CS 4621 Practicum Programming Assignment 2
Splines

out: Tuesday, 12 November 2013
due: Friday, 22 November 2013

1 Introduction

In this assignment, you will create a spline curve editor. You will go on to create a 3D surface of revolution using the generated curve with proper normal and texture coordinates for each vertex of the surface. Finally, these curves will be incorporated into the scene editor from Manipulators so that you can enhance your solar system scene with texured planets and a spline-based comet.

We have implemented a 2D curve editor for you where we have provided the following functionality:

- moving a control point,
- selecting a control point,
- adding a new control point,
- deleting the selected control point,
- resetting the curve to the start configuration,
- saving and loading the spline,
- rebuilding the mesh of the surface of revolution, and
- changing the epsilon that determines the termination criterion of the curve subdivision.

You will implement the following functionality:

1. Approximating the 2D spline curve for rendering by first converting from B-spline control points to equivalent Bézier curves, then subdividing the created Bézier curve(s) using de Casteljau's algorithm.
2. Creating the geometry of the corresponding surface of revolution with correct normals and texture coordinates.
3. Updating the scene traversal code to handle time-dependent spline paths. You will demonstrate this functionality by creating an animated comet with an interesting path.

To do this assignment, you *must* fill in all of the TODOs for Manipulators, Scene, and Shaders code except when it is marked as optional. It is critical that you have a correct implementation of the required TODOs. If you were unable to have one, please come to one of the TAs' office hours so that we can help you out.

In the following section we go over the provided interface. In the subsequent three sections we describe what you need to implement for your assignment. First, we describe how to generate a set of vertices approximating your curve by converting to Bézier segments and how to build the mesh for the corresponding surface of revolution. Then, we describe how you can change that code to support closed B-spline curves. Finally, we go on to integrate all of this functionality in the solar system by writing code to move an object along a curve.

2 The Curve Editor

When you start Problem1 you will see a panel, on the left, in the starting configuration with a minimum number of control points. You can edit the 2D curve using the panel. As you drag points around you are changing the points of the B-spline. The system imposes a limit on the minimum (5) and maximum (20) number of control points.

Your job is to update the spline curve as you drag the points around. A slider on the left is the epsilon or tolerance slider for the 2D spline curve; you will approximate the curve with enough line segments to ensure the angle between adjacent segments is less than this tolerance. For smaller values of epsilon you will subdivide more times, resulting in a more accurate approximation.

Next you must create a 3D surface of revolution that corresponds to the spline curve; the axis of rotation is the left edge of the rectangle in the 2D window. The 3D surface (once it is implemented) will be shown in the right panel. As the 2D curve is edited you have to update both the curve and the surface of revolution.

Initially, the associated 3D surface of revolution is a rounded cylinder (or capsule) for 5 control points. However, this surface will change as the user manipulates/adds/deletes points on the spline curve. As the user edits the spline curve, the associated surface of revolution should automatically be updated. If your program slows down, you can uncheck the "interactive" button. In this case the surface will only be updated when the user selects the "rebuild mesh" option from the "Action" menu. You should be able to achieve interactive rates as long as you are reasonably judicious with your memory allocation operations.

2.1 Editing the spline curve

We now describe the functionality we provide for the 2D curve editor.

- **Moving Control Points:** The user is able to move control points by clicking on a control point and dragging the mouse. As the user moves a point, the surface of revolution will be interactively updated to reflect the new spline curve (unless the "Interactive" box is unchecked).
- **Selecting Control Points:** When you click on a point, that point is selected (and will be shown in red). You will see why this is useful when you add control points.
- **Adding Control Points:** When the user selects the "Add Control Point" item in the "Action" menu, a new control point will be inserted into the list of control points after the selected one (defined above). The 2D location of the new control point will be halfway between the selected control point and the next one in the sequence. If the last control point (or no point) is selected, the editor does nothing.

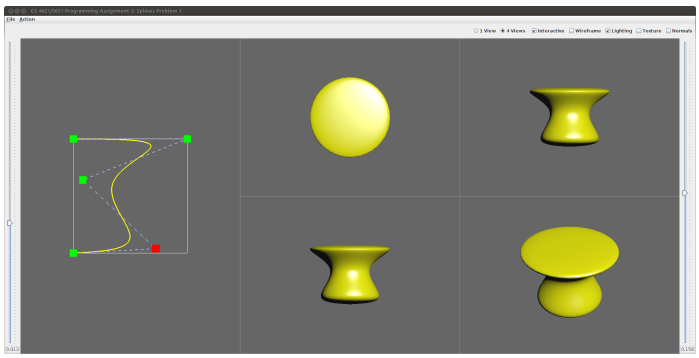

Figure 1: A reference screenshot. The red point has been "selected".

- **Deleting Control Points:** When the user selects this menu item, the selected control point will be removed. If no point is selected, the editor does nothing. In addition, the first and last control points cannot be removed in Problem 1. When control points are added/deleted/moved, the spline curve and surface of revolution should be updated appropriately.
- **Reset:** This option resets the curve and the surface of revolution to the original configuration.
- **Rebuild Mesh:** Updates the 3D mesh to correspond to the current spline curve. This is needed if the "Interactive" option is unchecked.
- **Saving and Loading:** You are also provided with the functionality to save your current curve to a file and reload it from disk.

2.2 Surface of revolution

The surface of revolution is drawn in the right panel and is updated as the user edits the curve. A surface of revolution is defined by taking a curve in the plane and rotating it about an axis: the surface swept out by the curve as it revolves is the surface of revolution. In our case the spline curve rotates about the y axis, which is the left side of the rectangle shown in the spline editor. (The desired surface is described more precisely in Section 3.4.)

For small epsilon, updating the mesh might be slow. In that case, uncheck the "Interactive" button to prevent interactive updates. To update the mesh you will then have to use the pulldown menu.

Figure 1 shows an example of the surface of revolution that could be produced by your program.

3 Problem 1: Open Curve

The first problem you will solve is that of rendering the open B-spline curve, and creating the corresponding surface of revolution.

The strategy you should use to do this is as follows:

1. Convert the B-spline control point sequence to a series of Bézier segments.
2. Compute a sequence of points on each Bézier curve using de Casteljau's algorithm.

3. Compute the tangents to the Bézier curve, and from them the normals.

4. Use these points and normals to generate the surface of revolution.

You can check your results against ours on the set of three curve files we provide in `data/curves`. We provide reference images in `data/reference` that were taken with the default GUI orientation. The file names that contain `open` are for this problem.

3.1 B-spline to Bézier construction

You should begin by implementing the `build` method in the `BSpline` class. The job of this method is to compute an approximation to the spline as a set of line segments, which will be used to draw it and to construct the mesh for the surface of revolution.

As we discussed in lecture, the B-spline curve is a series of segments, each defined by a cubic polynomial in the form

$$\mathbf{f}_{\text{Bsp}}(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M_{\text{Bsp}} P_{\text{Bsp}}$$

where P_{Bsp} is a 4-by-2 matrix whose rows are the four control points that influence the segment, and M_{Bsp} is the 4-by-4 spline matrix for the B-spline:

$$M_{\text{Bsp}} = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}.$$

On the other hand, a Bézier spline segment has the form

$$\mathbf{f}_{\text{Bez}}(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} M_{\text{Bez}} P_{\text{Bez}}$$

where

$$M_{\text{Bez}} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

Converting the B-spline control points P_{Bsp} (which are the ones the user adjusts in the editor) to the Bézier control points P_{Bez} (which you will use to approximate the curve) amounts to solving for the value of P_{Bez} that makes $\mathbf{f}_{\text{Bez}}$ equal to $\mathbf{f}_{\text{Bsp}}$.

Exercise 1. Find the matrix $M_{\text{Bsp-to-Bez}}$ so that if

$$P_{\text{Bez}} = M_{\text{Bsp-to-Bez}} P_{\text{Bsp}}$$

then $\mathbf{f}_{\text{Bez}}(u) = \mathbf{f}_{\text{Bsp}}(u)$ for all u.

You will find that this matrix has nice numbers and it is simple, once you gather the four B-spline control points for a particular segment, to neatly compute the Bézier control points.

The spline editor hands you a list of control points, numbered from 0 to $N - 1$. Finding which four control points influence a segment of the B-spline is mostly simple: segment i of the curve is influenced by control points $i - 1$, i, $i + 1$, and $i + 2$. Using this definition we can generate $N - 3$ segments from the N control points without falling off the ends of the sequence: the first segment

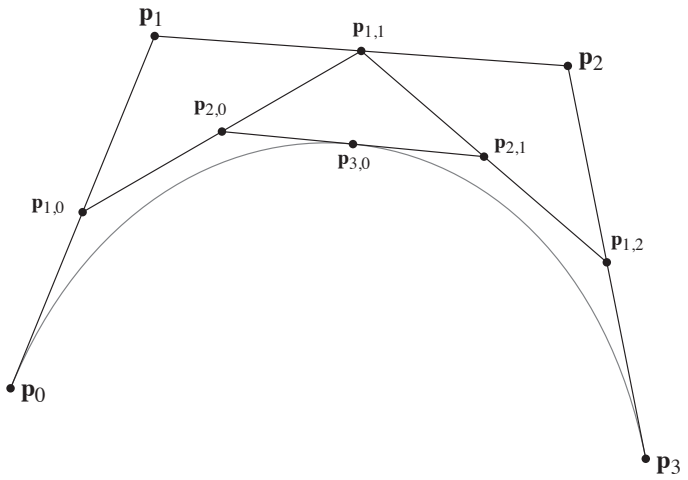

Figure 2: De Casteljau with u = 0.5

is numbered 1, and the last segment is numbered $N - 3$. We call these *interior segments*. Using only the interior segments is a perfectly reasonable way to define a curve, but for the surface of revolution it's important to be able to get the curve to start and end exactly on the axis.

There are a few ways to do this but the simplest for this assignment is to define two more "pretend" control points, numbered -1 and N, that are not actually part of the sequence but are just arranged so that segment 0 of the curve starts at control point 0 and segment $N - 2$ ends at control point $N - 1$. These extra control points are at positions defined by the actual control points, and you can figure out where they go by solving the following puzzle.

Exercise 2. Consider a B-spline segment $\mathbf{f}(u)$ that has four control points $\mathbf{p}_0, \ldots, \mathbf{p}_3$. Suppose $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ are given, and we want to position $\mathbf{p}_0$ so that the curve starts at $\mathbf{p}_1$. In other words, we want $\mathbf{f}(0) = \mathbf{p}_1$.

- Write the expression for $\mathbf{f}(0)$ as a function of the four control points.
- Find $\mathbf{p}_0$ as a function of other three control points.

With the answer to this problem in hand, you can write your code to use these extra points for the end segments, so that you have $N - 1$ segments for a curve with N control points.

We recommend you divide and conquer by first implementing your spline with just the interior segments (which will leave gaps at the ends), then adding the two end segments after that is working correctly.

3.2 De Casteljau

The motivation for converting the segments to Bézier form is the availability of a simple method for splitting Bézier segments in two. Using de Casteljau's algorithm, one can compute the set of points shown in Figure 2, ending with the point $\mathbf{p}_{3,0}$, which is a point on the cubic Bézier curve. The pattern is a simple recurrence in which points at each level are linear combinations of points at

the previous level:

$$\mathbf{p}_{k,i} = (1-u)\mathbf{p}_{k-1,i} + u\mathbf{p}_{k-1,i+1}$$
$$\mathbf{p}_{0,i} = \mathbf{p}_i$$

where the $\mathbf{p}_i$ with a single index are the control points.

But this construction not only gives you a point on the spline; it also gives you the control points of the two halves: they are $\mathbf{p}_0$, $\mathbf{p}_{1,0}$, $\mathbf{p}_{2,0}$, and $\mathbf{p}_{3,0}$ for the left half and $\mathbf{p}_{3,0}$, $\mathbf{p}_{2,1}$, $\mathbf{p}_{1,2}$, and $\mathbf{p}_3$ for the right half.

The simplest way to get the spline on the screen is to use this recurrence to compute $\mathbf{f}(u)$ (which is the point $\mathbf{p}_{3,0}$) for a sequence of regularly spaced u values. We encourage you to implement this first, to make sure you have the algorithm right. When you implement it, take a look at the method `Vector2f.interpolate`—it can make your code very simple. The problem with this approach is that it uses too many segments where the splines are fairly straight and too few where they are very curved. Using enough segments so that a tight curve (like the rim of a drinking glass being modeled as a surface of revolution) is artifact-free will cause you to use a huge number of segments everywhere.

A better way, which you should implement for your final handin, is to recursively split the segment, terminating when the control polygon is straight enough. The idea is as follows: to approximate a Bézier segment with lines, you can use a single segment if the angles between the edges of your control polygon are both less than half the angle tolerance, because the tangent to the curve cannot swing more than that much over the segment. If the angles are too big, subdivide and apply the idea recursively. The angles in the two halves will be smaller (less sharp) so ultimately the recursion will terminate for well-behaved curves. But you should also enforce a maximum on the number of levels of recursion (10 is a good choice) to prevent stack overflow in degenerate cases.

Your recursive algorithm should only emit points that are exactly on the spline (so don't add other points you may have computed that aren't on the curve).

3.3 Bézier Curve Normals

You should come up with a way to calculate the tangents to the spline for each point computed by your recursive algorithm. There are a few easy ways to do this, just pick one that is correct and makes sense to you, and explain it briefly in a code comment. From these tangents, compute 2D normal vectors that are perpendicular to the curve, which you will use in computing the 3D normals to the surface of revolution. We will evaluate the correctness of your normals by the correctness of shading and texturing, so make sure lighting works for the assignment. Use the `Normals` checkbox in the curve editor to visually debug your results.

3.4 Surface of Revolution Mesh

Mathematically, the surface of revolution corresponding to a 2D curve $\mathbf{f}(t)$ is a parametric surface defined by:

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

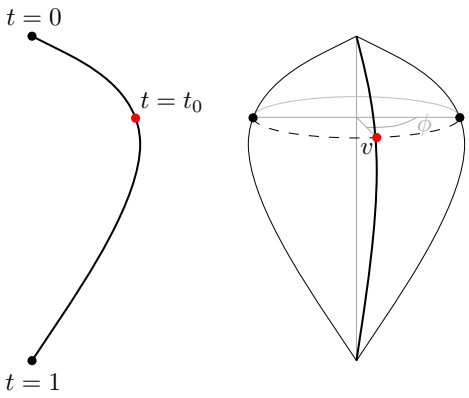

Figure 3: Assigning texture coordinates.

where $f_x(t)$ and $f_y(t)$ are the coordinates of the 2D spline curve. To display this surface in the scene you approximate it with triangles in much the same way as the sphere and cylinder (which are also surfaces of revolution, though not defined by spline curves). The mesh should be built so that the surface normal points away from the axis when $f_x > 0$ (which the spline editor enforces for you) and f_y is decreasing. (To see why it's necessary to mention which way f_y is going, think of editing the the curve to create a shape like a bowl: f_y will be *increasing* on the inside surface, and the normals will point towards the axis on the inside of the bowl. This should happen by itself if you have implemented everything the right way.) The normals should be the exact normals to the spline surface, and the triangles should be oriented so they are counterclockwise from the outside. The texture coordinates of the surface are defined by the parametric surface equation above.

Compute the positions, normals, and texture coordinates for the surface of revolution mesh by implementing the `buildMesh` method in the `RevolutionVolume` class.

Start with the vertices and normals in the `DiscreteCurve`; they are the adaptively spaced Bézier points you computed previously. Your `buildMesh` code should look similar to your sphere or cylinder code from Scene. One thing to note is that the surface of revolution mesh does not have a cap or base like the cylinder. The mesh will look closed at the ends if the endpoints of the spline are on the y axis, but as with the sphere there will be many vertices at the same position on the axis.

Besides the normals, you will also have to assign the 2D texture coordinates for the vertices of the surface of revolution.

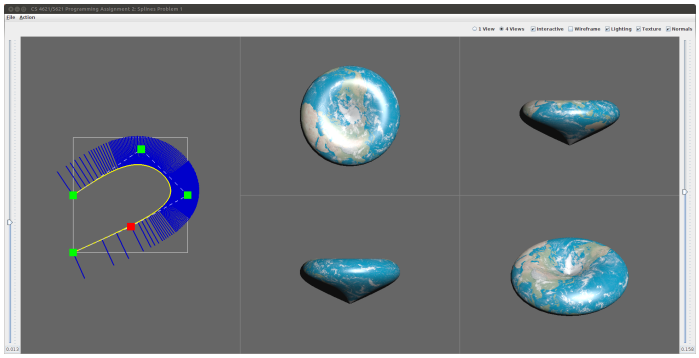

Figure 4: Textured Spline mesh

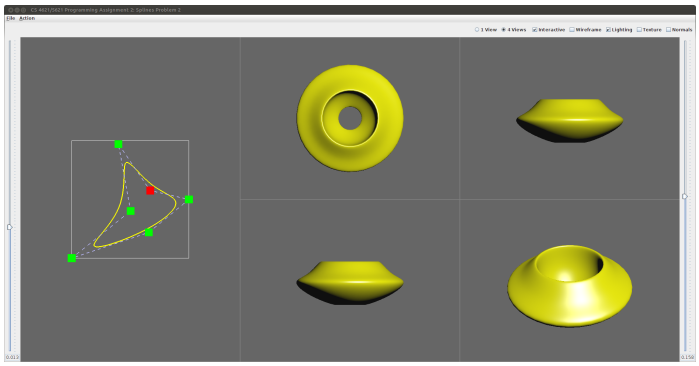

Figure 5: Closed curve mesh.

4 Problem 2: Closed Curve Splines

When modeling some kinds of objects, such as a doughnut or the One Ring, you would like a surface of revolution that is made by rotating a closed curve that is away from the axis, rather than an open curve with its endpoints on the axis (Figure 5). Our spline editor supports this in the SplinesP2 executable.

To make this mode work you have to implement the `isClosed` logical branch for the `build` method in the `BSpline` class. A closed spline is actually simpler than an open one: there are no special cases for the end segments. Instead the control point sequence "wraps around" so that control point 0 comes after control points $N - 1$. This should only involve a slight modification of your code for open splines, and indeed the result is probably simpler. It's a good idea to work things out on paper before you code this part.

5 Problem 3: Revisiting the Solar System Scene

Start by double-checking that you merged in your Manipulators code for the solar system. If you got your manipulators working it would also be wise to copy them into the appropriate files; otherwise you will be positioning the comet spline by guessing translation values. Search for TODOs with Manipulators and Scene in their text if you've forgotten where the code should go.

The time slider bar has been moved to the bottom of the problem 3 GUI. The scale is still 0 to 360; you should scale this to [0, 1] for the spline animation. Note that you will need to duplicate some of the manipulators code: the code from `SolarSystemAnimator` in `ManipP3.java` will need to be copied to `SolarSystemAnimatorP3` in `SplinesP3.java`. There are TODOs noting this.

Before you can add spline paths to the scene, you must write code to handle `SplineNode` scene objects. A `SplineNode` is an extension of a `MeshNode`; its mesh is the 2D spline path. It contains a new field named `splineOffset` which should be set to the translation from the spline's origin to the point on the spline path that corresponds to the current time.

After this work is done you must update your solar system scene file to make use of textured planets and a spline-based animated comet.

5.1 Update BSpline

We describe the method to implement the animation of the spline-based comet. We want to create an array that stores "normalized" lengths along the spline ranging from 0 to 1. Fill out the very end of `build` in the `BSpline` class to fill each `length_buffer` value with the total length of the spline from the start of the spline (the first vertex) to the vertex that corresponds to the current index. Once you have computed all of the lengths, divide all of them by the total length of the spline. The resulting array will contain sorted "normalized" lengths that range from 0 to 1.

5.2 Implement SplineNode.setTime

Fill out `setTime` in the `SplineNode` class. It should take a time value, account for the speed of the spline orbit, and fill `splineOffset` with the appropriate vertex position from the curve. There are a few steps to finding the appropriate vertex position.

First, get the `BSpline` curve from the `Spline` mesh of the `SplineNode`. Make the simplifying assumption that at time 0 the offset should match the first vertex in the spline and at time 1 it should match the last vertex in the spline; since the first and last vertices overlap this will be a complete orbit. Then use the `Arrays.binarySearch` function to find the vertex index that has the "normalized" length closest to the current time. Make sure you read the JavaDoc for this function; it doesn't do what you expect! This closest point gives you the x and y offset values; the z offset value should always be 0.

5.3 Update scene traversal

Modify the `SolarSystemAnimatorP3` code in the `SplinesP3.java` class to handle `SplineNode` objects. In short, you must translate each child of the SplineNode by the `splineOffset`.

Now you should try adding splines to the scene and see how they work. The children of the spline should follow its curve as you change the time, speeding up as you increase the spline's speed value. Note that splines are fully qualified scene objects so you can move them with the manipulators, and anything else that you can do to a `MeshNode`. We also added spline volumes to the GUI if you wish to play with them in a scene.

5.4 Improving the Solar System

To complete this part of the assignment, you must:

1. Change the materials of the Earth, Moon, and Mars to the corresponding texture materials. You should set their specular colors values to zero to remove the "shiny planet" effect.
2. Add an animated comet to the scene as follows: add a spline and form it into an artistic/interesting orbit around one of the objects in the scene (e.g., the Sun). Add an object as a child of the spline to represent a comet, you should use either a warped sphere or a spline volume. Set the material of the comet to the Comet texture material (or something else that you think looks good).
3. (Optional): Set the material of the Sun to the fire shader from PA2.

Save your final scene into the `data/scenes/splines` directory.

6 What to Submit

Submit a zip file containing your solution organized the same way as the code on CMS. Your solar system scene should be saved in the `data/scenes/splines` directory with a reasonable file name. Include a README in your zip that contains you and your partner's names and NetIDs, any problems with your solution, and anything else you want us to know.