Curves and Pipes

Due date: Monday, March 29, 11:59 pm

The goal of this assignment is to build a simple interactive modeller, that would allow you to create shapes called generalized cylinders, which can be thought of as pipes with variable cross-section. Generalized cylinders can be built out of curves. Your program should allow the user to define all necessary curves; it should compute and draw the generalized cylinder, and allow the user to choose the viewing direction.

The assignment has two parts: required and extra credit. You have to do only the required part, but you will receive bonus points if you do all of it.

Pipes and generalized cylinders. First, let us define a pipe – a simpler version of a generalized cylinder. Start with a curve $\mathbf{p}(t) = (x(t), y(t), z(t))$ in space. Take another curve $c(u) = (u(r), v(r))$. To distinguish between coordinates in the plane and in space I will use u and v for coordinates in the plane and (x, y, z) for coordinates in space. We use $c(r)$ as the cross-section of the pipe. The pipe itself is obtained if we drag the cross-section along the curve $\mathbf{p}(t)$ in space. To give a more precise definition, consider for each point of the curve $\mathbf{p}(t)$ the unit tangent vector $\tau(t)$ to the curve, and the unit unit normal vector $\mathbf{n}(t)$ to the curve; then define $\mathbf{b}(t)$ as the vector product $\tau(t) \times \mathbf{n}(t)$.[1] The surface of the pipe is defined parametrically as

$$P(t, r) = \mathbf{p}(t) + u(r)\mathbf{n}(t) + v(r)\mathbf{b}(t)$$

A generalized cylinder is slightly more complicated; instead of using a fixed cross-section, we allow the cross-section to vary; the initial curve $c(u)$ is used as a template, which can be deformed as we move along the curve $p(t)$. We will consider only two simple ways to deform the cross-section: scale and rotation. To define how the cross-section is scaled, we define how the scale factors in two directions change with the parameter t: $s = (s_u(t), s_v(t))$. To define the rotation of the cross-section, we define how the angle of rotation changes with t: $\theta = \theta(t)$. To get the formula for the surface, we apply the scale and the rotation to $c(t)$; at the point $p(t)$, the cross-section should be

$$u^t(r) = \cos\theta(t) s^u(t) u(r) - \sin\theta(t) s^v(t) v(r); \quad v^t(r) = \sin\theta(t) s^u(t) u(r) + \cos\theta(t) s^v(t) v(r)$$

.

The equation for the surface of the generalized cylinder is

$$P(t, r) = \mathbf{p}(t) + v^t(r)\mathbf{n}(t) + u^t(r)\mathbf{b}(t) \tag{1}$$

[1] (In fact, we will not use the precise normal to the curve, but a vector close enough to the normal for our purposes.

Implementing generalized cylinders. As you can see from the formulas above, a generalized cylinder is defined by a set of curves: $\mathbf{p}(t)$, $c(r)$, $s^u(t)$, $s^v(t)$, $\theta(t)$. Each of the curves can be specified as a B-spline. Note that $\mathbf{p}(t)$ is a curve in space, so it has 3D control points, $c(r)$ is a curve in the plane, so it has 2D control points, and, finally, the control points of 1d curves $s^u(t)$, $s^v(t)$ and $\theta(t)$ are simply numbers. However, the algorithm that one can use to compute the curves is exactly the same in all cases, and coincides with the one that we have discussed in class for the curves on the plane.

To describe the generalized cylinder as a B-spline, we specify control points (3D, 2D or 1D) for each of the curves. First, we choose the numbers of control points for the curves $\mathbf{p}(t)$ and $c(r)$. The numbers of control points for other curves are the same as for $\mathbf{p}(t)$. Suppose the first number is N and the second number is M; the control points are

- for $\mathbf{p}(t)$, $\mathbf{p}_0, \mathbf{p}_1, \ldots \mathbf{p}_{N-1}$;
- for $c(r)$, $c_0, c_1, \ldots c_{M-1}$;
- for $s^u(t)$, $s^u_0, s^u_1, \ldots s^u_{N-1}$;
- for $s^v(t)$, $s^v_0, s^v_1, \ldots s^v_{N-1}$;
- for $\theta(t)$, $\theta_0, \theta_1, \ldots \theta_{N-1}$.

Suppose all the control points are specified; how do we draw the generalized cylinder? The idea is to make lots of small quadrilaterals approximating the cylinder, and draw these quadrilaterals. Thus, we need to compute the vertices of the quadrilaterals. This is done in several steps.

First, we obtain a sufficiently close approximation to the curves described above; this is done by applying the subdivision algorithm for cubic splines that we have discussed in class several times (3 should be sufficient); the output of the algorithm is the new sets of control points which we will denote $\mathbf{p}^j_0, \mathbf{p}^j_1 \ldots \mathbf{p}^j_i, \ldots$, $c^j_0, c^j_1, \ldots c^j_k, \ldots$ etc. The numbers of control points after j subdivision steps will be $2^j(N-1)+1$ and $2^j(M-1)+1$. Next, for each control point $\mathbf{p}^j_i$ we compute the unit normal $\mathbf{n}_i$ and $\mathbf{b}_i$. This is done using the formulas that are listed below. Next, we compute a 2d array of vertices of the quadrilaterals that we are going to draw. We get formulas for the vertices if we replace curves with their control points in formula (1).

$$v_{ik} = \mathbf{p}^j_i + (\cos\theta^j_i [s^u]^j_i u^j_k - \sin\theta^j_i [s^v]^j_i v^j_k)\mathbf{n}^j_i + (\sin\theta^j_i [s^u]^j_i u^j_k + \cos\theta^j_i [s^v]^j_i v^j_k)\mathbf{b}^j_i,$$

where $i = 0 \ldots 2^j(N-1)$,$k = 0 \ldots 2^j(M-1)$

Finally, we simply draw $(N-1) \times (M-1)$ quadrilaterals with vertices $v_{ik}, v_{i\,k+1}, v_{i+1\,k+1}, v_{i+1\,k}$, $i = 0 \ldots 2^j(N-1)$, $k = 0 \ldots 2^j(M-1)$.

Tangents and approximate normals for splines. At a control point $\mathbf{p}^j_i$ (on any level of subdivision!) the tangent can be computed as $\mathbf{p}^j_{i+1} - \mathbf{p}^j_{i-1}$. For the last and the first control points, where either $\mathbf{p}^j_{i-1}$ or $\mathbf{p}^j_{i+1}$ are missing, we compute the tangents as $\mathbf{p}^j_1 - \mathbf{p}^j_0$ and $\mathbf{p}^j_n - \mathbf{p}^j_{n-1}$ respectively, if n is the number of the last point. The approximate normals are similar: for the point $\mathbf{p}_i^j$, the normal is $\mathbf{p}_{i+1}^j - 2\mathbf{p}_i^j + \mathbf{p}_{i-1}^j$. (these are in fact second derivatives of the parameterization of the curve, not true normals, but this is good enough for us). For the endpoints, we arbitrarily choose the normal to coincide with the normals at the adjacent internal control points, e.g. the normal at $\mathbf{p}_0^j$ is the same as at $\mathbf{p}_1^j$: $\mathbf{p}_2^j - 2\mathbf{p}_1^j + \mathbf{p}_0^j$. Sometimes, the normals obtained using these formulas are zero (for example, if all control points are on the same line). In this case, simply use the normal at a previous control point which is not zero. If all are zeroes, then use an arbitrary direction in space, not parallel to the tangent. Note that the tangents and approximate normals computed in this do not necessarily have unit length. You have to normalize them! Finally to get $\mathbf{b}_i^j$, compute the cross product of the *normalized* vectors τ_i^j and $\mathbf{n}_i^j$.

1 What your program should do

Required part. You are required to implement only pipes, that is, your cross-section may remain constant. Your program should have three windows: one for defining the cross-section $c(r)$, one for defining the space curve $\mathbf{p}(t)$, and one for showing the resulting pipe.

Each curve is defined by picking a number of control points. When you click on the cross-section window, you add a new control point to the cross-section curve. When you click on one of the space curve windows, you add a control point to the space curve.

The cross-section window. In this window, you are defining a planar curve, so the third coordinate of the points is not important – just use a fixed value, say, 0. Use `glOrtho2D(-1,1,-1,1)` to setup the projection in this window. In this way, your cross-section curve control points will be in the 2 by 2 square centered at zero. Each time you click in this window, add a control point to the curve $c(t)$, resubdivide and redraw the curve, recompute the pipe and redraw it in the result window. DO not forget to convert the pixel coordinates to the range $[-1, 1]$; the x coordinate in the window becomes u coordinate of the cross-section, the y coordinate becomes the v coordinate.

The space curve window. Specifying positions of points in 3D is difficult, so we simplify the task by starting with a planar curve. Again, setup the projection using `glOrtho2D(-1,1,-1,1)`. This will be the view of our curve from the side. Specify the curve exactly in the same way you have specified the cross-section curve; set the third coordinate, z, to some default value, e.g. 0. After a control point is added, the user should be able to change the z coordinate in small increments (say, 0.02), by pressing the left and right arrow keys (left to decrease, right to increase). Note that the appearance of the curve in this window will not change, so you need to update the view in the results window to see the changes.

The result window. In this window, you should draw the resulting pipe. Redraw the pipe each time a control point is added, or its position changed (as it happens when you move it in z direction in the space curve window). To understand better what is going on, it is useful to draw the coordinate axes, or some planes parallel to the coordinate planes. You should be able to change the viewpoint in this window in a manner similar to the previous assignment: choose the initial position of the camera somewhere far enough in the direction (1,1,1), looking at the point (0,0,0). Using the arrow keys or the mouse, the user should be able to rotate the camera around the y axis, while looking at the same point. If you want, you may add vertical motion of the camera.

If the key "C" is pressed in the results window, all curves should be removed and the number of control points reset to zero.

Command line options. Your program should allow you to specify the number of subdivision steps that you do as a command-line option.

Extra credit. Implement generalized cylinders. To do this, you will need to add 3 more windows, one for each curve $s^u(t)$, $s^v(t)$ and $\theta(t)$. The range of θ should be $-\pi$ to π, the range of scales may be chosen to be 0..10. These windows are different from the other windows: here you cannot add the control points, as the number of control points on these curves should be exactly the same as for the curve $\mathbf{p}(t)$. Each time a control point is added in the space curve window, add control points to the scale and rotation curves, with some default values (e.g. 1 for scales and 0 for rotation); do not forget to resubdivide and redraw the curves. The second difference is that the control points for the curves $s^u(t)$, $s^v(t)$ and $\theta(t)$ are 1D, i.e. are just numbers. This means that the curves can be displayed as plots of functions. To be more specific, to draw the curve $s^v(t)$ given its approximation $[s^v]_0^j, [s^v]_1^j, [s^v]_2^j, \ldots$, we draw line segments connecting 2D points with coordinates $(0, [s^v]_0^j), (1/2^j, [s^v]_0^j) \ldots (i/2^j, [s^v]_i^j) \ldots$. Note that the first coordinate depends only on the number of the point and we cannnot control it. In the scale and rotation windows, the control points can move only up and down. In the scale window, the vertical coordinate corresponds to the scale, in the rotation window, to the angle of rotation. The horizontal coordinate in both cases corresponds to the parameter t. To change the values of the control points in these windows we can do the following: Suppose the mouse was pressed (not clicked!) at a pixel (x, y). Convert x to the range $[0..N-1]$ where N is the number of control points, and find the closest one. We are going to modify the control point with this number. If we move the mouse without releasing the button, the value is changed proportionally to the displacement of the mouse from the initial position. Do not forget to update all the windows each time the mouse motion callback is called.

Even more extra credit. If you still have spare time left, make your user interface better: add a popup menu option to move control points in the cross-section and space curve windows, instead of creating new ones.

Restrictions . Do not use `gluNurbsCurve` and related functions: you will spend more time figuring out how to use them then writing your own subdivision and rendering code.

2 What to hand in.

Your working program, and some pictures of pipes or the generalized cylinders you were able to make. Pipes are rather limited, but generalized cylinders can be used to make a large variety of objects. Use a screen capture program to make these pictures.