CS 418 Program 3: Model I

out: Thursday, February 27, 2003
due: Monday, March 10, 2003

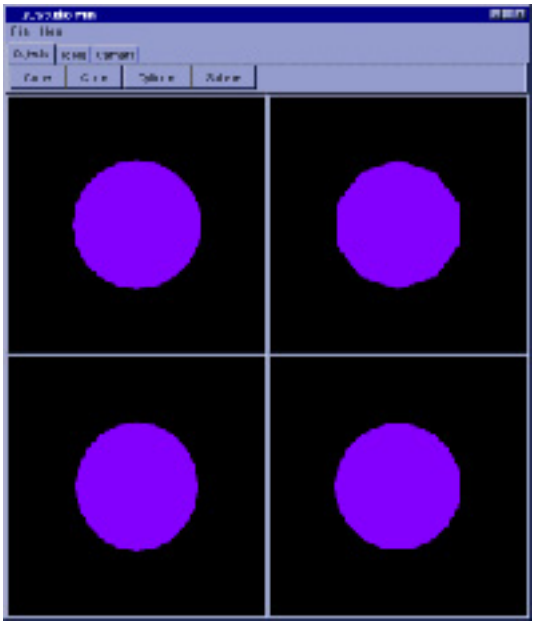
Your assignment, over the course of Programs 3 and 4, is to create a simple 3D modeler that resembles your drawing program in many ways but works in 3D rather than 2D. Like the drawing program it will feature object creation, transformations, viewing, and grouping; these features are part of this assignment. Assignment 4 will go on to add more 3D-specific features to the program. You will continue to use OpenGL for graphics, and in this assignment you will be responsible for creating more of the user interface, using Swing. Contrary to the Program 2 handout, this program does *not* build on the code you wrote for that assignment.

An example of how your modeler might look is shown in Figure 1. As long as you satisfy the requirements below, though, you can set things up however you like.

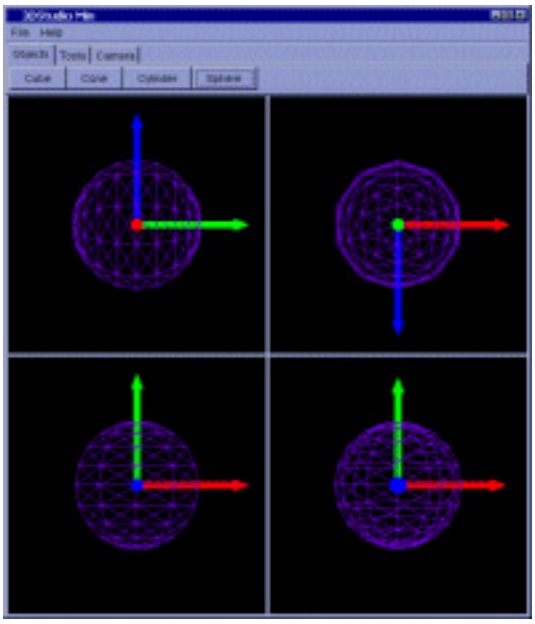
Requirements

To be specific, here is what your program needs to do:

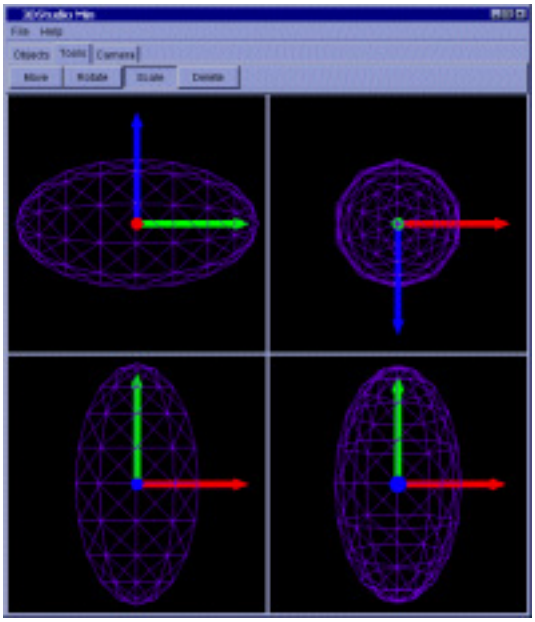
- The GUI (implemented using Swing) should consist of:
  - Four viewports that show front, top, and side orthographic views and one arbitrary perspective view of the scene you are modeling.
  - A "Main" tab for commonly used actions such as object transformations. For this assignment, selecting this tab will reveal a toolbar of buttons to select among the different types of object transformations that are available.
  - An "Objects" tab for object creation. Selecting this tab will reveal a toolbar of buttons to create the object types that are supported.
  - A "Viewing" tab for camera transformations. Selecting this tab will display a toolbar listing the camera transformations that are supported.
- You need to support creation of four basic objects: cubes, spheres, cones, and cylinders.



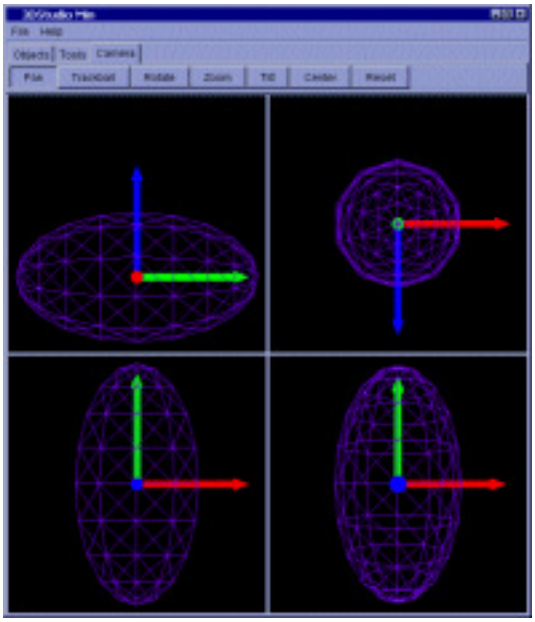
1. Create a sphere



2. Select the sphere



3. Apply a scale



4. Pan one of the cameras

Figure 1: Example screen shots of the finished program.

  - *Object creation:* Clicking on one of the toolbar buttons creates a new object of the appropriate type in canonical position. Each object type should be as large as fits in a 2x2x2 cube centered at the origin. That is,
    * The sphere has radius 1 and center at the origin.
    * The cube has corners at $(\pm 1, \pm 1, \pm 1)$.
    * The cylinder has radius 1 and its end faces are in the planes $y = \pm 1$.
    * The cone's base is at $y = -1$ with radius 1, and its apex is at $(0, 1, 0)$.
  - *Color:* Every new object created will have a random color (so that you can tell them apart).
  - *Generating triangles:* Use OpenGL primitives to create objects. Don't use the GLU convenience functions to generate triangles for you. For example, you are not permitted to use `gluSphere()` to create a sphere; instead you will have to write the code to approximate the sphere using triangles. Similar tesselation should be done for the other primitives as well. Draw using triangle strips and triangle fans when appropriate (for instance, a cone can be drawn with two triangle fans). Note that it is a little tricky getting the sphere tesselation correct.

    Your program should have reasonable performance, in that it should not slow down noticeably unless the user creates a very large number of objects. To this end you should compute the vertex positions only when necessary, and you should avoid excessively fine tesselation.

- You should allow objects to be selected by clicking on them. Multiple objects can be selected by shift-clicking. When an object or objects are selected:
  - A world-space coordinate system should be displayed on the screen with the object. The coordinate system should consist of 3 mutually perpendicular arrows representing the x, y and z axes, colored red, green, and blue respectively. The axes should be large enough to click on comfortably (use thick lines or draw small cylinders).
  - Selected objects should be displayed in wireframe to help you see the axes and to indicate that they are selected.
  - When multiple objects are selected, they should behave as if they are temporarily grouped together. That is, you should just display one set of axes, and the objects will transform together. (This is a departure from the behavior of the previous assignment.)
  - The selected object or objects can be deleted.
  - The selected object or objects can be grouped together, resulting in a single object that can be transformed as a group.
  - If the selected objects are groups, they can be ungrouped. Ungrouping must not result in any changes to the objects or their positions.

- You should allow a selected object to be transformed via translation, uniform and nonuniform scaling, and rotation.
  - *UI for transformations:* The user chooses a particular type of transformation by selecting it from the "Main" toolbar.

    Rotation and scaling are done using the displayed axes. When the user clicks on a particular axis, that axis is selected. The axis is displayed in yellow to indicate that it is selected.

    Subsequent mouse motions define the magnitude and sign of the transformation that will be applied to the selected object. Up and down in the screen specify positive and negative motion along the given axis. For instance, if rotation is the chosen transformation type and the x-axis is the selected axis, moving the cursor up in the screen will result in a positive rotation about the x-axis and moving the cursor down will result in a negative rotation. The magnitude of rotation will be determined by the distance the cursor has moved.

    For translation, the user just drags the mouse anywhere in the viewport. The object should move parallel to the camera plane so that its center point moves exactly as far in the image as the mouse was dragged.

    Note that when an object is translated the axes move with the object but when the object is rotated or scaled the axes stay fixed. That is, the center of the axes is defined by the center of the object, but the orientation of the axes always aligns with world coordinates.

- You should allow the user to adjust all the cameras to get the desired view. The modes that should be supplied (which are chosen from the "Viewing" toolbar) are:
  - Track: the camera moves parallel to its view plane as the user drags in the window. In the orthographic views the motion should be scaled so that the image moves exactly as far as the mouse was dragged.
  - Zoom: the camera's magnification (this is the size of the image in the orthographic case and the field of view in the perspective case) changes to enlarge or reduce the image.
  - Dolly: the camera translates along the view direction. For orthographic cameras, this has no effect on the image (other than moving the near and far clipping planes.)
  - Orbit: the camera's eye point rotates around its target point. Horizontal and vertical mouse motions are mapped directly to the spherical coordinates of the eye point. (An implementation of the basic orbiting calculation is provided in the framework, because you will return to a better version of this operation in a later assignment.)
  - Look at: set the camera's target point to the center of the selected object.
  - Reset: reset all camera parameters back to the defaults.

The last two modes respond to a click in a given viewport; you shold be able to, for instance, reset one of the views without resetting all four.

Framework code

For this project we are providing a framework very similar to the one for the previous assignment, but in 3D. It includes an implementation of picking using the object ID buffer. Note that, as in the previous assignment, the framework code is yours to modify in any way you like.

Groups

This project is to be done in groups of two.

Implementation notes

As objects are added to the scene you will create a scene graph, a hierarchical tree, to represent the scene. The structure is much like that used in the Program 2, except that the geometry and material properties of a shape are broken out into separate objects rather than being handled by subclasses of the shape class. You will probably implement grouping by making a new class that extends the `SceneNode` class. The scene graph as provided does not have links pointing up the hierarchy, but you are welcome to add them if they help you.

Note that the `vecmath` library has provisions for computing rotation matrices from axes and angles. Taking advantage of this (perhaps by exposing the functionality in the `Transform` class) could save you some work.

The framework contains an abstract class `Camera`, which represents the camera's position in space using the eye, target, up (also known as from, at, up) mechanism discussed in class (the camera is positioned at the eye point; its view direction is toward the target point; and the vertical axis of the image is defined by the up vector). Most of the manipulations on the camera can be done just by setting these points. The framework contains an implementation of orbiting the eye point around the target point, which also provides an example of building a coordinate frame and using it to carry out transformations in a more convenient way.

A subclass `PerspCamera` of `Camera` represents perspective cameras, and you will want to define another subclass for orthographic cameras.

Extra credit

As always, we encourage you to add extra features for extra credit. The ground rules are

that (a) a program that scores below 95/100 on the basic requirements cannot earn any extra credit and (b) no program can have a score greater then 115/100. Here is one suggestion that would be worth approximately 5 points out of 100 if implemented well:

- Add the option of transforming objects using manipulators that appear in the scene. This is the logical generalization of the resizing handles you used in the draw program: you would draw handles around the shape in 3D, and dragging the handles would cause various constrained rotations and scales that make the selected handle follow the mouse. SGI's OpenInventor system is a source for examples of manipulators; this page shows some pictures:
  `http://www.tgs.com/support/oiv_doc/DemosTools/Demos/sceneviewer/manips.htm`