

Daniel O'Dwyer 1801164 G&GP

by Daniel O'dwyer

FILE	12011_DANIEL_ODWYER_DANIEL_ODWYER_1801164_GGP_311302_180867 1710.DOCX (214.34K)		
TIME SUBMITTED	11-DEC-2020 11:13PM (UTC+0000)	WORD COUNT	893
SUBMISSION ID	139335090	CHARACTER COUNT	4118

Graphics & Games Programming ACCA6011

Parametric Curves

By

Daniel O'Dwyer 1801164

Table of Contents

1.1 Code Overview

Pages 3 – 7

1.2 Bezier Curve Calculation

Page 7

1.3 Screenshots

Pages 8 – 9

1.4 Bibliography

Page 10

1.1 Code Overview

```
void Bham(int x1, int x2, int y1, int y2)
{
    int dx, dy, i, e;
    int incx, incy, inc1, inc2;
    int x, y;

    dx = x2 - x1;
    dy = y2 - y1;

    if (dx < 0) dx = -dx;
    if (dy < 0) dy = -dy;
    incx = 1;
    if (x2 < x1) incx = -1;
    incy = 1;
    if (y2 < y1) incy = -1;
    x = x1; y = y1;
    if (dx > dy)
    {
        graphics.drawPixel(x, y);
        e = 2 * dy - dx;
        inc1 = 2 * (dy - dx);
        inc2 = 2 * dy;
        for (i = 0; i < dx; i++)
        {
            if (e >= 0)
            {
                y += incy;
                e += inc1;
            }
            else
            {
                e += inc2;
                x += incx;
                graphics.drawPixel(x, y);
            }
        }
    }
    else
    {
        graphics.drawPixel(x, y);
        e = 2 * dx - dy;
        inc1 = 2 * (dx - dy);
        inc2 = 2 * dx;
        for (i = 0; i < dy; i++) {
            if (e >= 0) {
                x += incx;
                e += inc1;
            }
            else
            {
                e += inc2;
                y += incy;
                graphics.drawPixel(x, y);
            }
        }
    }
}
```

Firstly, we have bresenham's line drawing algorithm. This is an essential piece of code as it allows us to calculate the correct coordinates to draw a line from start to finish. This algorithm checks the starting point of x and y. It draws the line differently depending on whether the x coordinates are bigger than the y coordinates or vice versa.

```

//Look up table for all possible point values
void LUTMatrix()
{
    float t = 0.00;

    for (int i = 0; i <= 100; i++)
    {
        lut[i][0] = pow((1 - t), 3);
        lut[i][1] = (pow((t), 3) * 3 - pow((t), 2) * 6) + 4;
        lut[i][2] = (pow((t), 3)*-3) + (pow((t), 2) * 3) + (t * 3) + 1;
        lut[i][3] = pow((t), 3);
        t += 0.01;
    }
}

```

The next major part of the code is the look up tables function. This function is crucial for pre calculating values for the curve equation. To render the curve correctly an equation is used to calculate the angle of the curve. Due to the number of variables the program has such as user defined control points, there are a lot of potential values for the equation. The look up table calculates these values just as the program starts, then the user can set the control points and the curve can be rendered in real time.

```

//Renders the curve as the user places points
void RendCurve()
{
    float x;
    float y;

    for (int i = 0; i <= 100; i++)
    {
        //Using the look up table for x co-ords
        x = lut[i][0] * controlPoints[0][0];
        x += lut[i][1] * controlPoints[1][0];
        x += lut[i][2] * controlPoints[2][0];
        x += lut[i][3] * controlPoints[3][0];

        //Using the look up table for y co-ords
        y = lut[i][0] * controlPoints[0][1];
        y += lut[i][1] * controlPoints[1][1];
        y += lut[i][2] * controlPoints[2][1];
        y += lut[i][3] * controlPoints[3][1];

        //Line calculation
        x = x * 0.166666;
        y = y * 0.166666;

        //Draws the line
        graphics.drawPixel(x, y);
    }
}

```

Following the look up tables we have a function that renders the curve. We have two floats that store x and y, we use these in the following for loop. The loop cycles up to 100 which is what we set our resolution to. If we raised it, the for loop would need to be changed to match it. In the loop we use the look up table results along with our x and y coordinates to calculate the placement of the curve. The control points are stored in a 2D array which makes accessing the x and y coordinates of the control points easier. After the base calculation is made each x and y calculation is multiplied by a sixth (0.16666 in code form). Finally, we use the graphics draw pixel command to render the complete curve.

```
void renderScene()
{
    // set a colour
    graphics.setColour( 0.5,0.5,1 );

    // set pixel size..
    graphics.setPointSize(8);

    // draw to an off screen buffer

    for (int i = 0; i < 4; i++)
    {
        graphics.drawPixel(controlPoints[i][0], controlPoints[i][1]);
    }

    RendCurve();

    // render the scene
    graphics.render();
}
```

Next, we have the render scene function. Graphics set colour allows us to change the colour of our pixels, we can change the red, green, and blue values individually. Graphics set point size allow us to change the size of our pixels. The for loop is what we use to render the control points after the user click in their desired place. It cycles up to 4 so the user is only allowed to place 4 control points. We also call the render curve function we looked at earlier. The scene is rendered lastly by calling the graphics render function.

```
int _tmain(int argc, _TCHAR* argv[])
{
    // init glut stuff..
    graphics.create(argc, argv);

    // good place for one-off initialisations and objects creation..
    LUTMatrix();
    // enter game loop..
    graphics.loop();

    return 0;
}
```

In the main function we call the look up table function. We do this so that it initialises once. If we placed it in the update function then it would keep running in the background of the program which would be inefficient and unnecessary.

```
bool bLayDownControlPoints = true;

//Mouse click co-ords
int mouse_X = 0;
int mouse_Y = 0;

int Count = 0;

float controlPoints[4][2];
```

In the render class source file, we have a few variables for storing key components. We have a bool to check whether all the control points are laid down, by default its set to true. We then have two ints for the x and y mouse coordinates. A count for keeping track of the amount of control points used. Lastly, there is a 2D array for the control points. Two sets of four coordinates.

```
//Mouse click function allows user to set control points
void MouseClick(int button, int state, int x, int y)
{
    //Allows the user to left click to set the control point, up to a maximum of 4 times
    if (button == GLUT_LEFT_BUTTON)
    {
        if (state == GLUT_DOWN)
        {
            if (bLayDownControlPoints == true)
            {
                controlPoints[Count][0] = x;
                controlPoints[Count][1] = 500 - y;

                Count++;

                if (Count > 3)
                {
                    bLayDownControlPoints = false;
                }
            }
        }
    }

    //Allows the user to right click, then left click to change the position of the control point
    if (button == GLUT_RIGHT_BUTTON)
    {
        if (state == GLUT_DOWN)
        {
            if (bLayDownControlPoints == false)
            {
                bLayDownControlPoints = true;
            }
            if (Count > 0)
            {
                Count--;
            }
        }
    }
}
```

The final piece in the puzzle is the mouse click function. The first part of the function checks if the left mouse button is being pressed. If it is then the bool statement is triggered, and it allows the user to place four control points. The count is incremented after each click and it stops after four control points have been reached. It stops more control points from being placed by switching the bool statement to false. If the right click button is pressed then the control points can be removed and replaced up to four times. It follows the same principles as the left click statement.

1.2 Bezier Curve Calculation

The main calculation in this program is the uniform non-rational B-spline curve. This is what is drawn when the control points are placed. The equation looks like this,

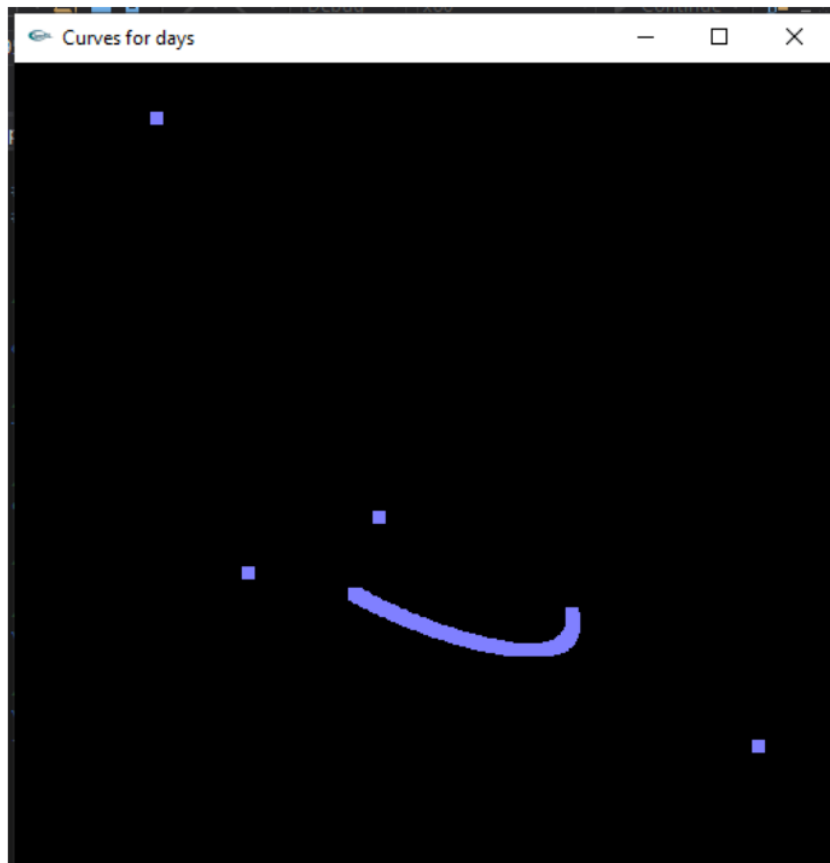
$$\frac{1}{6} \begin{pmatrix} (1-t)^3 & 3t^3 - 6t^2 + 4 & -3t^3 + 3t^2 + 3t + 1 & t^3 \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{pmatrix}$$

Earlier we discussed a part of the program that performs these calculations. In general, a B-spline is quite similar to a Bezier curve, and can be expressed as a function of t (using shifted basis functions), β_n (weighted by control points), and c_k : [1] [p 732].

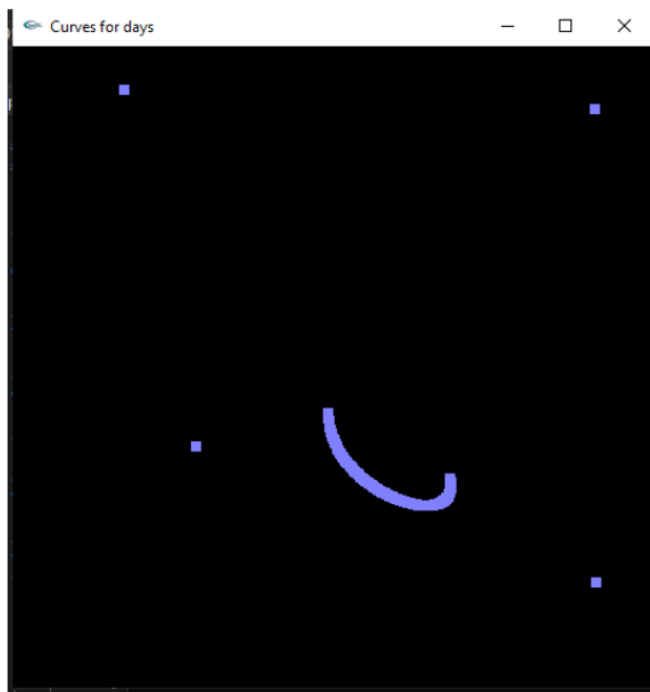
$$[1] \quad s_n(t) = \sum_k c_k \beta_n(t - k).$$

However, in this program we use a non-rational B spline which is a bit trickier to work out. Despite all of the generality B-splines provide, there are some functions that cannot be exactly represented using them. In particular, B-splines cannot represent conic sections. To represent such curves, a ratio of two polynomials is used. Nonuniform B-splines are used to represent both the numerator and the denominator. The most general form of these are nonuniform rational B-splines, or NURBS for short [2] [p 401, 402].

1.3 Screenshots



The first four control points are placed, and the curve is drawn.



One control point is changed. The curve changes with it.



All four control points are changed. The curve changes with them.

1.4 Bibliography

[1] Möller Tomas, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire, Real-time rendering. Boca Raton: CRC Press, 2019.

[2] S. Marschner and P. Shirley, Fundamentals of computer graphics. A K Peters/CRC Press, 2018.

FINAL GRADE

GENERAL COMMENTS

Instructor

76/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

GRADING FORM: SMU GENERIC_GJ

DANIEL O'DWYER

76

RESEARCH-INFORMED LITERATURE



Essential topics identified and reviewed here. Refs included. Relevant and appropriate

7

KNOWLEDGE AND UNDERSTANDING



A good understanding of the requisite curve rendering method demonstrated. Code works and shows a very good understanding of the graphics methods involved.

16

ANALYSIS



A good interpretation of the problem and synthesis of a working solution. Additional control points would have made this submission even better. Code works and shows a very good understanding of the graphics methods involved. Use of mouse control is nice to see.

16

PRACTICAL APPLICATION AND DEP.



Some very, very good work here. Additional control points would have been nice to see but nevertheless a good demonstration of graphics skills in OpenGL. Some additional user interface control through the use of a mouse would be nice to see here.

30

SKILLS FOR PROFESSIONAL PRAC.



A very good report. See comments earlier regarding references. Report structured. Some

7

good SE principles and practice in evidence here.