

# CS 4620 Program 2: Model

out: Tuesday 23 September 2008

**due: Friday 10 October 2008**

## 1 Introduction

In this assignment you will work on a 3D modeling system that uses simple primitives organized in a transformation hierarchy. The framework provides the user with a perspective view along with transformations, lights, and the camera.

The window layout, camera control, a tree-style hierarchy view, and type-in transformation and property editors are provided, as well as the beginnings of more sophisticated object manipulation. You will write the code that constructs triangle meshes for all the shapes in the scene, and performs graphical manipulation of transformations.

## 2 Framework overview

This section gives an overview of how the modeler works; there are more details in the following sections. There is a brief user guide on the assignment's web page, which details the operation from the user's point of view.

The modeler's central data structure is a tree. At the leaves are the individual objects in the scene, which can be cubes, spheres, cylinders, lights, or cameras. At the intermediate nodes are 3D affine transformations. Each transformation affects the world-space position of all the geometry below it, which means that the modeling transformation for any particular object is the product of all the transformations along the path from the root to that object's leaf node. Cameras and lights appear in the hierarchy, but they are special in that they are only allowed at the top level (as direct children of the root) and they are limited in number. Exactly one controllable camera exists (its view is displayed in the perspective window), and between zero and eight lights can exist (the eight-light limit is imposed by OpenGL).

The main window displays a tree control in the upper left pane, which you can use to select nodes in the hierarchy; you can also select nodes by clicking on the objects in the viewports. You can use menu commands to create new nodes, which will appear as children of the selected node. You can change the selected node's properties in the lower left pane. There are also three menu commands to manipulate the hierarchy: Group creates a new node with the selected nodes as children; Reparent moves the selected nodes so that they are children of another node; and Delete removes the selected nodes from the tree.

The right pane contains the perspective view of the scene. You can ALT-click (each mouse button will move in a different way) and drag in the view to move the view around

You can use the two panes together by selecting a transformation and then choosing a manipulator from the Edit menu. The manipulator, a graphical tool for changing the transformation's parameters, then appears in the viewport, and by dragging the parts of the manipulator around in the view you can alter the selected transformation.

The available leaf node types include three simple geometric objects: spheres, cubes, and cylinders. These objects are defined only in a canonical position and size, and all variations are handled by transformations. For instance, a sphere has no concept of a center and a radius; it is always a unit sphere, and the center and radius are adjusted by translation and scaling transformations that apply to the sphere. The only editable properties of these objects are shading parameters.

The program has a standard event-based structure, where an event generated by the user causes a change to the data structures and then the viewport is redrawn. Drawing in the viewport is done using the OpenGL 3D graphics API; all other drawing happens using the usual Java 2D graphics. When the scene is drawn, the tree of transformations is traversed, and at the leaves, the shapes are asked to generate triangle meshes in their own coordinates, which are then transformed into world coordinates as they are drawn.

### 3 Requirements

There are several core pieces of the program that need to be implemented before the framework becomes a functional modeling program. The general areas are mesh drawing, mesh generation, and manipulators.

#### 3.1 Scene drawing

To give you some practice using OpenGL, we leave it up to you to implement the drawing of the scene. This is probably the first thing you want to do, since it will help you debug your mesh generation and manipulators. There are three main tasks for this. First, implement `Mesh.render` to draw the mesh geometry. Then, complete `Transformation.render` to make sure the transformation hierarchy is properly applied. Finally, complete `Shape.render` to properly set the material parameters (such as diffuse and specular color) for each mesh. To help you debug the drawing, we have implemented mesh generation for cubes already.

#### 3.2 Mesh generation

The objects in the program are drawn as triangle meshes, but only for the cube (with its fixed 12-triangle tessellation) is there code to build the mesh. You must write code to generate shared-vertex triangle meshes to approximate cylinders, and spheres. In addition to the vertex positions and triangles connecting them, these meshes contain vertex normals for smooth shading. The density of the meshes is determined by a global flatness tolerance that the user can adjust (using the big slider on the right) to trade off accuracy for speed.

All primitives are sized to exactly fill the cube  $[-1, 1]^3$ .

The particular requirements for the individual shapes are:

1. *Cylinder*. The cylinder's axis should be the  $y$  axis. The side should be constructed with one row of triangles, and the end faces can be tessellated in any reasonable way you like ("reasonable" includes not generating degenerate triangles). The normals should be set up so that the sharp edges of the cylinder appear sharp, the flat ends appear sharp, and the side appears smooth.
2. *Sphere*. The sphere is a unit sphere centered at the origin. The normals should make the whole sphere look smooth.

### 3.3 Manipulators

The framework provides a means for editing transformations by typing in numbers. This can be useful if you want to apply a particular transformation exactly, but to get things where you want them it is much easier to position them interactively and visually.

A widely used approach to transforming objects in 3D is by the use of *manipulators*. Manipulators are user interface elements that appear in the 3D scene to give the user direct visual cues about the operations that can be done. Transformations are edited by clicking on parts of the manipulator and dragging. For instance, the translation manipulator displays the three coordinate axes, and the user moves the selected object by clicking on one axis and dragging in the direction of desired motion. By clicking on the center of the manipulator one can drag the objects around freely in the viewport, resulting in a translation parallel to the view plane.

It's important to remember that a manipulator operates on a transformation, not directly on an object itself. This transformation applies to all objects below it in the tree, and there may be other transformations above and below the transformation that is changing.

Transformations in the modeler are represented as a nonuniform scale followed by  $x$ ,  $y$ , and  $z$  rotations and then a translation; each manipulator affects one of these three pieces of the transformation. When the user adjusts one axis of a manipulator, the result is a change in exactly one number in the transformation (the  $x$ ,  $y$ , or  $z$  component of the translation, scale, or rotation).

When manipulators are drawn on the screen they should appear in 3D at the position of the origin in the coordinates below the transformation being edited. The arrows of the scale and translation manipulators should point exactly along the axes that correspond to  $x$ ,  $y$ , and  $z$  adjustments to the scale or translation component. We call these the *manipulation axes* of the manipulator. The circles of the rotation manipulator should be perpendicular to the three axes of rotation, unless the transformation being manipulated is below a nonuniform scale in the tree.

The framework provides infrastructure for drawing the manipulators and detecting when they have been clicked on. It is your job to position the manipulators and axes in world space and to map the user's mouse motions into changes to the selected transformation. You should implement the manipulators for translation, rotation, and scaling based on the requirements below. The general principle is that when the user drags the mouse in a direction that the manipulator handle can move, the handle should move with the mouse, staying glued to the mouse pointer. (There are exceptions where it is difficult or nonsensical to have the transformation follow the mouse.)

1. *Translation*. The translation manipulator displays three arrows that represent the directions of motion that will result from an  $x$ ,  $y$ , or  $z$  translation in the coordinates of the selected

transform. If the user clicks and drags exactly in the direction of the axis, the resulting translation should exactly follow the mouse. When the drag is not parallel to the selected axis, the translation should follow the mouse as much as possible while still operating along the selected axis. If the user clicks the center of the manipulator, you should apply a translation parallel to the view plane so that the origin of the manipulator moves the same distance as the mouse—that is, if you move the mouse to the right 5 pixels and up 2 pixels, the manipulator should move to the right 5 pixels and up 2 pixels, carrying the affected objects with it. This should be true no matter what transformations are above the selected one.

2. *Rotation.* The rotation manipulator displays three circles on a sphere surrounding the origin of the transformation's coordinates. Each circle is perpendicular to one of the rotation axes, and clicking on that circle and dragging performs a rotation around that axis. Because getting the transformation to follow the mouse is complicated for this manipulator, just map vertical mouse motion directly to the rotation angle.
3. *Scaling.* The scaling manipulator shows the three scaling axes by drawing three lines with small boxes at the ends. If the user clicks on one of these three axes and drags in the direction of the axis, a scale should be applied such that a point on an affected object that started under the mouse would stay under the mouse. If the user clicks on the center cube and drags, a uniform scale should be applied. The uniform scale need not follow the mouse; you can simply map vertical or horizontal mouse motion directly to scaling.

All manipulations should have the property that dragging the mouse back to the starting point and releasing causes no change.

## 4 Implementation Notes

This section contains discussion of how to implement the requirements; the suggestions here are just suggestions.

### 4.1 Scene drawing

To draw the mesh using OpenGL, fill out the `render` method of class `Mesh`. Be sure to send not only vertex positions, but also the corresponding normals for lighting. For `Transformation.render`, you will want to use `glPushMatrix` and `glPopMatrix` to manage the transformation matrices and you go down the hierarchy. OpenGL also provides transformation functions, such as `glRotatef`, that will facilitate implementation. Note that `super.render` calls `render` for all children of the transformation node, so you will want to use that where appropriate. We will provide a help session well before the project is due and also a lecture on the basics of OpenGL.

### 4.2 Mesh generation

Each of the shape types derives from the `Shape` class and for each you need to implement the `buildMesh` method to construct a `Mesh` object to represent the shape using triangles. The mesh must contain vertex positions, vertex normals, and a list of triangle indices. Each type of data should be packed into a single array of floats (or integers for the triangle indices) and, at the end

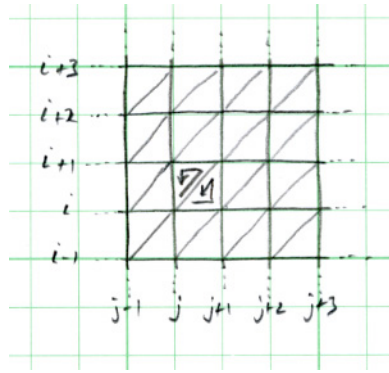


Figure 1: Diagram for surface triangulation.

of the method, they are used to set the mesh field of the shape to a new mesh created using the only constructor of the `Mesh` class. The data should be packed so that coordinates of an item are consecutive. For instance, the vertex mesh should be packed as:

$$x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, z_2, \dots$$

The number of triangles to use when creating a mesh is determined by a global flatness parameter that is found in the `THRESHOLD` field in the `Shape` class. Note that from a simple geometric argument you can find that the flatness of a circle that is divided into  $n$  segments is less than  $\pi/(4n)$ .

There are two things to note when building meshes. First, you will need to be very careful not to replicate vertices most of the time. If you replicate vertices for different triangles, the normals will not be smoothed properly when rendering. However, there are clearly times when you want to create sharp edges. To create a sharp edge in the mesh you must replicate the vertices and supply a different normal to each copy.

You can build your mesh as a  $(u, v)$  grid for all the surfaces of revolution (cylinders, spheres). The diagram in Figure 1 illustrates this idea. The gridlines can be evenly spaced in  $u$ , with the number of steps in  $u$  determined directly from the global flatness. For the sphere, regular subdivision in  $v$  is fine; for the cylinder only a one-row grid is needed.

### 4.3 Manipulators

The manipulators are all subclasses of `Manip`, and the main action is in the `dragged` method. This method has arguments `mousePosition`, which is the point in the viewport where the mouse is, and `mouseDelta`, which is the offset from the previous point that was reported. The `Manip` class also stores the point where the user first clicked in the field `pickedMousePoint`.

One good way of setting up the follows-mouse constraint for manipulation along axes is as follows. If the user clicks on a point on a manipulation axis, then drags to another point on the axes, this means the eye rays corresponding to the initial and final mouse positions both intersect the axis. By computing these intersection points you can figure out what the transformation is that takes one to the other. If the user's clicks aren't exactly on the the axis, this means the rays don't exactly intersect the axis, but we can get a reasonable result if we just use the point of closest approach (the "pseudo-intersection" point) instead of the exact intersection. These computations are pretty easy

to do using parametric lines, as we did in ray tracing. If you do it this way your work breaks down as:

1. Write a method to compute the parametric line (the viewing ray) in world space corresponding to a point in the image.
2. Write a method to compute the parametric line in world space that describes the manipulation axis. You had to do this anyway to position the manipulator in space.
3. Write a method to compute the pseudo-intersection of two parameteric lines, returning the two  $t$  values for the closest pair of points on the two lines.
4. Do the manipulation by using this machinery to compute the initial and final  $t$  values on the manipulation axis; then updating the transformation is a simple matter.

You may find that your manipulator behaves a little strangely if the vanishing point of the manipulation axis is in the image and the user drags past it. This is OK.

## 5 Extra Credit

“Starred” Problems:

1. Virtual trackball rotation: Add another rotation manipulator that functions like a virtual trackball. When active, the user should be able to click any point on the object, drag, and the object should rotate in the direction of the drag. Think of the mouse dragging motion as analogous to your palm’s motion on a trackball. How the trackball would rotate is how the object should rotate.

If you have other ideas, please talk to us first so we can steer you towards interesting additions and make sure we agree that it would be worth extra credit. Other possible ideas include adding more shapes, etc.

Let us re-emphasize that, as always, extra credit is only for programs that correctly implement the basic requirements.