

Assignment 1: Shapes

Help Session: Tuesday, September 22, 7:00 pm, location TBA

Algo Due: Tuesday, September 22, 5:00 pm

Program Due: Friday, September 25, 11:59 pm

1 Introduction

Computer graphics often deals with images which, as far as most of this course is concerned, are of three-dimensional scenes. The catch, however, is that your 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) In this assignment, you will be writing the portion of this process that pertains to *tessellating* 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 tessellating objects is to simplify the process of displaying them. Similar to Brush, a demo of the project is included in the stencil code we give you.

One of the most exciting aspects of this class is the “building-block” nature of the assignments. In other words, you won't just handin your code and expect never to use it again, but rather you can expect to see the fruits of your labor from this assignment even in the final one (or so we hope!). This of course means that careful planning and a good design are paramount (but you already knew that). Throughout the rest of this handout there will be a couple 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 viewing in either a solid or wireframe mode, ability to manipulate the orientation of the shape, and some sliders to allow you to modify how finely tessellated the object is. 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, there is one hitch. 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 will eventually notice slowdowns in the redraw speed (especially if we were drawing multiple shapes to the screen, as you will be doing 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 tessellation values below a certain point, as you lose determining features of the shape you are tessellating. For example, the demo's sphere uses 2 as a lower bound for the first tessellation value, and uses 3 for the second. Given anything less, the sphere would collapse to the point where it wouldn't just look ugly; it simply wouldn't pass as a sphere at all.

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 tessellation values, will compute the actual three-dimensional triangles needed to “simulate” the object. You will be using the OpenGL commands `glNormal3f` and `glVertex4f` to draw your shapes. You only need to tessellate four objects: a cube, a cylinder, a cone, and a sphere.

The tessellation values will take on different meanings depending on the object you are tessellating. 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.” 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. Take a look at the demo if you have trouble visualizing how the parameters affect the shapes. Slice lines are like longitude, stack lines are like latitude.

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. To avoid behavior such as this have a look at the minimum parameter values used in the demo.

The actual details of the tessellation 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 perpendicular, 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 when you are thinking about how to tessellate that there are 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 when they don’t, you should **not** recompute all the triangles, just redraw them. In other words, you will need to keep track of all the triangles drawn for a particular shape.

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 be used in upcoming assignments other than Shapes.

The demo also has two “special” shapes: a torus and a geodesic sphere. These are intended to inspire you to create your own shapes. In your program we have given you two extra “special” slots which you can use in whichever way you want. 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.

4 Shape Specification

Now when we say “tessellate some shape,” you’re going to need a lot more information than just tessellation parameters. Where a shape is found in the scene, as well as its size and orientation are important in writing the good, consistent tessellators 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 transform (scale/rotate/translate...) the shape to meet the demands of a particular scene. 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 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

To start work on this assignment, you will need to first copy the support code to your directory (preferably a subdirectory thereof). The support code can be found in `/course/cs123/asgn/shapes`. A Makefile is provided, and all the modifications that need to be made to it are for you to add the names of your object files to the `SRCFILES` line. Also provided is a [compilable] `main.cpp` which will help you see how you put the pieces of the support code together. In addition you are provided with stencil code for `MyShapesCanvas` and `MyShapesControl` that you will write. The support code comes prepackaged to draw a single triangle; this should help you get started on the right track.

There are two main classes that you will need to use together. The first of these two classes is the control interface for the program, `MyShapesControl`:

```
class MyShapesCanvas;

class MyShapesControl : public QObject
{
public:
    MyShapesControl(MyShapesCanvas *scene = NULL);
    virtual ~MyShapesControl();

    virtual void setScene(MyShapesCanvas *scene);

    virtual void setShape(int shape);
    virtual void setStyle(int style);
    virtual void setRotation(double x, double y);
    virtual void setParameters(int p1, int p2);
    virtual void setScale(float s);

protected:
    MyShapesCanvas *m_scene;
};
```

Constants: (from `ShapesCanvas.h`)

```
SHAPE_NONE = 0
SHAPE_CUBE = 1
SHAPE_CONE = 2
SHAPE_CYLINDER = 3
SHAPE_SPHERE = 4
SHAPE_SPECIAL = 5
SHAPE_SPECIAL2 = 6
SHAPE_MESH = 7

STYLE_SOLID = 0
STYLE_WIREFRAME = 1
```

This class will provide you a means of getting all of the user’s input. You need to fill in the various methods of `MyShapesControl` for the class to do something meaningful. The `setShape` method is called when the user chooses a shape with the “Shape” pulldown menu, and is passed a `SHAPE` constant (defined in `ShapesCanvas.h`) that corresponds to the selected shape. `setStyle` is called when the user selects one of the “Rendering Options” radio buttons, and likewise is passed the appropriate `STYLE` constant (also defined in `ShapesCanvas.h`). When the user changes the orientation angle (with the rotation sliders), `setRotation` is called with the new angles. When the user modifies one of the parameters, `setParameters` is called with the new values. Lastly, `setScale` is called when the user changes the scale slider. The `setScale` call is new this year, as we wanted to give you more functionality and expose you to more of the OpenGL API. To set the scale of your scenes you will need to call `glScalef`; see the GL section below.

In Brush, you were dealing solely with two-dimensional images, and used a subclass of `CS123Canvas` called `BrushCanvas`. For three-dimensional applications, instead of a `CS123Canvas`, you will be using `CS123OpenGLCanvas` and its subclasses. In particular, for Shapes you will be subclassing off of `ShapesCanvas` in your implementation of `MyShapesCanvas`. Here is the pertinent information for `MyShapesCanvas`:

```
class MyShapesCanvas : public ShapesCanvas
{
public:
    MyShapesCanvas();
    virtual ~MyShapesCanvas();

    // This is inherited from ShapesCanvas. We will draw the shape herein.
    // This is the main drawing method where your OpenGL calls should go.
    virtual void drawScene();
};
```

This class subclasses `ShapesCanvas` which subclasses the generic `CS123OpenGLCanvas`. In order to set the orientation of the scene you are viewing, you should use the `setRotation` method defined in `ShapesCanvas`, to which you can pass the values you get from `MyShapesControl`. Probably the most important method of all is the `drawScene` method, which you should redefine to draw your shape. This is all pretty simple, but there remains one important thing to tell you: how you actually draw things on the screen. This where the OpenGL commands come in:

- Notes:
 - For more useful information on the OpenGL commands see the OpenGL handout on the **References** section of our site. **Please note that some of this is geared towards later assignments.** You will not need to change drawing colors in shapes, for instance.
 - The skeleton code will arrive drawing a triangle to get you started
 - Please pay close attention to the calling semantics of `glVertex4f()`, `glNormal3f()`, `glBegin()`, and `glEnd()`. It is not completely intuitive, but if you’re careful there will be no problems.
- `glVertex4f(float x, float y, float z, float w)`
 - `x`, `y`, `z` and `w` correspond to the coordinates of the vertex. **For your purposes, the `w` coordinate will always equal 1.** This is called the homogenous coordinate, and it is a clever linear algebra trick that simplifies matrix transformations. See the lecture notes on transformations. (*Note:* There is a `glVertex3f()` call that automatically specifies the `w=1`, but the TA staff thinks it would be good for everyone to think about **why** we specify `w=1`)

- glVertex4f() will send the specified vertex coordinates to GL, and GL will use its current state to fill out the rest of the vertex information. (This "current state" includes the current normal and current color, among other things)
- must be called between glBegin() and glEnd() (called "chronologically," or in the flow of control).
- **glNormal3f(float x, float y, float z)**
 - 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 specified herein.
 - glNormal3f() does not *have* to be called within glBegin()/glEnd(), but it probably will be and there's nothing wrong with that.
- **glScalef(float x, float y, float z)**
 - x, y and z correspond to the x, y and z axis scales of the objects drawn.
 - Be sure to call this function *before* a glBegin() / glEnd() block.
- **glBegin() / glEnd()**
 - glBegin() takes one parameter. For cs123, this will probably always be GL_TRIANGLES, though you could optimize your code to use more efficient data paths. You should call glEnd as soon as you are finished making calls to glVertex4f().
 - Note that when the program is inside a glBegin() / glEnd() block, many GL calls change their semantics; most importantly, several calls cease to affect anything. This will not cause many problems in shapes, but in later assignments you may run into trouble with this if you're not careful. (You've been warned!)

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.
- 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 structure to organize your points. A vector or a deque would be the most obvious choices. If you're new to the stl, be sure to check out the C++ intro lecture slides on the cs123 site for some standard stl examples which shouldn't be hard to convert for use in shapes.
- 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 vector 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. You'll 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 that the individual triangles must be passed to GL 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 so concentrate on generating all the vertices and normals for a shape instead.

The other two important methods, both defined in superclasses of `MyShapesCanvas`, are `redraw` and `setDrawMode`. `redraw` will force a redraw of the scene and eventually call your implementation of `drawScene` in `MyShapesCanvas`. `setDrawMode` will switch the scene from wireframe mode to solid mode and back. It takes one of the `STYLE` constants defined in `ShapesCanvas.h` as a parameter. Note: Unlike Brush, you should not be calling the `update` method of `CS123OpenGLCanvas` to redraw your OpenGL scene; it is important that you use `redraw` instead.

6 Extra Credit

Well, for one thing there are the lots of possible special objects. This is your chance to be creative. The demo includes a torus and a geodesic sphere as its special shapes. It also includes an implementation of a mesh loader/displayer as under the Mesh option. Again, meshes are not required for Shapes 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 cs123. You could also make an interesting algorithmic shape (using a fractal, perhaps).

There is an obvious method for tessellating spheres, and another really cool way of doing it. Using the former method is acceptable, but using the latter will earn you extra credit. Choose wisely. Check the demo's Special 2 for an example of an alternative tessellation.

7 Handing In

To hand in, just type `/course/cs123/bin/cs123_handin shapes` from your shapes directory. You should also be able to type `make handin`, which essentially does the same thing. Please include a README with your handin containing basic information about your design decisions and any known bugs or extra credit.

Handout:	Thursday, September 17	
Help Session:	Tuesday, September 22	7:00 PM, location TBA
Algorithm Handin:	Tuesday, September 22	5:00 PM in the cs123 handin bin on the 2nd floor
Electronic Handin:	Friday, September 25	11:59 PM

8 FAQ

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 regards to tessellation. 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 SciLi 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)

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 possibility 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 `MyShapesCanvas::drawScene()`. This is called automatically when GL is ready to draw triangles. (Or when you explicitly call `CS123OpenGLCanvas::redraw()`)

3. **Why are the shapes a funny color?**

We use colored light to make it easy to see when shapes are being drawn inside out. This is supposed to be a "feature", though you can always override it in a subclass if you are so moved.