Computer Graphics – Lab Assignment
Bézier Curves

September 26, 2010

1 Introduction

Bézier curves were developed in the early 1960's by Pierre Bézier while he was working for Renault, where he used the smooth curves to design automobile bodies.

Bézier curves exist in different *degrees*, differing in the amount of control points. For example *quadratic* Bézier curves, having three control points, are used in Microsoft's TrueType fonts. Most 3D modeling packages and DTP software such as Adobe Illustrator can work with *cubic* Bézier curves, which are defined by four control points.

Although using more control points for a single curve is theoretically possible, the computational effort rises and doesn't really offer much benefit, as individual smaller Bézier curves can be joined together to form longer smooth curves.

2 Bézier Curves

We repeat here the definition of a Bézier curve $P(u)$ of arbitrary degree n. Here, u is the *curve parameter*, with $0 \leq u \leq 1$. You can also read section 15.6 in the book of Shirley et. al.

$$P(u) = \sum_{i=0}^{n} B_i^n(u) P_i \tag{1}$$

Here, $P_0, P_1, \ldots, P_n$ are the $n+1$ control points, $B_i^n(u)$ is the ith Bernstein polynomial of degree n:

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

You will probably know that $\binom{n}{k}$ is called the binomial distribution

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

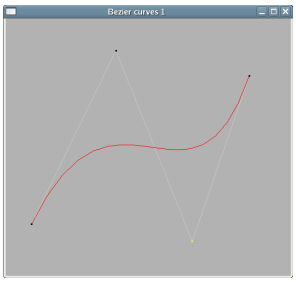

Figure 1: A Bézier curve (red line) with its 4 control points

and $n!$ is the factorial of n, meaning the product $1 \cdot 2 \cdot ... \cdot n$. Note that 0! is defined to have the value 1.

3 Part 1 - Drawing a Bézier curve

In the first part of this assignment you will have to write a function that draws a Bézier curve of an arbitrary degree on the screen. As a Bézier curve is an infinitely smooth curve we can't directly draw it as such. So we're going to approximate it using a number of straight line segments.

After building the framework you should have an executable called `singlecurve`. This executable uses the functions in `bezier.c` to draw a Bézier curve of varying degree.

In *bezier.c* there are two functions that you have to fill in to correctly draw the curve:

```
void evaluate_bezier_curve(float *x, float *y,
        control_point p[], int num_points, float u);
void draw_bezier_curve(int num_segments,
        control_point p[], int num_points);
```

See the comments for these functions for more information on what exactly they should do. See Figure 1 for example output.

The framework provides ways to modify the curve being drawn. The individual control points can be moved using the mouse. By clicking and holding the right mouse button you select the closest control point. If you then move the mouse the selected point will move along.

The number of segments used to draw the curve can be changed using the `[` and `]` keys. The degree of the Bézier curve can be changed with `-` and `+` (there's a maximum of 6 control points).

Test your implementations of the functions by varying the degree of the curve, moving control points, etc.

Note that the second part of this assignment makes use of the code you write in this part, so try to make sure it is correct.

4 Part 2 - Using Bézier curves

An area of computer graphics where parametric curves, such as Bézier curves, are often used is in animation. There, curves are used to control 3D objects over time, say a race car's position and orientation on a race track. There's at least two ways of using curves for this:

- Specify curve control points in 3D. You could, for example, define a camera path this way by simply placing curve control points at the necessary positions in a 3D scene. By varying the curve parameter u as the animation progresses you can compute updated camera positions for each frame.
- Use parametric curves as 2D functions to control an object's values, such as its X position or rotation around the Y axis. This method is more flexible than having just a 3D path, as you can use curves to control all kinds of values, not just position. This is the method used in this part of the assignment.

To make things a bit more clear start the executable called `multicurve`. If you did the curve drawing part correctly you should now see a number of colored lines in the bottom part of the application window. The top part will show a 3D scene (you can manipulate the viewpoint in the latter with the mouse).

Each of the colored smooth lines, which we will call *control lines*, actually consists of five cubic Bézier curves. The curves are connected by making the last control point of a curve be the same as the first control point of the next curve.

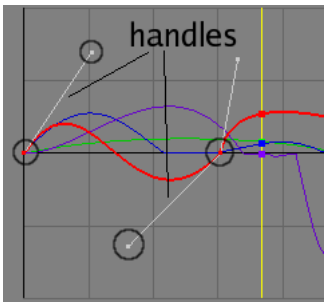

Figure 2: One of the Bézier curves making up the red line

See Figure 2, it shows the first Bézier curve making up the red control line. The four control points of the curve are encircled. The curve always runs through its first and last control point (the end points). The two *handles* controlling the curve's slope are shown as grey lines. In the first part of this assignment the curve was drawn with another line running from the second control point to the third one. In most 3D applications in which you can work with curves this line is left out and only lines from the first to second and third to fourth control point are drawn. This way of drawing the curve highlights

the idea that the grey handles can be used to control the slope of the curve connected to the corresponding end point.

Manipulation of the curves is the same as in the first part, i.e. using the right mouse button and dragging. Play around a bit with the curves to see how to they behave when you move control points. You should notice there are some constraints on where control points can be placed. Try to figure what the reason for these constraints is. Also note that the handles allow for local control of a curve, e.g. the rightmost handle in Figure 2 does not influence the first curve of the red line, only the second curve.

The current animation time is shown with a vertical yellow line, as can be seen in Figure 2. The control line values for the current time are shown with colored markers on the curves.

The 3D scene consists of a "robotic" arm, two rectangular blocks and the infamous Utah teapot. The movement of the robot arm is controlled by the control line values for the current time. Three lines determine the rotation *speed* (not rotation angle!) of the different parts making up the arm, while the fourth line determines the grabber opening/closing speed. Only one control line can be edited at one time, which can be chosen with the keys `1` up to `4`.

We need a way to determine control line values as the animation progresses, to control the arm. For this, function `intersect_cubic_bezier_curve` in `bezier.c` is used. As mentioned above, we use the Bézier curves as 2D functions over time (x), while a curve's value for a certain time is y. For each frame of the animation the framework will call the function multiple times to test for intersection between the cubic Bézier curves making up the control lines and the vertical time line (which is of the form $x = current\_time$).

Tasks:

- Think of a strategy that uses `evaluate_bezier_curve` to test successive values of the curve parameter `u` to find an intersection with the time line (if there is one). Hints:

  1. Remember that due to the constraints placed on control points the resulting curve can be treated as a function
  2. The iteration may produce an approximate intersection point, for example, one with an x-value that is within 10^{-3} of the searched x.

  Fill in function `intersect_cubic_bezier_curve` in `bezier.c` based on your strategy. A curve intersection point returned by this function will be shown as a colored marker in the curve area of the application by the framework.

- Edit the control lines to make the arm perform the following task: let it grab the teapot from the block it's standing on and place it on the other block. By starting up `multicurve` with a file name as argument you can then save your solution to file. *Include this file with the solution code you hand in.*

With the `s` key you can save the current curves to file, or use `r` to reload from file. You can select the file to use as a command-line argument to `multicurve` (which defaults to *curves.txt*). The application will not allow you to quit when there are outstanding modifications to the curves, to make sure you don't accidentally lose any changes. You either need to `s`ave the changes to file or `r`eload the curves from file before quitting.

You can use the `a` key to toggle the animation on/off or use `A` to restart it. Note that it is not possible to jump to a certain time in the animation, because the physics simulation engine used cannot predict what will happen, nor can you go backwards in time.

You can put a control point exactly at Y=0 by selecting it and pressing the `z` key.

5 Grading

You can receive up to 4 points for a correct implementation of drawing a Bézier curve (2 for each of the two functions to fill in). The curve drawing functions should be able to handle curves of arbitrary degree.

Correctly computing the intersection between a curve and the time line can give you up to 3 points.

Making the robot arm perform the described feat is worth 2 points. If you don't manage to place the teapot on the other block you can still get points for seriously trying.

You can get one additional point for writing clean, well-structured and well-commented code (on the opposite side, unreadable or overly complex code might cost you a point).