CIS 441/541 Project G

Bezier Curves and Surfaces

March 5, 2019

Before you start

- Starter code is provided. But it requires your Project 1 code for functions: ViewTransformation(), CameraTransformation(), DeviceTransformation(), and ColorPixel() (or whatever function you use to color a pixel). You are asked to implement three new functions: Bezier_Divide(), BezierCurve.Draw(), and BezierSurface.Draw().
- Reuse CMakeLists.txt from project 1F.
- A text file teapot.txt containing points data is provides.

Getting Started

Splines allow designers to create natural looking shapes by specifying just a few control points.

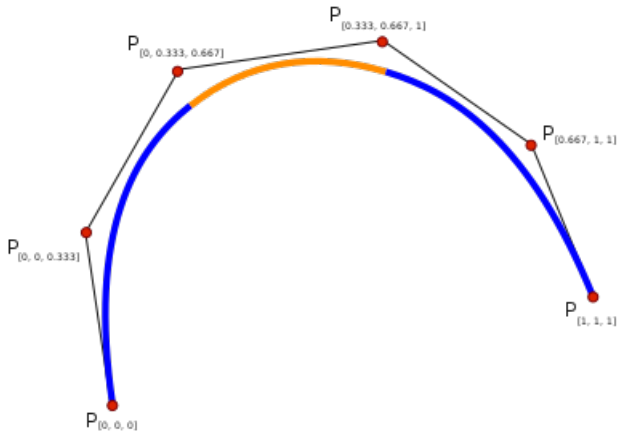

Figure 1: Splines

Bezier curves and surfaces are commonly used in computer graphics. We will implement them to produce several images including the classic teapot.

We say a curve is of order n if it has $n+1$ control points labeled from $k = 0$ to $k = n$. In this project, we will implement Bezier curves of order 3.

The de Casteljau Algorithm defines a recursive procedure on the control points:

$$p_i^l = (1-u)p_p i^{l-1} + up_{i+1}^{l-1}.$$

The level of the control points is given by $l \in [1, n]$, where n is the order of the curve. The initial control points are given by p_i^0, where $i \in [0, n]$.The point $p_0^n(u)$ is the point on the curve at parameter value u. The level 0 control points are the original control points. The level 1 control points are defined as weighted sums of the level 0 control points, and so on. Note that the parameter value u specifies the weight.

We will be using parameter value $u = \frac{1}{2}$. Then for a given set of starting points: p_0^0, p_1^0, p_1^0, p_1^0, we compute three points for level 1: p_0^1, p_1^1, p_1^1, two points for level 2: p_0^2 and p_1^2, and one point for level 3: p_0^3. We can them use them to define two sub-curves, specified by q and r.

As shown in the figure 3, a curve with 4 points will be divided into two sub-curves sharing a point, which is 7 points in total. Those points are:

$$\begin{aligned}
q_0 &= p_0^0 = p_0 \\
q_1 &= p_0^1 = \frac{1}{2}(p_0 + p_1) \\
q_2 &= p_0^2 = \frac{1}{2}q_1 + \frac{1}{4}(p_1 + p_2) \\
q_3 &= p_0^3 = \frac{1}{2}(q_2 + r_1) \\
r_0 &= q_3 \\
r_1 &= p_1^2 = \frac{1}{2}r_2 + \frac{1}{4}(p_1 + p_2) \\
r_2 &= p_2^1 = \frac{1}{2}(p_2 + p_3) \\
r_3 &= p_3^0 = p_3
\end{aligned} \tag{0.1}$$

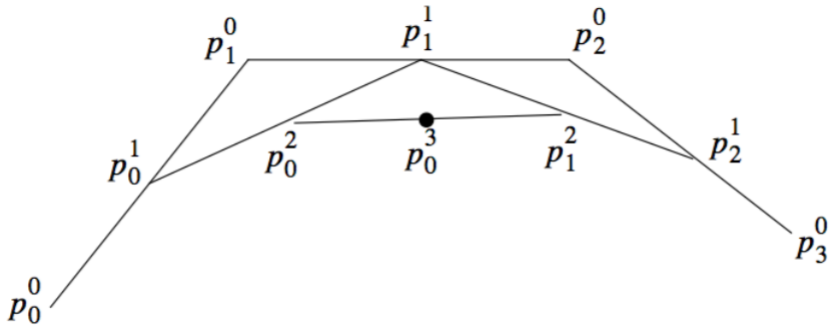

Figure 2: Eample of order 3 with u=$\frac{1}{2}$

Above is the essential dividing procedure you'll need to implement first in this project. Bezier_Divide() function is for this purpose. It takes 9 arrays of size 4: pX[4], pY[4], pZ[4] are the points to start with, qX[4], qY[4], qZ[4] and rX[4], rY[4], rZ[4] are the results. The order of points is important.

Then you will need to implement BezierCurve.Draw() method, which takes a Screen pointer as parameter and draws the curve on the screen. It should first check if the curve is small enough. If so, it creates 3 line segments in bewteen 4 points and draw the lines. If not, it divides the curve into two and calls Draw() on each of them. Note that Line class and its Draw() function are already defined for you. At this point, you should be able to use getLeaves() method and draw the curves. Notice Camera has been set for you. You might need to transform your curves before drawing them, just like what you did for triangles. Try to iterate 100 times to make 100 images and then conbine them into a gif. (ImageMagick is really handy.)

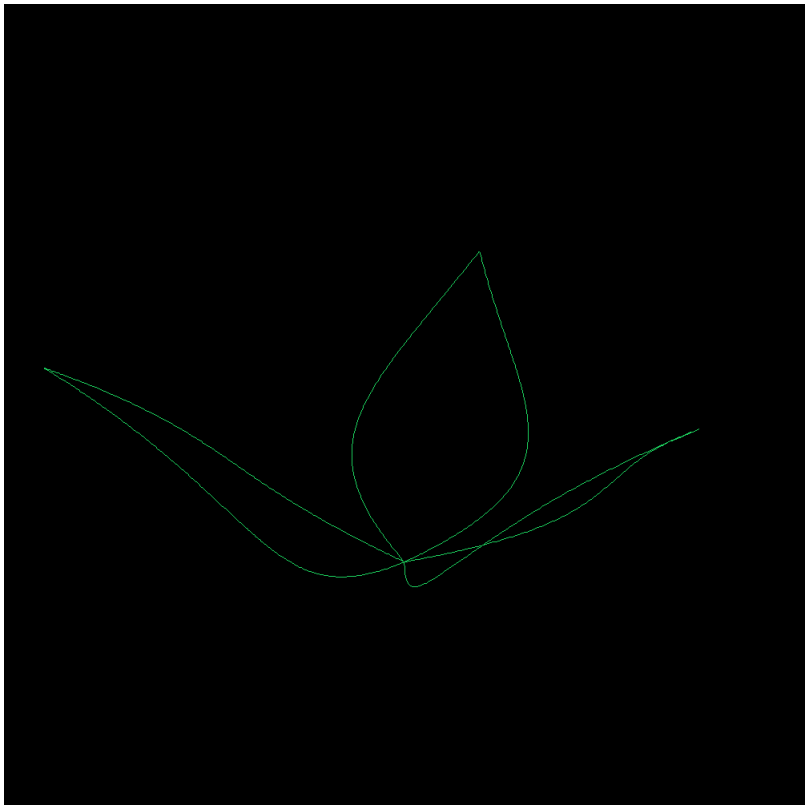

Figure 3: Test image for Bezier curves

Now let's get to Bezier surfaces. A Bezier surface is shown in Figure4. Naturally, if you divide a curve into 2 sub-curves, you want to divide a surface into 4 sub-surfaces. Meaning there are 49 points total. Some of them are shared by sub-surfaces. But how to do the division exactly? First, we will do the de Casteljau dividing algorithm on one dimension first, for example, rows. This procedure gives us 28 points, 7 on each row. Then we will apply the dividing procedure on those points in the other direction, by taking 4 points from each column.

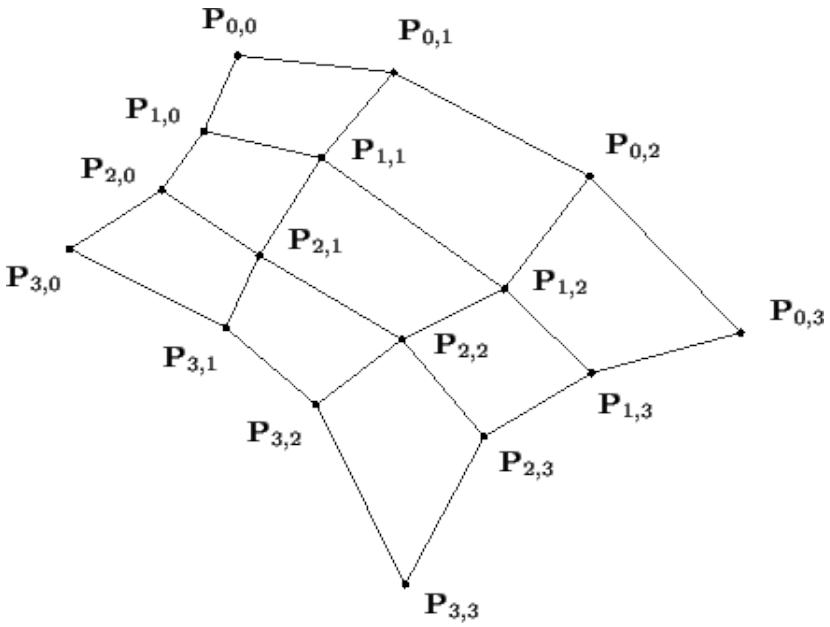

Figure 4: Bezier surface

Since it's a bit time-consuming to calculate if the surface is small enough to draw, we take another parameter divisions in our BezierSurface.Draw() method, additional to the Screen pointer. When the parameter is not 0, we perform the division and call the Draw() method on each of four sub-surfaces with parameter divisions-1. And when we draw our self-defied surfaces, we specify how many times we want to divide. Usually 3-5 is good. Now you should be able to call getSurfaces() to draw the following image. As before, you need a transformation from world coordinates to screen coordinates.

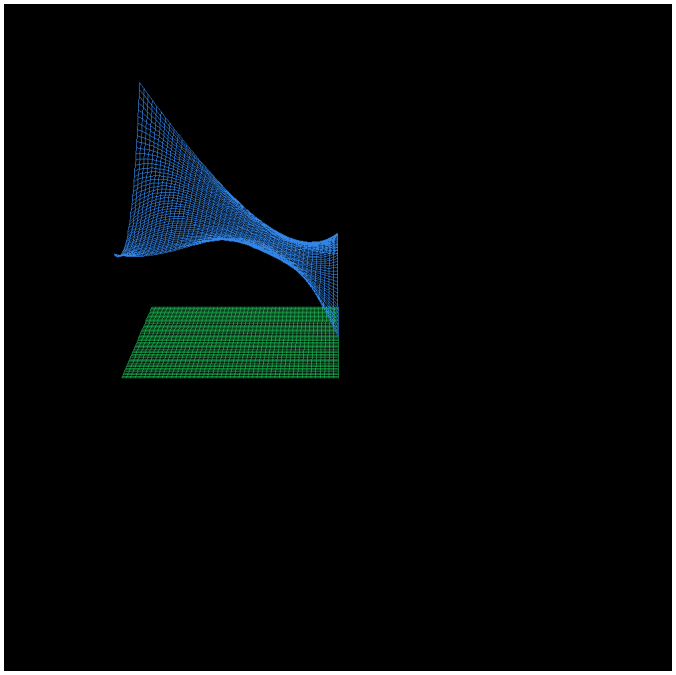

Figure 5: Test image for Bezier surfaces

Now read the teapot.txt file, which contains 32 surfaces. Each line in between the line saying "3 3" specifies a 3D point on a same surface. They are in order. I multiply the coordinates by 5 and subtract 3.5 from y value to move it down a bit. Image is shown below. If you want, you can try something else as well. Klein Bottle would be fun to draw.

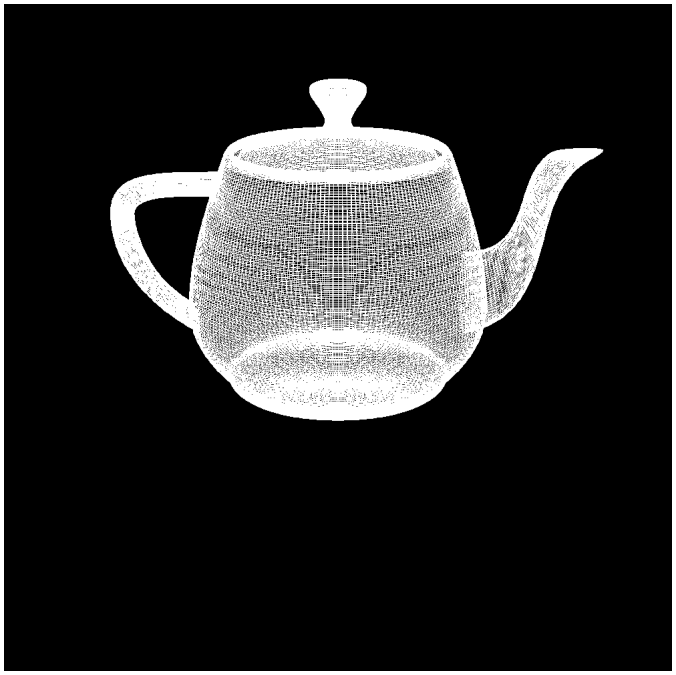

Figure 6: Teapot defined by a set of Bezier curves

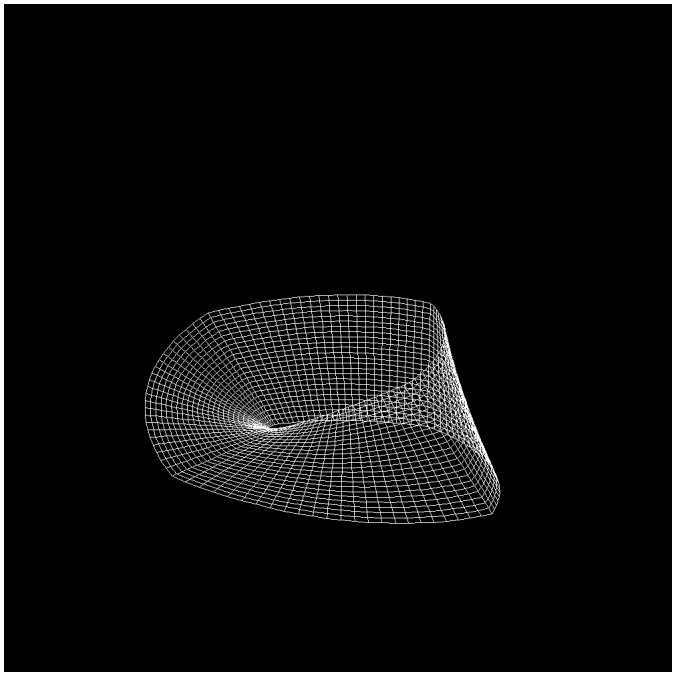

Figure 7: Teapot defined by a set of Bezier curves

Warning!

You have to be really careful about your math and indices. Otherwise, it will take forever to debug and you will not enjoy the project as you should have. Also, creating a subdirectory for storing images is a good idea. Then you can write your images into ./images/frame.png.

What to submit

- Code: .cxx file
- Teapot images or other images produced by Bezier surfaces.

Credit to

This project is based on the projects from CS351 Computer Graphics class in 2017 spring semester and the lecture notes of Professor Bruce Maxwell of Colby College, ME.