CS348a: Computer Graphics Handout # 7
Geometric Modeling
Stanford University Tuesday, 12 February 2002

Homework #3: B-splines; rational curves; surfaces; programming in OpenGL; trackball implementation [135 points]
Due Date: Tuesday, 26 February 2002

Theory and implementation

This is a mixed theory/implementation homework. Problems 3 and 4 in this assignment ask you to develop the theory of and implement in OpenGL the viewing of a NURBed torus. Problems 4 and 5 are the programming part and, as explained in the guidelines given with Homework 1, you may work on this programming part in groups of up to three students and hand in a joint write-up. You must still hand in individual solutions to problems 1, 2 and 3.

Problem 1. [15 points]

Find the periodic, uniform, C^2-continuous, cubic, polynomial spline curve with four segments that interpolates the four points $(1,0)$, $(0,1)$, $(-1,0)$, and $(0,-1)$ in the plane. That is, your spline curve F should consist of four cubic polynomial segments $F([0\mathinner{..}1])$, $F([1\mathinner{..}2])$, $F([2\mathinner{..}3])$, and $F([3\mathinner{..}4])$. It should satisfy $F(0)=(1,0)$, $F(1)=(0,1)$, $F(2)=(-1,0)$, $F(3)=(0,-1)$, and $F(t)=F(t+4)$ for all t. And it should have C^2 continuity at all four joints. Draw a picture that shows the four de Boor points of the spline F and the four Bézier points of each of its four cubic segments.

Find the Cartesian coordinates of the point $F(1/2)$, one of the two points where your spline curve cuts the line $X=Y$. Is the point $F(1/2)$ inside, on, or outside of the unit circle?

Problem 2. [20 points]

We define a quadratic, rational curve segment $F([0\mathinner{..}1])$ in the plane by specifying its three Bézier sites $(w;x,y)$ as follows:

$$\begin{aligned} f(0,0) &= (1;-1,-1)\\ f(0,1) &= (0;c,0)\\ f(1,1) &= (1;1,1), \end{aligned}$$

where c is a nonzero, real parameter. Note that the corresponding Bézier points are the point $(-1,-1)$, the point at infinity in the horizontal direction, and the point $(1,1)$. Give a formula for the polar value $f(t_1,t_2)$ and a formula for the diagonal value $F(t)$. Find the Cartesian coordinates of the points $F(1/2)$ and $F(\infty)$. Find the slope of the curve F at the points $F(1/2)$ and $F(\infty)$. Sketch the curve F when $c=1/2$ and when $c=3$.

Your sketches should include the four points $F(0)$, $F(1/2)$, $F(1)$, and $F(\infty)$, as well as the four corresponding tangent lines.

The curve traced out by F is a conic section — in fact, an ellipse. Find the implicit equation of that ellipse. (Hint: Your formula for $F(t)$ will have the form $F(t) = (w(t); x(t), y(t))$ for quadratic polynomials w, x, and y. Find the implicit equation relating the Cartesian coordinates $X = x/w$ and $Y = y/w$ by eliminating t from the two quadratic equations $w(t)X - x(t) = 0$ and $w(t)Y - y(t) = 0$.)

Problem 3. [20 points]

Let r and R be real numbers with $0 < r < R$. As the angles α and β vary, the varying point $V = (X, Y, Z)$ given by

$$\begin{aligned} X &:= (R + r\cos\alpha)\cos\beta \\ Y &:= (R + r\cos\alpha)\sin\beta \\ Z &:= r\sin\alpha \end{aligned}$$

traces out a *torus*, that is, the surface of an ideal bagel. The torus has rotational symmetry about the Z axis, which passes through the middle of the torus hole. Any plane π through the Z axis cuts the torus in two circles of radius r, whose centers lie along the line where π cuts the XY plane, R units on each side of the origin. Varying α moves the point V around one of those circles. Varying β rotates the plane π around the Z axis.

Find a biquadratic, rational parameterization of this torus. That is, express each of the homogeneous coordinates $[w; x, y, z]$ of the varying point V as a polynomial in two parameters that has degree at most 2 in each parameter when the other is held fixed. For consistency in notation, use Q and T as your two parameters, writing

$$V(Q; T) = [w(Q; T); x(Q; T), y(Q; T), z(Q; T)]$$

for certain polynomials w, x, y, and z. Hint: Let $Q := \tan(\alpha/2)$ and $T := \tan(\beta/2)$.

Homogenize and polarize your parameterization. You may carry out these two steps in either order; the result will be the same. If you homogenize first, use p as the weight coordinate for the Q parameter, writing $Q = q/p$, and use s as the weight coordinate for the T parameter, writing $T = t/s$. If you polarize first, split the T parameter into two separate parameters T_1 and T_2, and split the Q parameter into Q_1 and Q_2. The homogenized polar form v of V will have the form

$$v((p_1; q_1), (p_2; q_2); (s_1; t_1), (s_2; t_2)) = [w; x, y, z],$$

where each coordinate w, x, y, and z is a polynomial in the eight variables p_1, q_1, p_2, q_2, s_1, t_1, s_2, and t_2.

One quarter of the torus consists of points that have Y and Z positive. (If you followed the hint above, those points will correspond, under your parameterization, to parameter pairs $(Q; T)$ where both Q and T are positive.) Describe this portion of the torus as a biquadratic, rational Bézier surface patch, that is, as a rectangular, tensor-product patch of degree $(2; 2)$. In particular, give the coordinates of the nine Bézier sites of this patch. (If you followed the hint above, the patch will be $V([0 \mathinner{..} \infty] \times [0 \mathinner{..} \infty])$.) Hint: Don't be distressed if one of the nine Bézier sites turns out to be the zero site, that is, the site all four of whose coordinates are zero.

Problem 4. [30 points]

In the previous problem, you developed a representation of a torus by using a splined surface consisting of four biquadratic, rational Bézier surface patches. In this problem you will implement a slightly more general toroidal surface, using the facilities of the OpenGL graphics library on Linux, SGI, Windows, or possibly other workstations. In addition to modeling the torus, you will use the OpenGL API to render an image of the torus on the screen and to set up a rudimentary user interface for playing with the parameters defining this generalized torus, as well as controlling the viewer's position relative to the torus. Unlike the previous theoretical problems, in this problem the emphasis is on implementing things that you already know in the context of a widely available (and very nice to use) graphics platform.

In this problem you must use the C programming language. OpenGL will give you access to a set of graphics libraries that you can use to implement modeling and rendering operations. In particular, OpenGL provides very good NURB support, so to specify your spline you need do nothing more than calculate the appropriate Bézier control sites. But since your goal is to produce and interact with an image, you must also become familiar with some of the OpenGL facilities dealing with rendering. Specifically, you will need to understand how to set up lights illuminating your model, how to define the material properties of the surface of your model, how to specify the viewpoint, and how to use z-buffering (the depth buffer) for hidden-surface elimination and double-buffering to create smooth animations. OpenGL provides excellent facilities for dealing with all these issues — each of them will require only a few additional lines of code in your program. Nevertheless, especially if you are unfamiliar with basic graphics concepts, you may want to spend a couple of hours just browsing through the rendering chapters of any standard graphics text. The OpenGL manual itself is a pretty good reference — the course reader contains the NURBS section of the older GL manual (the precursor to OpenGL). Numerous books exist that describe OpenGL in detail; a few of these are in the recommended book list handed out earlier. A more precise description of what you will need to use is given later in this handout.

In order to interact with your image, you will build a set of interactors using OpenGL and a `forms` package provided for you. You will not need anything but menus and sliders, the latter for inputting real parameters to your program. While your program is running you should be able to interactively modify the parameters defining the generalized torus, as well as the position of the view point, and immediately see the results. Documentation on the forms package is contained in the course reader. We will be using a version of `forms` that has support for OpenGL canvasses, and the forms manual will be available in the class directory and in the web page of the course.

Specifying a generalized torus

As in Problem 3, we consider the torus given by the equations

$$\begin{aligned} X &:= (R + r\cos\alpha)\cos\beta \\ Y &:= (R + r\cos\alpha)\sin\beta \\ Z &:= r\sin\alpha, \end{aligned}$$

where $0 < r < R$ and α and β are free parameters. As you discovered in Problem 3, we can decompose the surface of that torus into four rational, biquadratic surface patches, according to the signs of Y and Z. The middle Bézier site of each of the resulting four patches turns out to be the zero site — the site whose four coordinates are all 0.

For this problem, we will generalize that torus in two ways. First, we remove the restriction that $r < R$; we allow both of the radii r and R to be arbitrary positive numbers. Second, we consider replacing the middle Bézier site — which, for the true torus, is the zero site — with the mid-point of the surface patch, scaled by some multiplier m. For example, consider the patch in which Y and Z are both positive. The mid-point of that patch corresponds to the parameter values $\alpha = \beta = 90$ and has the coordinates $(X, Y, Z) = (0, R, r)$. We place the middle Bézier site of that patch at the site $(w; x, y, z) = (m; 0, mR, mr)$, for some real number m.

Note that choosing $r > R$ leads to a torus that intersects itself at a cone-like singular point on the Z axis. Note also that choosing any value of m different from 0 leads to a surface that is not a torus, even though the boundary curves of each patch don't move. In fact, the resulting surface doesn't even have tangent continuity.

Viewing the torus

Your torus will be illuminated by two lights placed in the scene. Please see the header file `/usr/class/cs348a/source/pp1/pp1.h` for the position and characteristics that you should use for the light sources. The torus itself will be made up of a material whose characteristics are also defined in that header file. Please use this header file as part of your code.

You should open and paint two windows: a main window, where the torus itself is displayed, and a secondary smaller window, where the coordinate axes, the torus, and the viewpoint are all displayed. The latter window is there to help you in understanding how to move the viewpoint around the torus.

Interacting with the view

You need to implement two kinds of interactions. First of all, your users should be able to modify any of the free parameters r, R, and m defining the torus. They will do this by moving sliders corresponding to the free parameters. They should also be allowed to change the viewpoint by moving three additional sliders corresponding to the coordinates of the viewpoint in the natural (X, Y, Z) coordinate frame of the torus.

Sample program

In the directory `/usr/class/cs348a/source/pp1` there is a sample program called `pp1.c` that implements part of the functionality of the program that you need to write. In this sample program, a sphere is visualized instead of the torus. We suggest that you modify this code, including the information to create and visualize the torus. An interface defined using forms can be found at the file `pp1_ui.fd`. Just type `fdesign pp1_ui` to see what the interface looks like, and use `fdesign` to modify the interface as desired. The output of `fdesign` is a C program with appropriate calls to the forms library. The forms library can be found in Linux machines at Sweet Hall in the directory `/usr/class/cs348a/source/xforms`

OpenGL has a special way to tesselate NURBS, and in order to have reasonable performance, change the following properties of the NURB object you create in OpenGL in the following way:

```
gluNurbsProperty(theNurb,GLU_SAMPLING_METHOD, GLU_DOMAIN_DISTANCE);
gluNurbsProperty(theNurb,GLU_U_STEP,15);
gluNurbsProperty(theNurb,GLU_V_STEP,15);
```

These properties control how fine the tesselation will be. A performance penalty will be added if the step values are increased, and the default values tesselate the nurbs in great detail.

Problem 5. [50 points]

In problem 4 above (visualizing a NURBed torus), you used three sliders to set the position of the viewpoint. In this problem, we ask you to implement a more intuitive view specification method, called the *virtual trackball.* By implementing this interaction technique you will

- experience a practical application of quaternions,
- implement one of the most popular navigation metaphors,
- taste the harsh reality of coding mouse-driven interaction,

The virtual trackball metaphor was first suggested by Chen, Mountford, and Sellen in their SIGGRAPH ’88 paper [CMS88], and subsequently extended by Gavin Bell. The implementation we propose is a slight modification of Bell’s work, put together by João Comba, a former CS348a TA, and his fellow Ph.D. student Apostolos Lerios a few years back.

From mouse input to viewpoint rotation

The next three subsections discuss how user-initiated mouse movement is mapped onto viewpoint rotation.

Step 1: Normalized mouse coordinates

The `forms` library reports to our code the mouse position using window coordinates; these vary from $(0, 0)$ (top-left corner of the window)[1] to (w, h) where w and h are the window width and height, in pixels. In order to design a GUI (graphical user interface) with friendly HCI (human-computer interaction), we would like to make the view specification system independent of window size. So, we map window coordinates along the shortest of the two axes (horizontal or vertical) to the range $(-1, 1)$; that is, if $h < w$, a vertical window coordinate of 0 maps to $y = 1$ and h maps to $y = -1$. The longer axis is then mapped to the range $(-a, a)$ where a is chosen so that the transformation does not distort angles ($a = w/h$ if $h < w$; see figure 1). We now have a pair of real numbers, the *normalized mouse coordinates*, representing the mouse position relative to the window's center, in terms independent of window size.

Step 2: Mapping 2D points on the window plane to 3D

We then map each 2D point on the window plane (the xy plane) onto a 3D point: we define the z axis so that it comes out of the screen and points towards the viewer. We would like this mapping to generate points on a virtual trackball, centered at the window's center (see figure 1). Then, as the user moves the mouse, we will be rotating that virtual trackball; by conceptually gluing the trackball onto the scene, the trackball rotation will then rotate the viewpoint.

To map a 2D point onto a 3D point on the trackball, we can use

$$z = \sqrt{r^2 - x^2 - y^2}$$

where r is the trackball radius. The problem with this approach is that we either have to use a huge trackball (with radius r large enough to cover the whole screen[2]), or we get an imaginary z for points such that $r^2 < x^2 + y^2$. Using a huge trackball creates a bad GUI since the user has to move the mouse *a lot* to generate a small rotation: an empirically pleasant value for r is 0.8.

To fix the problem of imaginary z, we map every 2D point onto a surface which is a sphere near the origin (for accurate rotation) and smoothly turns into a circularly-symmetric hyperbola away from the origin. Figure 2 shows the cross-section of the

[1] Real-life warning: the *X Window System* puts the origin at the top-left corner of the window, but *OpenGL* puts the origin at the bottom-left corner. Keep your origins straight!

[2] "You mean, the whole *window*?" No, the whole *screen*: `forms` reports mouse movement to the application even if the mouse is not in the canvas, provided the user clicks-and-drags; in this case, we receive window coordinates outside the $(0, 0)$ to (w, h) range.

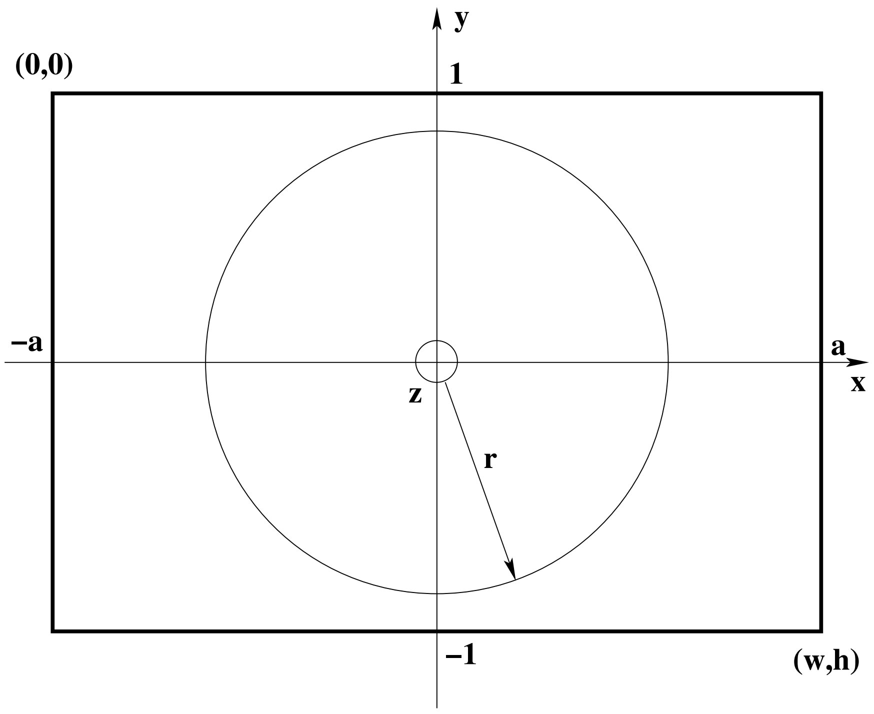

Figure 1: The virtual trackball and the axes of normalized mouse coordinates: $(0,0)$ and (w,h) are window coordinates, while -1, 1, $-a$, and a mark normalized mouse coordinates.

resulting surface, and here is the equation defining it:

$$z = \begin{cases} \sqrt{r^2 - x^2 - y^2} & \text{if } \sqrt{x^2+y^2} < \frac{r}{\sqrt{2}} \\ \frac{r^2}{2\sqrt{x^2+y^2}} & \text{otherwise} \end{cases}$$

Note that the resulting surface is continuous in position as well as derivative.

Step 3: Mapping pairs of 3D points to a rotation

When the user presses down mouse button 1 over the OpenGL canvas, we just record the mouse position $\vec{p_1}$ (a 3D point, calculated as described above). As soon as the mouse is dragged to a new position $\vec{p_2}$ (while the same button is held pressed), we incrementally rotate the virtual trackball and thus the viewpoint. As new mouse positions arrive, we keep track of the most recent position as well as the current one, and continually apply incremental rotations to the trackball. But how do we compute these incremental rotations?

Imagine you had a real trackball in front of you, put your finger on it, and moved it a little bit from $\vec{p_1}$ to $\vec{p_2}$. It's easy to see that the net result is that you rotated the trackball about the axis $\vec{\alpha}/|\vec{\alpha}|$, where $\vec{\alpha} = \vec{p_1} \times \vec{p_2}$, by an angle $\widehat{\vec{p_1}\vec{p_2}}$ (computed as the

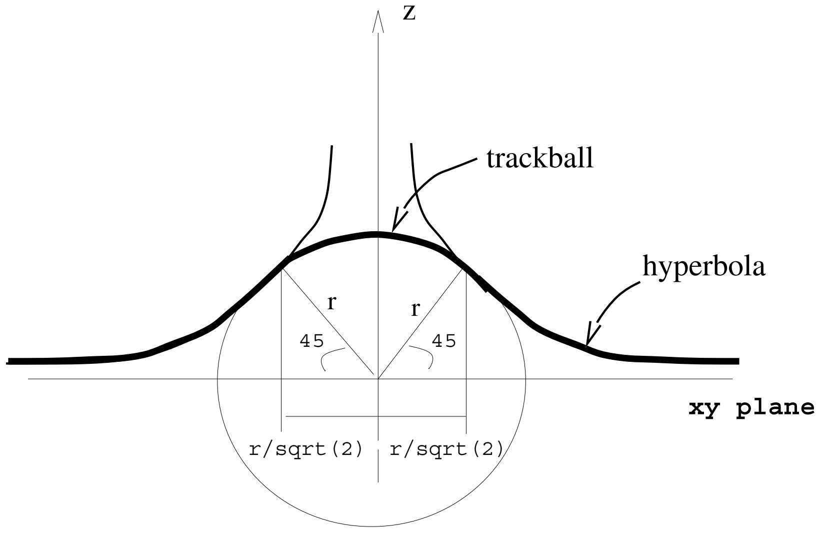

Figure 2: A cross section of the surface used to map 2D points on the window plane to 3D.

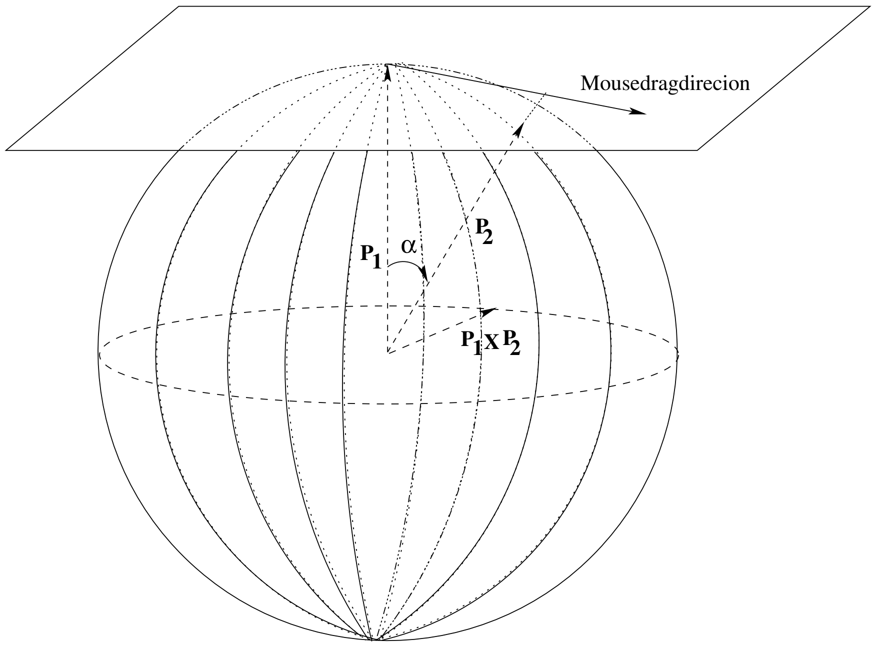

Figure 3: Mapping a 3-D point pair to a rotation.

inverse cosine of the dot product of $\vec{p_1}/|\vec{p_1}|$ and $\vec{p_2}/|\vec{p_2}|$). So, this is the rotation which we need to apply to the scene every time the user drags the mouse from $\vec{p_1}$ to $\vec{p_2}$ (see Figure 3). Piece of cake!

Coding all this

Now that the theory is out of the way, we need to address the practical issues of coding this view specification system.

Representing rotations

A naive way to represent rotations is to use rotation matrices. As the user moves the mouse, incremental rotations are generated and multiplied onto a master scene rotation matrix (initialized to the identity matrix). Sounds good but it doesn't work: in theory, multiplying two rotation matrices should yield another rotation matrix; not so in practice, due to roundoff errors in floating-point arithmetic.

So, we use quaternions, instead, to represent all rotations in our code. Not only are roundoff errors less severe this way — as fewer arithmetic operations are needed to compose two quaternions than to multiply two matrices —, but in case they do occur, we can easily recover a valid rotation by renormalizing the quaternion of the master scene rotation!

Givens

In the directory `/usr/class/cs348a/source/pp1-extra` you will find `trackball.c` which contains part of the program you need to write, including the `forms` routines that handle mouse interaction: `forms` calls the procedure `CanvasPointerMotion()` when the mouse moves over the OpenGL canvas. Routines that perform basic vector and quaternion operations have also been provided. You just need to write the code that computes the quaternion representing the incremental rotation of the trackball (procedure `TrackBall()`). In the same file, you need to write the code that computes the composition of two rotations expressed in quaternion form (procedure `ComposeQuats()`).

Finally, integrate this interface with problem 4 and use it for visualizing the torus.

What to hand in

Remember that for programming assignments you are allowed to work in teams of up to three students. To submit your code, please make an `.tar` or `.zip` archive of all your files and e-mail it to the TA. Be sure to include a `README` file that gives the names of your group members and and an index to your program files and functions. In addition, please hand in with your paper-and-pencil homework a short write-up about how your code works. We will set up a time in Sweet Hall during which you will be able to demo your program to the instructor and TA.

References

[CMS88] Michael Chen, S. Joy Mountford, and Abigail Sellen. A study in interactive 3-D rotation using 2-D control devices. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 121–129, August 1988.