Project 2 Shapes

Introduction to Computer Graphics, Fall 2012

1 Introduction

One of the fundamental applications of computer graphics is to display three-dimensional scenes. The catch, however, is that 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 to compose a scene using only triangles, then project 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.)

In this assignment, you will be writing the portion of this process that pertains to *tesselating* objects. That is, you will be breaking up the "standard" shapes—or *primitives*—into a lot of triangles that, when put together, look as much like the desired shapes as possible. Flat-faced objects will be pretty simple, and will come out looking just like the actual shape. 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 using more triangles, but keep in mind that more to draw means more to compute, and a major motivation behind tesselating objects is to simplify the process of displaying them. Check out the demo for the project in **/course/cs123/bin/cs123_demo** to get an idea of what a completed program would look like. Make sure you show the Shapes dock with the *Toolbars > Shapes/Sceneview* menu item, since it won't be visible by default.

One of the most exciting aspects of this class is the "building-block" nature of the assignments. In other words, you won't just hand in your code and expect to never use it again, but rather you can expect to your code to be reused throughout the course. This of course means that careful planning and a good design are paramount. Throughout the rest of this handout there will be a couple of "gems" that we highly suggest you consider in your design.

2 Demo

The demo, as you have seen (you have played with it, haven't you?), presents you with a list of shapes from which to choose, the option of selecting either a solid or wireframe view, and some sliders to allow you to modify how finely tesselated the object is. You can drag and drop around the 3D canvas with your mouse to view all sides of the shape. Drag using the right mouse button to change the camera angle. Use the scroll wheel to zoom in and out.

Notice that if you use really high tesselation values (i.e. using more, smaller triangles), even the shapes that have curved surfaces look really good. On the other hand, you may notice that the time it takes to draw the shape is roughly proportional to the number of triangles used in the tessellation. If you try to rotate a finely tesselated shape, you will eventually notice slowdowns in the draw speed. The problem will become even more apparent if there were *multiple* detailed objects in the scene, as there will be later in the course. You should also keep in mind that there is another end to the spectrum; it is sometimes not desirable to allow tesselation values below a certain point, as you lose determining features of the shape you are tesselating.

3 Requirements

The wireframe/solid transition as well as the shape's orientation is all taken care of by the support code; you don't need to worry about that. What you must do, however, is write the routines that, given a shape and two tesselation values, compute the vertices, edges, and normals of the triangles needed to approximate the shape in 3D. You will be using the OpenGL commands *glNormal3f* and *glVertex3f* to draw your shapes. You are required to tesselate four objects: a cube, a cylinder, a cone, and a sphere.

The tesselation values will take on different meanings depending on the object you are tesselating. For the radially symmetric shapes (sphere, cylinder, and cone), the first parameter should represent the number of "stacks," and the second should be the number of "slices." Take a look at the demo if you have trouble visualizing how the parameters affect the shapes. Slice lines are like longitude, and stack lines are like latitude.

Note: *The tesselation 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. To avoid behavior such as this have a look at the minimum parameter values used in the demo. In your implementation of this assignment, you should bound the parameters at appropriate values to avoid unsightly or improper results.*

The actual details of the tesselation are left up to you (including both how to generate the triangles and the surface normals for them). Surface normals are vectors that are "normal," or perpindicular, to a surface. They are used for lighting calculations and shading, as you will see for yourself later in the semester. It is possible to generate a normal for each triangle (which uses less memory) or for every vertex (this produces better shading, making your shape look more curved). It is only possible to use normals per-triangle where the surface is flat. Keep in mind there are methods of tesselation that depart greatly from what is shown in the demo; it is up to you to try to find them. As for the shapes you will be tesselating, look in the *Shape Specification* section for details. Explain your tessellation design decisions in your **README_Shapes.txt** file.

Important: While the specifics of your tesselation code are up to you, you are expected to design your program in an extensible, object-oriented way. This means no 1000 line branch structures and minimal code duplication. You *will* lose points if you do not follow these guidelines.

Another important consideration when tesselating shapes is that whenever the user modifies one of the drawing parameters (i.e., the orientation, tesselation 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 tesselated, but when they don't, you should *not* recompute all the triangles, but rather *redraw* the triangles you have already computed. In other words, you will need to keep track of all the triangles drawn for a particular shape. Again, this organization can be done in a simple, *object-oriented* and *extensible* way. You will want to take some time to think about organizing your code. Remember that this code will be used in upcoming assignments other than Shapes.

The demo also has additional "special" shapes: a Möbius strip, Sierpinski triangle, and even a Utah teapot. These examples are intended to inspire you to create your own special shapes. In your program we have given you extra radio buttons which you can use in whichever way you want. Particularly interesting extensions will earn you extra credit. See the *Extra Credit* section below for details.

4 Shape Specification

Now when we say "tesselate some shape," you're going to need a lot more information than just tesselation parameters. The location of a shape as well as its size and orientation are important in writing the good, consistent tesselators that you will need for later assignments. To simplify matters and eliminate lots of special cases, a trick that is often followed is to deal with a shape only at some set location, and mathematically move or distort the shape to meet the demands of a particular scene. Your TAs have decided to be nice, and they are going to let you use this trick rather than force you to write completely generic tesselators for all the shapes (you have no idea how lucky you are...). Here are the specifications for the shapes you will be tesselating (all are centered at the origin):

Cube: The cube has unit length edges. Hence, it goes from -0.5 to 0.5 along all three axes.

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. So the extents are once again -0.5 to 0.5 along all axes.

Cone: The cone also fits within a unit cube, and is similar to the cylinder but with the top (the end of the cylinder at $Y = 0.5$) pinched to a point.

Sphere: The sphere is centered at the origin, and has a radius of 0.5.

5 Support Code

You will be using the same support code as you used last time. The skeleton code ships with triangle-drawing example code to get you started. See the OpenGL lab for additional reference material.

Begin by looking at the *ShapesScene* class in shapesscene.cpp/h. At a minimum, you'll need to fill in the *renderGeometry*(...) method. As before, you can use the static *Settings* object to get information from the GUI. The relevant members include:

```
/* Shapes */
int shapeType;
int shapeParameter1;
int shapeParameter2;
float shapeParameter3;
bool useLighting;
bool drawWireframe;
bool drawNormals;
```

Shape parameter 1 will normally control the X tessellation parameter, and shape parameter 2 will normally control the Y tessellation parameter. The third shape parameter can be used at your own discretion for use in any extra credit shapes you might create (again, see the Extra Credit section below). The shapeType parameter will select the shape that is to be drawn. Values include SHAPE_CUBE, SHAPE_CONE, SHAPE_SPHERE, SHAPE_CYLINDER, SHAPE_SPECIAL_1, SHAPE_SPECIAL_2, SHAPE_SPECIAL_3.

You will have to create your own subclasses for Shapes. Feel free to create files and directories; extend the support code as needed. If you do create new files, be sure to update CS123.pro to point Qt creator and the compilers at your new source files.

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 whatever 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.

- 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)
- Use an STL or Qt structure to organize your points. A list or a deque 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. You may want to build up a list using the stl, then flatten it into an array after it's been finalized.

You will have to do something similar for your normals. If you have questions about your initial designs, please do come to a TA on hours and ask for advice. Again, just to emphasize: you'll be living with this code for the entire semester: you don't want to have to struggle with bad design choices a few months down the line.

Note that the individual triangles must be passed to OpenGL in counter-clockwise order, or it will draw the faces backwards (and they will be invisible). This is a common cause of much agony, so be careful! Another mistake that will create problems is using transformations. In Shapes, the necessary transformations are taken care of for you. Concentrate on generating all the verticies and normals for a shape instead.

Compile your project by typing *qmake* (this will generate a Makefile), and then *make* (this will build your code). The executable, as before, is called *CS123.*

6 Extra Credit

Again, this is your chance to be creative. There is room in the stencil code for some extra shapes. You could make an interesting algorithmic shape (a fractal, perhaps), or you could find a mesh file somewhere and implement a mesh reader. You could also write code that can tessellate your mesh at arbitrary positions, rotations, and sizes (although you'd have to modify the GUI a bit to allow for these extra parameters). Again, we're looking for quality over quantity. Be creative and have fun!

7 Handing In

You'll probably notice that you'll be handing in your Brush code along with your Shapes code. Doon't worry about it. Rename your README from Brush to README_Brush.txt to avoid any confusion.

To hand in your assignment, type ***/course/cs123/bin/cs123_handin shapes*** at a shell prompt. Please include a README_Shapes.txt with your handin containing basic information about your design decisions and any known bugs or extra credit.

8 FAQ

8.1 I am very confused about this assignment. What am I supposed to do?

We understand that there is not much in the textbook with regard to tesselation. The math in this assignment doesn't go beyond that which you learned in high school geometry, unless you attempt more complex shapes. A good approach to take if you are flustered is to try to place the triangle in the support code somewhere useful, such as on the side of the cube. Calculate, on paper, where the rest of the triangles will be placed, and think of a good way to parameterize their placement.

If you're still feeling confused about shape parameterization, go to the Sci-Li and get a college calculus textbook; the MA18 book has a section on parametric representation, for instance. And be thankful we're not asking you to integrate over these solids… yet.

8.2 Some of my triangles just aren't appearing on the screen. What is wrong?

There are a few possibilities. The first is that you are simply drawing the triangles in the wrong place. The second is that you are specifying the coordinates of triangles in the wrong order. Remember, if you don't pass the vertices in *counterclockwise* order (with respect to the normal of the triangle) they will not appear on the screen. The third possibily is that you're calling *glVertex*() outside of a *glBegin*()/*glEnd*() block, in which case nothing will happen. If none of these help, make sure that your drawing is taking place inside of *renderGeometry*(). This is called automatically when OpenGL is ready to draw triangles.