Assignment 2: Tessellation of shapes

Written Homework Due: Tuesday, March 24 (9:30 AM)
Programming Project Due: Wednesday, March 25 (5:00 PM)

1 Introduction

Graphics cards are best at drawing triangles. In this assignment, you will be handling the process that pertains to *tessellating* objects. That is, you will be breaking up the "standard" shapes into a lot of triangles that, when put together, look as much like actual shapes as possible. Flat-faced objects will be pretty simple, and will come out looking just like the actual thing. 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 the whole idea behind tessellating objects is to simplify the process of displaying them. Check out the demo for the project (choose shapes from the lab menu) to get an idea of what a completed program would look like.

2 Demo

The demo lets you choose between a few different shapes, choose whether they wish to view the shape as a solid or a wireframe, and the orientation of the shape. The interface also allows you to modify how finely tessellated the object is. That is, how small the triangles are. Notice that if you use really high tessellation values (i.e., using more, smaller triangles), even the shapes that have curved surfaces look really good. However, 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 tessellated shape, you have very jerky movement. You should also keep in mind that there is another end to the spectrum; it is sometimes not desirable to allow tessellation values below a certain point, as you lose determining features of the shape you are tessellating. For an example, the demo's cylinder uses 3 as a lower bound for the first tessellation value. Given anything less the cylinder would collapse to a flat surface.

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 will do, however, is to write the routines that, given a shape and two tessellation values, will compute the actual three-dimensional triangles needed to "simulate" the surface of the object. You will also need to write the OpenGL code to draw the triangles on the screen. You only need to tessellate four objects: cube, cylinder, cone, and sphere.

As for the tessellation values, they will take on different meanings depending on the object you are tessellating. For the radially symmetric shapes (sphere, cone, and cylinder), the first parameter

should represent the number of "slices", and the second should be the number of "stacks". For the cube, it really only makes sense to utilize one of the tessellation values, which would be proportional to the amount one of the faces is subdivided. For all of shapes, the best way to visualize what you must do is to play with the demo. The actual details of the tessellation, however, is left up to you, and *must be described in detail in your written handin* (including both how to generate the triangles and the surface normals for them). There are, in fact, methods of tessellation 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 tessellating themselves, look in the **Shape Specification** section for details.

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 others don't. **Do not** recompute all the triangles, just redraw them, when parameters change doesn't affect the actual tessellation. In other words, you will need to keep track of all the triangles drawn for a particular shape.

The demo also has two "special" shapes. They don't have to use the tessellation values. In your own program, you are free to do whatever you want for extra shapes. Particularly interesting "special" shapes will earn you extra credit. See the Extra Credit section for details.

4 Shape Specification

Here are the specifications for the shapes you will be tessellating (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 axis.

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 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 point of the cone is at the positive Y axis. So the extents are once again -0.5 to 0.5 along all axes.

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

5 Support Code

The support code is in the project you've already downloaded. You will create a new class (`Shape`) and additional classes to extend it. Remember that a good object oriented design principle is to keep as much functionality in the base class as possible to avoid replicating code. Since all your shapes are going to be composed of triangles, they will all be drawn the same way for example. The `ShapesInterface` class has methods for getting the widget parameters and calling the appropriate `changed` routine in `ShapesUI`. Again, you should never edit the `ShapesInterface` C++ files directly; use Fluid. You will need to edit `ShapesUI` so that the shapes actually are created (`ShapesUI::change*`) and drawn (`ShapesUI::draw`). All the places were you might need to add code are flagged for you with comments. You should not need to edit anything beyond these markers.

You will be using OpenGL to draw the triangles. The shading and lighting is taken care of; you simply need to tell OpenGL the points and normals for each triangle. Each triangle is bracketed by a `glBegin( GL_TRIANGLE)` and `glEnd` call. Whenever you call `glVertex` between these two functions, OpenGL sends that point to the drawing pipeline *with whatever data is already set*, i.e., the normal will be whatever the last call to `glNormal` set it to be.

This leads to an interesting hack in OpenGL — you can make "curved" triangles by setting a different normal for each vertex. This makes curved surfaces look much better (we'll cover how this works later in the course).

An important thing to note is we cannot simply write the vertices to OpenGL in any order. OpenGL assumes that the vertices are passed in counterclockwise order (judged as if you were looking at the triangle). If you accidentally specify the coordinates in clockwise order, the triangle won't appear. In fact, the triangles on the opposite side of your shape will appear. This will be especially noticeable when you rotate your shape around.

Points and vectors are provided in `vecmath.h`, you should use these data types to store your points and normals.

6 Coding style

You must use inheritance for your shape class — you will be unhappy later if you don't. You will want some method for adding and storing triangles (I recommend having an "add quadrilateral" method as well). You will also want a draw routine, and probably a tessellate as well (deleting and re-allocing the shape just to change the number of triangles is not nice).

- **Memory management:** Again, I recommend using the STL to handle this. You will be penalized heavily for simply allocating a huge number of triangles and hoping you don't run out. You should also avoid new'ing and deleting shapes any more than necessary (in fact, you can avoid this altogether — check out the way the different labs are managed in `GLWindow`).
- **Efficiency:** You do not want to be re-calculating the points and normals every time they're drawn; you simply want to draw them. Hence, you will need some method of storing them.
- **Debugging:** Do not use doubles as your `for` loop variables. (If you want to see why not, do a loop from 0 to 1 with a step size of 1/2. Do the same thing with 1/3. What happened?). You should also assert that the first and last values are what you expect them to be, i.e., that the top and bottom of the cone are at ± 0.5. There is a method in `vecmath.h` called `isZero`; you should use this when comparing doubles that may not be exactly equal due to round-off error.
- **Other debugging hints:** First, write any numbers that are doubles with a decimal point, i.e., 0.0, not 0. This will tell readers of your code, and the compiler, that this is a double and not an integer. Second, whenever you divide by an int and assign to a double (or expect a double out) explicitly cast the int to a double. Otherwise, it will do integer division.

7 Extra Credit

Add optional special objects. This is your chance to be creative. Make it an interesting algorithmic shape, or use a mesh data file. For more information about mesh data files, see the FAQ section.

There is an obvious method for tessellating spheres, and a really cool way of doing it (which the demo uses). Using the former method is acceptable, but using the latter will earn you extra credit. Choose wisely. More info on the alternate tessellation after the initial handin.

Replace the cylinder, sphere, and cone draw method with one that uses triangle strips and triangle fans (obviously, you should keep the original draw method for the cube). Note that all of these shapes have the same basic triangulation — to get full credit, you should only implement one draw routine and just pass in the point and normal information for the appropriate shape.

8 FAQ

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

  There are two possibilities. The first is simply you are 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 coordinates in counterclockwise order (with respect to the normal) to the `drawTriangle` function, they will not appear on the screen.

- **Mesh data files.** Mesh data files are available for download at

  `http://web.engr.oregonstate.edu/~grimmc/meshes.php`

  They all have the same format. The vertices are listed first, numbered from 1. The faces are listed next; most of the meshes are triangular (each face has three vertices) but some are not. The faces are defined by a list of 3 (or more) vertices. See the URL above for pictures and more info.

- **Triangle tricks.** Some tricks I've found useful for doing tessellations.

  - When making squares, draw only one triangle. Check that your triangle making routine works. Then draw the second triangle but not the first. Then draw both together. This makes it much easier to visually determine if the triangles are going in the right places and are oriented the right way.
  - Do the same trick for alternating rows. This can be accomplished by laying out the vertices in a regular pattern (say a circle of them) and making faces from them (say a band around the cylinder). Write all the faces down, then comment out half of them.
  - I find it much easier to do the cube hard-wired by hand, *e.g.,* write a routine to do each face. You can do it with some clever indexing, but I find I spent more time debugging my clever indexing than just writing it out by hand the first time.
  - Walk through your triangles (in the debugger) as they're drawn and draw them on a piece of paper at the same time (put a break point in the draw triangle routine). This can be tedious, but you'll get really good at drawing things in 3D and it's a (relatively) easy way to find out which triangle is not going where you expect. For those of you unfamiliar with Visual Studio, right click on a variable to bring up the Quick Watch window. Type out the variable name w/index, then say "Add Watch". This is an easy way to get all variable values to show up at each loop step without typing them in by hand.