CS 465 Program 4: Modeller

out: 30 October 2004
due: 16 November 2004

1 Introduction

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

The basic framework including the window layout, a tree-style hierarchy view, a spline editor, type-in transformation editors, and an orthographic camera is provided. You will write the code that draws all the shapes in the scene using the OpenGL API, evaluates splines and curved surfaces, edits transformations interactively, and implements the perspective camera. You'll also demonstrate your finished product by creating a model of a human figure with hierarchical joints.

2 Principle of operation

This section gives an overview of how the modeler works; there are more details in the following sections.

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 modelling 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 four cameras exist—one for each viewport—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 left pane, which you use to select nodes in the hierarchy. You can use menu commands to create new nodes, which will appear as children of the selected node, and to bring up a dialog that allows you to change properties of the object. You can also use cut and paste to move nodes around.

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 interacting with the viewports. In a way that depends on the type of transformation, this will change the transformation being applied by that node. You can also move the perspective camera by selecting it and dragging in the orthographic views, or you can move a light by selecting it and dragging in the orthographic views.

The available leaf nodes 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 scaling and translation transformations that apply to the sphere. The only editable property of these objects is the color.

The final leaf node type is a surface of revolution built from a Bézier spline. The rotation is always around the y axis, with other axes achieved by applying translations and rotations to the object. The editing dialog for the spline surface includes an 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.

All operations with the spline are based on adaptively spaced lists of points, which are generated by recursively splitting the spline. There are no explicit polynomial evaluations anywhere in the code! The fundamental spline operation, on which all others are built, is splitting a segment into two parts.

The architecture of the program has the usual 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 objects are called to draw themselves (in their own local coordinates) using OpenGL calls.

3 Requirements

There are several core pieces of the program that need to be implemented before the framework becomes a functional modelling program. The four general areas are splines, OpenGL drawing, editing transformations, and the perspective camera.

1. *Splines.* The framework includes a spline editor, but the spline object itself, defined in the class `shape.BezierSegment`, is not implemented. You must implement the methods `divide` and `getPoints`.

   The `divide` method returns a pair of spline segments, which together should describe exactly the same curve as the original segment. The junction between the two segments should occur at the specified t value along the original segment. See Shirley 13.2 and 13.3. 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 parameter values. It should do this using recursive subdivision,

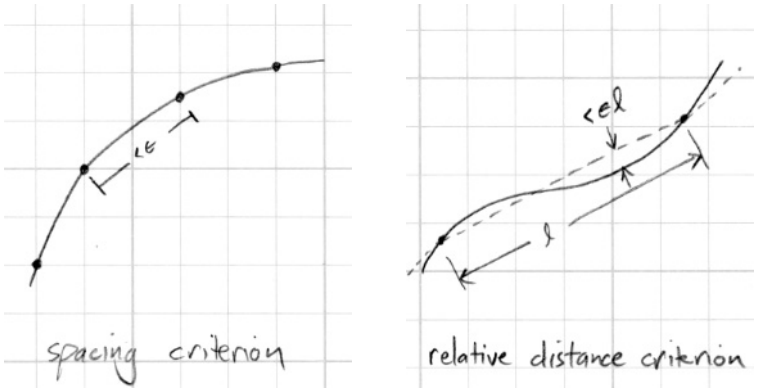

Figure 1: The two subdivision criteria.

splitting using `divide` with $t = 0.5$ at each level. You should implement three different termination criteria (see figure), each using a tolerance ϵ that is given by the caller:

(a) *Uniform spacing.* The recursion should terminate when the distance between adjacent points is less than ϵ. This option is used to generate a set of points to test when finding the closest point on the spline to where the user clicked.

(b) *Flatness.* The recursion should terminate when the spline's convex hull property guarantees that the segment of spline under consideration is within ϵl of the polygon defined by the points, where l is the distance from the first to the last endpoint. This option, which limits the angle between adjacent segments, is used to tessellate the surface of revolution.

One step in the recursive subdivision process consists of examining a spline segment to see if it's well enough approximated by the line segment between its two endpoints. If it is we write out a point; if not we split the segment in two and repeat the same procedure on each half. The flatness criterion should terminate when distance from the line joining the two endpoints to the farther of the middle two control points is less than ϵ times the distance between the endpoints.

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.

2. *OpenGL drawing.* For each of the shape nodes, you need to implement the `drawShape` method, which is responsible for making OpenGL calls to send triangles into the graphics pipeline. Your code should compute vertices and vertex normals and arrange them into an efficient set of triangle strips. You then draw the strips using `glBegin`, `glEnd`, and the appropriate `glVertex*` and `glNormal*` variants. You're also responsible for setting the color. The implementation for `Cube` is provided, but note that the cube differs from the other shapes in that all its edges are meant to be sharp, rather than smooth.

   Here are some notes on the individual shapes:

   (a) *Cylinder.* Draw the side of the cylinder with a single triangle strip, and use a triangle fan per end cap.

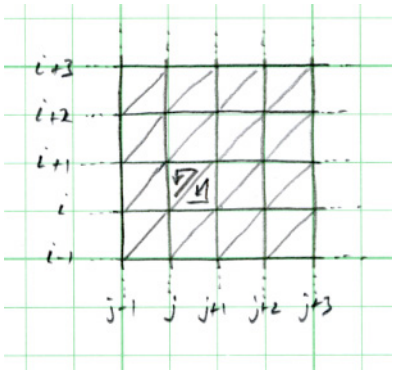

Figure 2: Diagram for surface triangulation.

(b) *Sphere.* Draw the sphere using a latitude-longitude grid, using the obvious triangle strips. The topological part of this code can be very similar to the code for surfaces of revolution (after all, a sphere is a surface of revolution.)

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

$$s_x(u,v) = c_x(v)cos(2\pi u)$$
$$s_y(u,v) = c_y(v)$$
$$s_z(u,v) = -c_x(v)sin(2\pi u)$$

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 surface should be tessellated finely enough to satisfy the relative distance criterion along the u and v directions.

You should build your mesh as a grid that's regularly spaced in u and spaced according to the points generated by `getPoints` in v. The diagram in Figure 2 illustrates this idea. 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.

You can ensure your u spacing satisfies the relative distance criterion by ensuring that the angle in radians between adjacent points is less than 8ϵ.

3. *Inserting and removing transformations.* The framework provides a command to add a new transformation (with no children) as a child of the selected node, to insert a new transformation as a parent of the selected node, and to eliminate a transformation by pushing it down to its children. The first two operations are trivial tree manipulations and are fully implemented by the framework. Eliminating a transformation should not affect the positions of anything in the scene. This requires updating some transformations, and you need to implement that. Be sure to use `TreeModel#insertNodeInto()` and `TreeModel#removeNodeFromParent()` to add and remove children nodes from `DefaultMutableTreeNode`s. If the node being collapsed has any `Shape` objects as its children, and is not an identity transform, then the current `Transformation` should be replicated under its parent, with the `Shape` as the sole child of the newly replicated node.

4. *Editing transformations.* The framework provides a means for editing transformations by typing in numbers. This is often 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. You should implement interactive editing for translations, rotations, and scales (but not general transformations). When the user drags the mouse in one of the orthographic views while a transformation is selected, the framework calls the transformation's `update` method with an offset that is derived from the distance between the last mouse event and the current one. The correct behavior depends on the type of transformation:

(a) *Translation.* When editing a translation, the translation vector should be updated so that the objects under the translation follow the mouse—that is, if you move the mouse to the right by 5 pixels, all the affected objects should move to the right be 5 pixels. This should be true no matter what transformations are above the selected one.

(b) *Rotation.* When editing a rotation, vertical motion of the mouse should translate to changing the angle of the rotation.

(c) *Scaling.* When editing a scale, dragging up and down with no modifier keys should result in uniform scaling; upward motion scales larger, and downward motion scales smaller. Dragging all the way from the bottom to the top of the viewport should scale by a factor of 10. Dragging with the ALT modifier key should apply a nonuniform scale: right/left motion should scale along the axis that is closest to horizontal in the viewport, and up/down motion should scale along the axis that is closest to vertical. Again, dragging the full width or height of the viewport should scale by a factor of 10.

5. *Perspective camera.* The orthographic camera is fully implemented in the framework, so you can see objects in the orthographic viewports and you can move the orthographic cameras around. For an orthographic camera, motion along the view direction is mostly meaningless, and for these cameras we want the view directions to stay fixed. So the only so the only available motions are track (move camera perpendicular to view direction) and zoom (change size of view rectangle). The framework allows zooming by SHIFT-dragging and tracking by CTRL-dragging. Note that when you track the camera, objects in the view move exactly as far as the mouse does.

   The framework does provide a means of editing the perspective camera, too: if you select the perspective camera, then you can edit it by dragging in the orthographic views (to alter the camera's eye point and target point together) or by ALT-dragging (to alter the camera's target point only). This way you can test out the perspective camera before you have normal camera motion working.

   The perspective camera should be movable using the following operations:

   (a) *Orbit.* (by simply dragging) Move the camera around on a sphere around its target point. The camera should continue to aim toward the same target point. Vertical dragging causes a rotation around a horizontal axis, while horizontal dragging causes rotation around a vertical axis.

   (b) *Zoom.* (by SHIFT-dragging) Change the camera's angle of view. Dragging all the way from the bottom to the top of the window should change the image magnification by a factor of 2.

   (c) *Track.* (by CTRL-dragging) Move the camera perpendicular to the view direction in such a way that an object at the same distance as the target point would stay under the mouse cursor.

(d) *Pan.* (by ALT-dragging) Change the view direction while keeping the camera fixed. As with orbiting, Vertical dragging causes a rotation around a horizontal axis, while horizontal dragging causes rotation around a vertical axis.

(e) *Dolly.* (by CTRL-SHIFT-dragging) Move the camera along the view direction, toward the target point for upward motion, and away for downwards motion.

4 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.

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