Graphics and Game Technology

Assignment 3

Orthogonal projection

and

Bézier curves

1 Part 1 - Orthogonal projection

We have previously seen that all world vertices that we add to our scene are first multiplied by the modelview matrix to produce all the necessary transformations (such as translations and rotations) of the scene and/or the camera, with "camera view coordinates" of all the vertices as a result. After these transformations, we still have three-dimensional coordinates but they are now relative to the camera perspective. Next to the camera perspective, the programmer also has to define a view volume. The view volume determines what the camera sees. Everything that lies outside of the view volume will be invisible (or "clipped").

In the next processing step in the graphics pipeline, OpenGL rescales the coordinates from the view volume to fit into a unit cube where they can subsequently be clipped very easily.

In the case of the orthogonal projection, the view volume has the simple form of a three-dimensional box. The box is defined by the distance of the front clipping plane and back clipping plane, and by the distances of the left and right clipping planes as well as the top and bottom clipping planes to the orgin (that now correspond to the position of the camera).

1.1 gluOrtho

OpenGL distinguishes between two different matrices by which the world vertex coordinates are multiplied:

$$\begin{pmatrix} x_{projection} \\ y_{projection} \\ z_{projection} \end{pmatrix} = M_{projection} M_{modelview} \begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \end{pmatrix}$$

The modelview matrix stores translations and rotations, while the projection matrix is used to store the view volume transformation. To switch between the different matrix stacks, call `glMatrixMode(GL_PROJECTION)` to enable the projection matrix or `glMatrixMode(GL_MODELVIEW)` for the modelview matrix. Subsequently, you can use the functions you already know to modify the stack.

1.2 myOrtho

Create the transformation matrix that we can use to set up the orthogonal projection. Your function receives as parameters the dimensions of the view volume:

```
void myOrtho(GLdouble left,
             GLdouble right,
             GLdouble bottom,
             GLdouble top,
             GLdouble near,
             GLdouble far);
```

The resulting matrix does only some scaling and simple translation. For example, the camera coordinates $(left, top, near)$ have to be mapped to the projection coordinates $(1, 1, 1)$ while $(right, bottom, far)$ has to be mapped to a vector $(-1, -1, -1)$. This is the cube. Concerning the translation, it is easy to see that the projection origin corresponds to the camera coordinates $((left - right)/2, (top - bottom)/2, (far - near)/2)$.

2 Part 2 - Bézier Curves

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 have three control points. These are for example used in Microsoft's TrueType fonts. Most 3D modeling packages and Desktop Publishing (DTP) software such as Adobe Illustrator 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 to compute them increases. In addition, this does not really offer much benefit, as individual smaller Bézier curves can be joined together to form longer smooth curves.

2.1 Definition

We repeat here the definition of a Bézier curve $P(u)$ of arbitrary degree n. Here, u is the *curve parameter*, with $0 \leq u \leq 1$. Also look up the appropriate section 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 probably know already that $\binom{n}{k}$ is called the binomial distribution

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

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.

2.2 Drawing a Bézier curve

In the first part of this assignment you must 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 cannot directly draw it as such. So we are going to approximate it using a finite 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 complete in order 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 with 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 is a maximum of 6 control points).

Test your implementation 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 make sure it is correct.

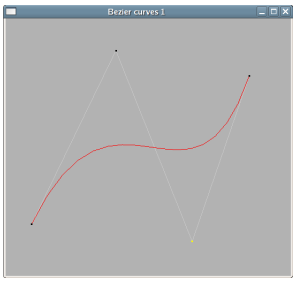

Figure 1: A Bézier curve (red line) with its 4 control points..

2.3 Animation with 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 the position of a race car and its orientation on the race track. There are 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 are what we call *control lines*. They actually consist of five cubic Bézier curves. The curves are connected by making the last control point of a curve the same as the first control point of the next curve.

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 that support 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.

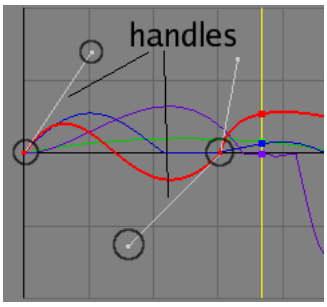

Figure 2: One of the Bézier curves making up the red line.

Manipulation of the curves is the same as in the first part, i.e. using the right mouse button and dragging. Play around with the curves to see how they behave when you move control points. You should notice there are some constraints on where control points can be placed. Try to figure out 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 vertical yellow line in Figure 2 shows the current animation time. 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 famous Utah teapot (see the screenshot in Figure 3). 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 that make up the control lines and the vertical time line (which is of the form $x = current\_time$).
Your 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.

Compete function `intersect_cubic_bezier_curve` in `bezier.c` using 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 is standing on and place it on the other block. By starting up `multicurve` with a filename as argument you can then save your solution to file. *Include this file with the solution code you hand in.*

The `s` key saves the current curves to file. 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.

The `a` key toggles the animation on/off. 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.

3 Grading

If you correctly implement part 1 (myOrtho) you will get up to 2 points.

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 2 points.

Making the robot arm perform the described feat is worth 1 point. This is hard; if you don't manage to place the teapot on the other block you still get the point 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).

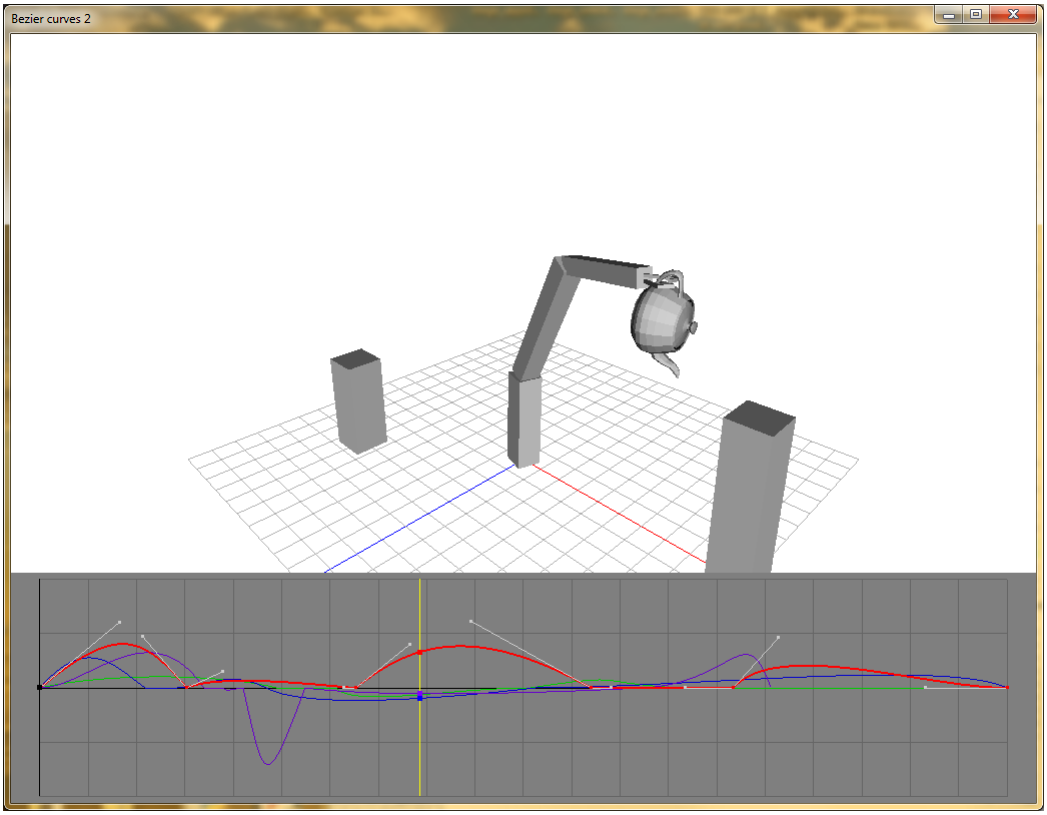

Figure 3: Screenshot of the completed assignment with the robot at work with the Utah teapot.