CS 418 Program 4: Model II

(revised April 7, 2003)

out: Friday, March 28, 2003
due: Monday, April 14, 2003

In this assignment you will extend the modeler that you wrote in Program 3, adding features that will help you when you add a ray tracing renderer in the final assignment of the semester. The new additions are

- Saving and loading models using a file format we define. This will allow you to work on your models in multiple sessions, and since the file format is standard it will mean you can share models with one another. Your program will be a much more useful modeling tool for the ray tracing assignment.
- Triangle meshes. This provides access to many models from other sources, and also will finally let you begin to test the performance limits of the graphics cards in the lab.
- Generalized cylinders. This type of object is based on two curves in 2D, which we'll call the transverse curve $\mathbf{f}(s)$ and the radial curve $\mathbf{g}(t)$. Together they define a parametric surface:

  $$\mathbf{p}(s,t) = \begin{bmatrix} f_2(s)g_1(t) \\ g_2(t) \\ f_1(s)g_1(t) \end{bmatrix}$$

  If the transverse curve is a circle and the radial curve is a line segment, the resulting surface is a cylinder. When the curves differ from this, the radial curve varies the radius along the cylinder, and the transverse curve defines the cross-section of the cylinder. This is why these shapes are called generalized cylinders, or GCs.

  In your program the radial curve will be defined by a Bézier spline and the transverse curve is either a circle or a Bézier spline. The shapes that are formed using a circle as the transverse curve are called surfaces or revolution and are good for rotationally symmetric objects like vases, drinking glasses, and turned table legs. With a general cross section you can model a wider variety of different shapes.
- Lighting and materials. You will use OpenGL's lighting features to allow for point light sources and diffuse and specular shading on objects.

Requirements

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

- File format:
  - You should provide a standard "Open" menu command to read files. The loaded scene should replace whatever is currently present.
  - Your program must be able to read any file that uses the tags defined by the parser as it is provided. You can find several test files on the web site, but during grading we will use different test files.
  - If you are implementing extra features and want to define new tags in the file format, that is OK. However, you must be very careful that your changes are reverse compatible so that you can still read any file that can be read by the basic implementation. *If you extend the file format in such a way that it breaks the ability to read the standard format, that will be considered an error in your implementation for the purposes of grading.*
  - The file format has name references, so for instance you can define a triangle mesh and tag it with the name "bunny", then later on incorporate it into the model by including a reference to the name "bunny". The intended purpose is to make it convenient to include large triangle meshes in a scene while keeping the file for the scene small and comprehensible. Although the name reference mechanism is general and can be used to create multiple references to any object, you are only required to support multiple references to geometry.
- Triangle meshes:
  - A triangle mesh, consisting of a list of vertex positions and a list of vertex indices for the triangles, should be supported as an object type. Meshes can be transformed just like any other object, but they can't be edited or otherwise manipulated—just drawn.
  - Your code for dealing with triangle meshes should be reasonably efficient. To this end you should draw them using display lists.
  - The only way for a triangle mesh to get into your program is via the file format, so you will need to implement the parser first.
  - The file format supports the storage of vertex normals, and if they are there you should use them when drawing. But since they are optional in the files you will have to recognize when they are not there and compute them yourself; see implementation notes below.
- Generalized cylinders:

  These two object types are based on 2D Bézier curves.

  - When you initially create a GC (using a button in the same way as the simple shapes) the transverse curve is the unit circle and the radial curve is a line segment from $(1, -1)$ to $(1, 1)$, resulting in a cylinder (minus the end caps).
  - When the cylinder is selected, an editing window should appear that is split into two spline editing areas, one for the transverse curve and one for the radial curve. In each window, as soon as the user has entered enough points to define a segment of curve, the spline replaces the default circle or line segment. Either spline is allowed to be open or closed. When you edit the splines, the GC in the 3D windows should update in real time as you move the control points.
  - In the transverse window you should draw something to indicate the origin, and in the radial window you should draw the y axis.
  - For triangle generation, the splines should be sampled adaptively (using recursive refinement is a convenient way to do this) until they meet the flatness criteria mentioned below under tessellation adjustment.
  - You may (and should) reuse your spline editing code from Program 2.

- Lighting and materials:

  For this assignment, you should enable lighting in OpenGL so that objects are drawn with shading. You should set up your program to support multiple material types (that is, multiple types of shading), and you should implement just two: Lambertian (diffuse only) and Phong (diffuse plus specular).

  - Provide a reasonable interface to let the user set material properties. One good possibility for the UI is to have a "Material Inspector" window that is brought up using a menu item. This window has a pop-up menu at the top to choose the material type (Lambertian or Phong) and an area below where one or three sliders appears depending on which material type is chosen. When an object is selected the inspector window updates itself to show the current setting for that object, and then the user can use the controls in the window to change the material properties.
  - In order for shading to be useful there have to be lights. You should allow up to 8 point light sources, each of which has a position and a color. Provide a reasonable user interface to creating, deleting, and moving lights and setting their colors. One recommendation for a way to do this is to have a toggling menu item called "Show lights," and when it is turned on draw the locations of the lights as a set of small spheres that can be selected and translated using the usual transformation interface. For setting the colors of lights you might provide a light inspector that works like the material inspector.
  - For lighting to work you need to provide surface normals. These normals should be computed from the exact geometry (not from the triangles that approximate it) for all object types other than the triangle mesh. For example, the normals

to the sphere should always point exactly away from the origin. For triangle meshes the normals have to be computed from the triangles, of course.

- You will want to make a couple of default lights so that objects are visible before the user defines any lights. A reasonable choice might be one at $(10, 0, 0)$ with unit intensity and one at $(-10, 0, 0)$ with intensity 0.5.

- Other features:
  - Duplicate: you should be able to duplicate an object, creating another copy that can be transformed to move it to a new location.
  - Tessellation adjustment: you should provide a global adjustment that controls how finely objects are tessellated. It should be in the form of the maximum allowed angle between two adjacent faces on a smooth surface; for instance, if the tessellation parameter is set to 5 degrees, then a cylinder should be drawn with 72 facets. This parameter should affect the sphere, cylinder, cone, and GC, but not the triangle mesh. For the GC, it should control the sampling density on both splines.
  - In general, your program should be fast enough that it does not slow down noticeably when the user is just moving the camera and objects, even if the objects include fairly large triangle meshes or fairly complicated generalized cylinders. Of course it will be possible to slow the program down with a large enough scene, but we expect you to come within an order of magnitude of the capabilities of the systems in the lab.

Framework code

For this project we are providing a parser that should make it fairly easy to integrate the file format into your system. The parser is designed to interact with the rest of your code through a set of interfaces, one for each type of object that can be encoded in a file. The basic idea is that you create a `Parser` object, point it to implementations of all the interfaces that it needs (e.g. tell it that `Cube` is the class that should be used to represent cubes), then ask it to read a file. As it reads it will create instances of the classes you gave it and set their properties using accessor methods (`setColor`, `setTransform`, etc.) that are defined in the interfaces.

More details are available on the Web site.

Groups

This project is to be done in groups of two. In cases where groups have been re-formed because of students who dropped the class, any or all of the code that was part of either partner's previous assignments is fair game to build on.

Implementation notes

To support lighting you will need to compute vertex normals for all the geometry. The lighting requirements ask you to use the exact normals for all geometry other than the triangle mesh. For the original shapes this is pretty straightforward. For the spline-based shapes, you should compute the surface normal based on the tangents to the two splines.

For triangle meshes the normal to the underlying geometry is not available, and you will have to compute vertex normals by averaging the face normals of the adjacent triangles. This is easy to do in one pass through the triangle list, in which you add each triangle's face normal to the normals for its three vertices, followed by a normalization pass through the vertex normals.

More advice on various details will be forthcoming in section.

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 are some suggestions that would be worth between 2 and 5 points out of 100 if implemented well:

- Trackball rotation interface. Implement the popular single-mode interface to 3D rotation known as the virtual trackball, which will let you rotate the camera or an object around all three axes, rather than just one (the current rotation interface) or two (the current camera orbit mode) at a time. Implement it both as an additional camera motion mode (similar to camera orbiting) and also as an additional means of transforming objects.

  Since this used to be a required feature, there is a helpful handout on the web site that explains the geometry behind the virtual trackball.

- Provide a means to cap the ends of GCs when the radial spline doesn't end on the axis. Making this work in general requires tesselating an arbitrary polygon.

- Provide the option to draw triangle meshes using vertex arrays rather than display lists, and implement a frame-rate display that shows you how many frames per second you are able to render. This will let you compare the performance of the two approaches; on most systems vertex arrays are expected to be faster.

- Add an "Import" or "Merge" command that works like the "Open" command but adds the new objects to the scene rather than replacing the scene. This is easy and would be very handy in creating more interesting scenes.

- Improve the spline editing UI:

- Allow the user to introduce corners into the curve by selectively removing continuity constratints between segments. You will need to think about these corners when you generate normals for the resulting shape.
- Optionally constrain the endpoints of the radial curve to lie on the y axis.
- Optionally further constrain the endpoint tangents of the radial curve to be perpendicular to the y axis.