CS 465 Program 4: Modeler

out: 28 October 2005
due: 15 November 2005

1 Introduction

In this assignment you will work on a 3D modeling system that uses simple primitives and curved surfaces organized in a transformation hierarchy. The framework provides the user with orthographic and perspective views along with tools for editing shapes, transformations, lights, and the camera.

The window layout, camera control, a tree-style hierarchy view, a 2D spline editor, 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, evaluates splines and curved surfaces, 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, surfaces of revolution, 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 is split into four viewports that view the scene with three perpendicular orthographic views and one perspective view. Each viewport has a camera associated with it, and by clicking and dragging in a viewport with various modifier keys you can move the associated camera. The orthographic cameras always maintain their view directions, but the perspective camera is fully adjustable.

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 viewports, and by dragging the parts of the manipulator around in those views 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 final leaf node type is a surface of revolution built from a Bézier spline, which can be edited using the interactive spline editor. The spline editor maintains a list of spline segments and displays them along with the control polygon and handles at the control points. The user can grab these handles and move them to edit the spline, and the editor enforces appropriate continuity conditions between adjacent segments.

The program has a standard event-based structure, where an event generated by the user causes a change to the data structures and then all the viewports are redrawn. Drawing in the viewports 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 three general areas are mesh generation, splines, and manipulators.

3.1 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, spheres, and spline revolutions. In addition to the vertex positions and triangles connecting them, these meshes contain vertex normals for smooth shading and texture coordinates for texture mapping. The density of the meshes is determined by a global flatness tolerance that the user can adjust to trade off accuracy for speed. This tolerance is to be interpreted according to the flatness criterion defined below for the spline.

All primitives (except the revolution, whose size is determined by the spline) are sized to exactly

fill the cube $[-1,1]^3$. Texture coordinates (u,v) are always in the unit square $[0,1]\times[0,1]$, and the texture should be right-reading (rather than mirrored) when viewed from the outside of the object.

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. The texture coordinates should have u going around the cylinder counterclockwise (as seen from a +y vantage point) starting and ending at the $+x$ axis; v should run from bottom to top. The texture coordinates on the end caps should cause the circular area inscribed in the unit square in (u,v) to appear on each end.

2. *Sphere.* The sphere is a unit sphere centered at the origin. The normals should make the whole sphere look smooth, and the texture coordinates should be latitude and longitude with the directions arranged like the cylinder, with u running from the x axis eastward around the equator and v running from the $y=-1$ pole to the $y=1$ pole.

3. *Spline revolution.* The surface of revolution corresponding to a 2D curve $c(t)$ is a parametric surface defined by:

$$\begin{aligned} s_x(u,v) &= c_x(v)\cos(2\pi u) \\ s_y(u,v) &= c_y(v) \\ s_z(u,v) &= -c_x(v)\sin(2\pi u) \end{aligned}$$

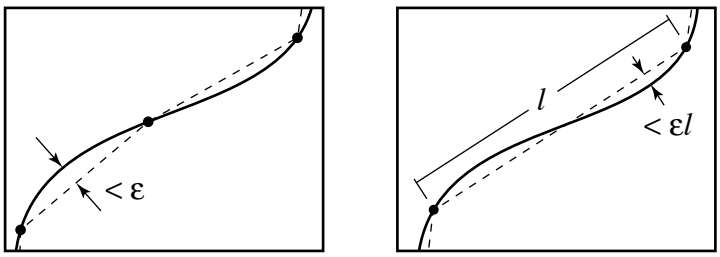
   where (c_x, c_y) are the coordinates of the 2D spline curve. The mesh should be built so that the surface normals face outward and the surface is oriented outward when $c_x > 0$ and c_y is increasing (the default initial position of the spline has the surface facing outward). The normals should be the exact normals to the spline surface. The texture coordinates of the surface are defined so that the u values are simply the t values of the spline curve and the v values run around the shape in the same way as the cylinder and sphere.

3.2 Splines

The framework includes a Bézier spline editor, but is missing the code to implement the spline itself. You have to write code to divide a Bézier segment into two pieces (this is used by the spline editor to allow the user to add extra control points) and to generate a set of adaptively spaced points to approximate the curve. The generated points are used for two different purposes: in the editor, to display the curve in the window, and in the modeler, to generate a mesh to display the surface of revolution. These two purposes have different error criteria: for 2D drawing it's important for the polygon that's drawn to stay close to the true curve, whereas for 3D rendering it's important for the surface normals to stay close to the true normals. For this reason there are two different criteria for deciding when there are enough points:

1. *Distance.* The polygon defined by the points must lie entirely within a distance ϵ of the spline curve.

2. *Flatness.* Each segment of the polygon defined by the points must lie entirely within a distance ϵl of the spline curve, where l is the length of the segment.



distance criterion flatness criterion

Figure 1: The two curve approximation criteria.

There should be a lower bound on the t spacing of the generated points, to prevent the subdivision getting out of hand in some pathalogical configuration.

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.

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 area 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 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, texture coordinates 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 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:

$$x0, y0, z0, x1, y1, z1, x2, y2, z2, ...$$

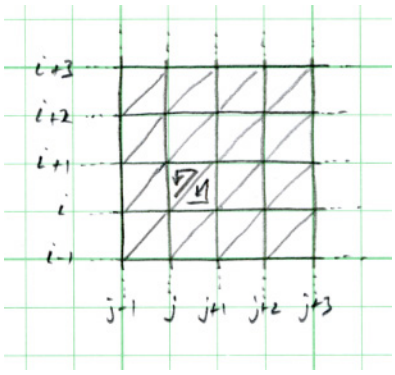


Figure 2: Diagram for surface triangulation.

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 $pi/(4n)$. [Steve please check.] The `getPoints()` spline method should be used to get a list of spline points obeying the current tolerance.

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. Finally, you will need to replicate vertices along the any line where a texture would wrap around to itself. For example, if you wrap a texture around a cylinder, at some point the left and right edges will meet. The triangles that connect these texture edges must have replicated vertices in order for the texture coordinates near the end to make sense.

You can build your mesh as a (u, v) grid for all the surfaces of revolution (cylinders, spheres, spline revolutions). The diagram in Figure 2 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 spline the adaptively spaced v steps are computed by the spline evaluation code. For the sphere, regular subdivision in v is fine; for the cylinder only a one-row grid is needed. You only need to evaluate the spline once, and then you can live of that set of 2D points to generate all the 3D points.

4.2 Splines

Spline segments are are represented by the `BezierSpline` class. A complete spline is one or more BezierSpline objects forming a linked list. Each segment stores references to its control points, and the end control points are shared between neighbors. In addition, each spline stores some information about its control points. First, it stores two booleans indicating whether the first and last endpoints are corners. If a point is a corner, the spline editor will allow the tangents before and after the point to differ, but not otherwise. Second, the segment stores its starting and ending t values. The t values for the entire spline run from 0 to 1, and when a segment is subdivided, its t range gets divide between the new smaller segments. Finally, each segment stores pointers to its left and right neighboring segments.

You must implement the methods `divide` and `getPoints` in the BezierSpline class. The first

method, `divide`, takes an input t value and returns two new segments representing the left and right parts of the original segment. For convenience in specifing the spliting point, the input t value is relative to the min and max values of the current spline. Thus an input t value of 0.5 should split the segment halfway between its min and max t. The return segments should point to each other, have a single new control point shared between them, and have the appropriate new global t value at the split point. See Shirley Chapter 15 specifically Section 15.6. You can tell if it works by using the editor to split a segment: if the curve does not move and a new junction is introduced exactly where you clicked, it is working.

The `getPoints` method generates an ordered list of points that are on the curve, optionally along with their tangents and t values. The placement of the points along the curve should be adaptive, computed by recursively splitting each spline in the list into smaller splines until the selected stopping criterion is satisfied:

One step in the recursive subdivision process consists of examining a segment to see if it's well enough approximated by the line segment between its two endpoints. If it is, we add a point to the list; if not we split the segment in two and repeat the same procedure on each half. You should limit the recursion depth to 8, so that no more than 256 points are ever generated for one segment. Take care to ensure that each point is only output once, and that the spline goes all the way to the ends.

Both of the termination criteria can be bounded using the spline's convex hull property. Your program should guarantee that the termination criterion is met by the output points.

The method will take as input three lists to be filled with points, tangents and t values. If any list is null, you do not need to generate that piece of data for call (the points list will never be null). The last two parameters are first a constant that defines the stopping criterion as one of the above and the current error tolerance.

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

We recommend talking to us about your proposed extra credit first so we can steer you towards interesting mappings and make sure we agree that it would be worth extra credit.

Some ideas: Double cross section splines; spline curve extruded along an axis; trackball rotation; rotate manip follows mouse; add more shapes.

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