in your input handlers (glutPostRedisplay).

Hopefully this will help with your next assignment.

3. Extra Credit

Well, for one thing there are lots of possible special objects. This is your chance to be creative. You can include a geodesic shape as special shapes. You can also include an implementation of a mesh loader/displayer under the Mesh option. Again, meshes are not required for Stitcher and though support for meshes should not be too hard once you've gone through the basic shapes, it will be strictly extra credit for all assignments in cs1566. You could also make an interesting algorithmic shape (using a fractal, perhaps). At the very least, you could add a top and a bottom to the cylinder (hint: think pizza slices). You won't get any EC for playing with color or basic user input this time (color is there only to help you debug), but if you do some fun animation with the timer we'll give you… let's say lots.

For those interested in the mesh option we provide a few mesh examples (download them from Assignments on the course website). The meshes are in the Wavefront OBJ file text format (exact details in: http://www.javaview.de/guide/formats/Format_Obj.html).

Some files specify vertex normals and texture coordinates. You can ignore the texture coordinates, but you might want the vertex normals. To specify vertex normals, use glNormal3f(x, y, z), preferably interspersed with the glVertex4f() calls. Note:

- x, y and z correspond to the coordinates of the normal
- until glNormal3f() is called again, any calls to glVertex4f will create a vertex with the normal previously specified.
- glNormal3f() does not have to be called within glBegin()/glEnd(), but it probably will be and there's nothing wrong with that. ☺

Note: EC will not count en lieu of buggy or missing features, so don't try EC unless you're done with the basic assignment.

4. Grading

Task	Points
Stitcher algorithm handin	40
Bottom and top of cube show up	3
'b' draws unit cube, 'v' shows the cube normals	3
'c' draws default cylinder, 'v' shows the cyl normals	15
's' draws default sphere, 'v' shows the sphere normals	10
'n' draws default cone, 'v' shows the cone normals	10
'm' draws default torus, 'v' shows the torus normals	8
'+'/'-' increases/decreases slice tessellation	4
'<'/'>'increases/decreases stack tessellation	4
README turned in	3
TOTAL	100

EC points are at the discretion of the grader, somewhere between 5 and 20 points are possible.

5. Example Code

• stitcher.h — header file for the program
• stitcher.c — main file for the program
• Makefile — Makefile; if using Windows VS or Mac OSX , you won't need it.
• Readme.txt — text file containing your name, ID, description of Extra Credit work, and any additional comments.

6. Handing In

To hand in this assignment, follow the Submit link on the course website and upload the following files:
• your modified source files (stitcher.c(pp) and stitcher.h, any additional h/c/cpp files)
• your Makefile (if any)
• filled in Readme.txt file (please save it as plain text).

Do NOT submit any executables.
Use ftp only as an emergency backup plan (and notify the TAs immediately via email).