

Graphics and Games Programming Assignment 1 Report

by Ben Kiddie

FILE	12005_BEN_KIDDIE_GRAPHICS_AND_GAMES_PROGRAMMING_ASSIGNMENT_1_REPORT_311302_2037682988.DOCX (799.45K)		
TIME SUBMITTED	11-DEC-2020 11:53PM (UTC+0000)	WORD COUNT	1434
SUBMISSION ID	139340023	CHARACTER COUNT	7536

Graphics and Games Programming

Assignment 1 – Parametric Curves

by

Ben Kiddie

Student Number: 1800242

Submitted: 11 / 12 / 2020

Abstract

Table of Contents

1. Generating the Curve – Methods and Functions.....	1
2. Point on Curve Calculation.....	12
3.	14
4. Bibliography	18

Table of Figures

Figure 1 Setting up the user interface with <code>cRenderClass::Create</code>	1
Figure 2 Mouse manages the placement of control points.....	2
Figure 3 Keyboard allows the user to toggle curve placements.....	2
Figure 4 <code>SpecialKeyboard</code> manages control point steps and curve resolution changes	3
Figure 5 <code>renderScene</code> performs all the necessary decisions on how control points and curves are rendered	3
Figure 6 Struct types stored in <code>globals.h</code>	4
Figure 7 <code>cBSpline</code> manages the storage and calculations of curves and control points, in addition to interfaces for such data	5
Figure 8 Initialisation of our container class, <code>cBSpline</code>	6
Figure 9 <code>InitialiseBasisFunctionsLUT</code> pre-calculates and stores all potential basis functions ...	7
Figure 10 <code>PlaceControlPoint</code> is called when a left mouse click is performed.....	8
Figure 11 <code>UndoControlPoint</code> lets users reset a control points' position	9
Figure 12 <code>mContinuity</code> interface method	9
Figure 13 <code>DeleteCurveSegment</code> allows users to select a previously placed curve and seamlessly erase it	9
Figure 14 <code>ControlPointStepForward</code> and <code>ControlPointStepBack</code> allow users to navigate across control points	10
Figure 15 Increase and Decrease resolution, or pixels per line.....	10
Figure 16 <code>GetCurveSegments</code> returns the entirety of stored control points	11
Figure 17 <code>GetCurrentResolution</code> returns a value which can change at any point by the user	11
Figure 18 Point on a cubic curve formula [1, Eq. 17.8].....	12
Figure 19 Simplified derivation of a point on a uniform non-rational b-spline [2]	12
Figure 20 Control points placed on-screen for use in calculation testing	13
Figure 21 The four positions, of both X and Y coordinates, in addition to the calculated X and Y values	13

1. Generating the Curve – Methods and Functions

```
void cRenderClass::create(int argc, _TCHAR* argv[])
{
    // initialise the glut library
    glutInit(&argc, argv);

    // set up the initial display mode
    // need both double buffering and z-buffering
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

    // set the initial window position
    glutInitWindowPosition(100, 100);

    // set the initial window size
    glutInitWindowSize(SCREEN_WIDTH, SCREEN_HEIGHT);

    // create and name the window
    glutCreateWindow("Drawing Uniform Non-rational B-Spline Curves");

    // reshape callback for current window
    glutReshapeFunc(winReshapeFunc);

    // set display callback for current window
    glutDisplayFunc(renderScene);

    // set up the global idle callback
    glutIdleFunc(update);

    // Setup user interface functions
    glutMouseFunc(Mouse);
    glutKeyboardFunc(Keyboard);
    glutSpecialFunc(SpecialKeyboard);
}
```

Figure 1 Setting up the user interface with cRenderClass::Create

The author modified a cRenderClass template to provide a user interface. The adjustments made include the methods called at the bottom, which provide keyboard and mouse interactions.

```

void Mouse(int button, int state, int x, int y)
{
    switch (button)
    {
    case GLUT_LEFT_BUTTON:
        if (state == GLUT_DOWN)
        {
            bSplineCurves.PlaceControlPoint(x, y);
        }
        break;

    case GLUT_RIGHT_BUTTON:
        if (state == GLUT_DOWN)
        {
            bSplineCurves.UndoControlPoint();
        }
        break;

    default:
        break;
    }
}

```

Figure 2 Mouse manages the placement of control points

Mouse allows a user to put down a new control point or move a currently selected control point. Additionally, points can be 'undone' – where the previously placed point is moved outside the viewable window.

```

void Keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
    case 'c':
        bSplineCurves.ToggleContinuity();
        break;
    case 'C':
        bSplineCurves.ToggleContinuity();
        break;
    case 'd':
        bSplineCurves.DeleteCurveSegment();
        break;
    case 'D':
        bSplineCurves.DeleteCurveSegment();
        break;
    }
}

```

Figure 3 Keyboard allows the user to toggle curve placements

Pressing 'c' with the keyboard toggles continuity. When enabled, the next control point placement will put down a whole new curve with C2 continuity. Pressing 'd' will delete the curve currently selected.

```

void SpecialKeyboard(int key, int x, int y)
{
    switch (key)
    {
        case GLUT_KEY_LEFT:
            bSplineCurves.ControlPointStepBack();
            break;
        case GLUT_KEY_RIGHT:
            bSplineCurves.ControlPointStepForward();
            break;
        case GLUT_KEY_PAGE_UP:
            bSplineCurves.IncreaseResolution();
            break;
        case GLUT_KEY_PAGE_DOWN:
            bSplineCurves.DecreaseResolution();
            break;
    }
}

```

Figure 4 SpecialKeyboard manages control point steps and curve resolution changes

SpecialKeyboard allows the user to perform ‘stepping’ by pressing left or right on the keyboard. This is a process by which the user can select previously placed points by moving incrementally backwards, or forwards if already stepped back. Page Up and Page Down keys increase and decrease resolution (i.e. pixels per curve) respectively.

```

void renderScene()
{
    std::vector<curveSegment> curveSegments = bSplineCurves.GetCurveSegments(); // temporarily store the current curve segments data with their associated control points and co
    vector pointOnCurve; // stores the x and y values for which pixel we want to render on a curve
    int i, j; // Stores the index values for which curve and knot we have looped to - used in calculating points on a curve to render
    int curvesToRender = curveSegments.size(); // If we have zero curves, stop the program from accessing values that do not exist. Also determines the number of curves to iterate
    int currentResolution = bSplineCurves.GetCurrentResolution(); // Stores the number of knots (i.e. pixels) we are rendering on our curve

    // We get the control point selected on a specific curve and render it differently to the user. Only renders differently if user has stepped backwards.
    int currentControlPoint = bSplineCurves.GetCurrentControlPoint();
    int currentCurveSegment = bSplineCurves.GetCurrentCurveSegment();

    if (curvesToRender > 0) // Prevents the program from rendering non-existent curves
    {
        for (i = 0; i < curvesToRender; i++) // Render the control points and curve pixels associated with the current curve we are looking at, 'i'
        {
            // Initialise control point rendering
            if (bSplineCurves.GetContinuity() == true)
            {
                graphics.setColour(0, 255, 0);
            }
            else
            {
                graphics.setColour(255, 255, 255);
            }
            graphics.setPointSize(2);

            // Render each control point for the current curve, 'i', with initialised render values
            for (j = 0; j < 4; j++)
            {
                graphics.drawPixel(curveSegments[i].controlPoints[j][0], curveSegments[i].controlPoints[j][1]);
            }

            // Initialise curve rendering
            graphics.setColour(curveSegments[i].curveColour.r, curveSegments[i].curveColour.g, curveSegments[i].curveColour.b);

            for (j = 0; j <= currentResolution; j++) // Render each pixel on our curve with x and y values obtained by passing in the curve we are on (utilises current control p
            {
                pointOnCurve = bSplineCurves.CalculatePointOnCurve(float(j) / bSplineCurves.GetCurrentResolution(), i);
                graphics.drawPixel(pointOnCurve.GetComponent(0), pointOnCurve.GetComponent(1));
            }

            // Render the control point which the user has currently selected differently as a visual aid
            graphics.setPointSize(6);
            graphics.setColour(255, 0, 0);
            graphics.drawPixel(curveSegments[currentCurveSegment].controlPoints[currentControlPoint][0], curveSegments[currentCurveSegment].controlPoints[currentControlPoint][1]);
        }
    }

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

```

Figure 5 renderScene performs all the necessary decisions on how control points and curves are rendered

renderScene manages the rendering for three items. First is control points – we render the colour for these differently based on the state of continuity, where the number of control points is four per curve. Next is the control point currently selected, which renders larger and in red relative to other control points. Lastly, we render pixels on our curve, for which the X and Y coordinates are stored in a vector calculated through the CalculatePointOnCurve function.

```
cVector cBSpline::CalculatePointOnCurve(float t, int curveIndex) // Returns the X and Y coordinates for a pixel t
{
    // Stores a calculated integer index to access the appropriate pre-calculated basis functions for a given knot
    int basisIndex = static_cast<int>(t * mMaxResolution);

    int b1 = (basisIndex * 4) + 0, b2 = (basisIndex * 4) + 1, b3 = (basisIndex * 4) + 2, b4 = (basisIndex * 4) + 3;

    // Calculate and store the X and Y component of our pixel on the curve with a weighting provided by our basis fu
    float xComponent = 0.166666
        * ((mBasisFunctionsLUT[b1] * mCurveSegments[curveIndex].controlPoints[0][0])
          + (mBasisFunctionsLUT[b2] * mCurveSegments[curveIndex].controlPoints[1][0])
          + (mBasisFunctionsLUT[b3] * mCurveSegments[curveIndex].controlPoints[2][0])
          + (mBasisFunctionsLUT[b4] * mCurveSegments[curveIndex].controlPoints[3][0]));

    float yComponent = 0.166666
        * ((mBasisFunctionsLUT[b1] * mCurveSegments[curveIndex].controlPoints[0][1])
          + (mBasisFunctionsLUT[b2] * mCurveSegments[curveIndex].controlPoints[1][1])
          + (mBasisFunctionsLUT[b3] * mCurveSegments[curveIndex].controlPoints[2][1])
          + (mBasisFunctionsLUT[b4] * mCurveSegments[curveIndex].controlPoints[3][1]));

    //If (t == 0.5) // This tests our t value of 0.5 for accuracy
    //{
    //    std::cout << mCurveSegments[curveIndex].controlPoints[0][0] << ", " << mCurveSegments[curveIndex].controlPoi
    //}

    // Instantiate and return a vector with coordinates set to our weighted pixel
    cVector pointOnCurve = cVector(xComponent,yComponent,0);
    return pointOnCurve.GetVector();
}
```

Figure 6 CalculatePointOnCurve is the means for which we render our b-spline curve

This function, given the knot value and set of control points garnered from our curve index, returns a vector with X and Y positions for rendering a pixel on our curve.

```
struct sRGB // Holds an RGB val
{
    float r, g, b;
};

struct sCurveSegment // Each
{
    float controlPoints[4][2];
    sRGB curveColour;
};
```

Figure 7 Struct types stored in globals.h

sRGB is used in storing colour combinations. sCurveSegment defines four control points of X and Y coordinates, with an associated colour.

```

class cBSpline
{
public:
    cBSpline();
    ~cBSpline();

    void IncreaseResolution();
    void DecreaseResolution();
    int GetCurrentResolution();
    //int GetMaxResolution();

    void InitialiseBasisFunctionsLUT();

    void AddNewCurve(int colourIndex);
    std::vector<sCurveSegment> GetCurveSegments();

    int GetCurrentControlPoint();
    void PlaceControlPoint(int x, int y);
    void UndoControlPoint();
    void MoveControlPoint(int x, int y);

    void ControlPointStepForward();
    void ControlPointStepBack();
    //int GetControlPointStepTracker();

    int GetCurrentCurveSegment();
    void DeleteCurveSegment();
    void NextCurveSegment();
    void PreviousCurveSegment();

    void ToggleContinuity();
    bool GetContinuity();

    cVector CalculatePointOnCurve(float t, int curveIndex);

private:
    int mCurrentResolution = 0; // Used in calculating the number of segments
    int mMaxResolution = 0; // The upper limit of pixels to be used
    int mMinResolution = 0; // Lower limit of pixels per curve segment
    int mResolutionMod = 0; // Increases or decreases pixels per segment

    float *mBasisFunctionsLUT; // Pre-calculates the basis functions

    std::vector<sCurveSegment> mCurveSegments; // Holds all curve segments

    int mCurrentControlPoint = 0; // Tracks the current control point
    int mControlPointStepTracker = 0; // Tracks the number of steps

    int mCurrentCurveSegment = 0; // Tracks the current curve segment

    bool mContinuity = false; // When set to false curves are not continuous
};

```

Figure 8 cBSpline manages the storage and calculations of curves and control points, in addition to interfaces for such data

cBSpline provides the foundations for storage and interfacing of curve data. Of note is the vector of curve segments, where we store all our control points. Furthermore, a look-up table holding pre-calculated values used in weighting assigned to this class.

```

cBSpline::cBSpline()
{
    mMaxResolution = 500;
    mMinResolution = 100;
    mResolutionMod = 50;

    mCurrentResolution = mMaxResolution;
    InitialiseBasisFunctionsLUT(); // St
    AddNewCurve(0); // Initialise the cur
}

cBSpline::~cBSpline()
{
}

```

Figure 9 Initialisation of our container class, cBSpline

The constructor ensures the program starts correctly by initialising the curve vector. The basis functions look-up table is automatically populated on program start-up.

```

void cBSpline::AddNewCurve(int colourIndex) // The co
{
    sCurveSegment curveSegment;
    curveSegment.curveColour = colours[colourIndex];
    mCurveSegments.push_back(curveSegment);
}

```

Figure 10 AddNewCurve pushes a new curve object into our vector, ready for control point assignment

The parameter colourIndex can be in the range 0-6. Once all details of our new curve are established, we add it to the vector.

```

void cBSpline::InitialiseBasisFunctionsLUT() // Calculates the basis functions for all values of 't'
{
    mBasisFunctionsLUT = new float[(mMaxResolution * 4) + 4]; // Instantiate a new array which encompasses all potential basis functions
    float t; // Stores the current knot number for which corresponding basis functions are calculated
    for (int i = 0; i <= mMaxResolution; i++) // For every possible knot (i.e. pixel) we render on our screen
    {
        t = float(i) / float(mMaxResolution); // Holds the index position of the knot we are looking for
        for (int j = 0; j < 4; j++) // For each basis function, we provide our current knot 't' and store the result in the array
        {
            switch (j)
            {
                case 0:
                    mBasisFunctionsLUT[(i * 4) + j] = (1 - t) * (1 - t) * (1 - t);
                    break;

                case 1:
                    mBasisFunctionsLUT[(i * 4) + j] = (3 * (t * t * t)) + (-6 * (t * t)) + 4.0f;
                    break;

                case 2:
                    mBasisFunctionsLUT[(i * 4) + j] = (-3 * (t * t * t)) + (3 * (t * t)) + (3 * t) + 1.0f;
                    break;

                case 3:
                    mBasisFunctionsLUT[(i * 4) + j] = t * t * t;
                    break;
            }
        }
    }
}

```

Figure 11 InitialiseBasisFunctionsLUT pre-calculates and stores all potential basis functions

For all values 0 – mMaxResolution, we calculate four basis functions. The results are stored in an array, mBasisFunctionsLUT.

```

void cBSpline::PlaceControlPoint(int x, int y) // PlaceControlPoint takes an X and Y value from the function MouseClick
{
    // Standard control point placement - we utilise passed in X and Y values from our mouse click to add a control point
    if (mContinuity == false)
    {
        if (mControlPointStepTracker == 0) // This check is to ensure we add new control points if the user has not stepped backwards
        {
            mCurveSegments[mCurrentCurveSegment].controlPoints[mCurrentControlPoint][0] = x;
            mCurveSegments[mCurrentCurveSegment].controlPoints[mCurrentControlPoint][1] = 500 - y;

            // Ensure the user is stepped over to the next curve if we land on control point four, since curves have 4 control points
            if (mCurrentControlPoint < 3)
            {
                mCurrentControlPoint++;
            }
            else
            {
                NextCurveSegment();
                AddNewCurve(mCurrentCurveSegment % 7);
            }
        }
        else // As we have stepped backwards, the user will instead move currently selected control points
        {
            MoveControlPoint(x, y);
        }
    }
    else if (mCurrentCurveSegment != 0 && mControlPointStepTracker == 0) // This check ensures that we don't try to add a new curve if we are at the first curve
    {
        sCurveSegment curveSegment;

        // Set the colour of the new curve to be equal to the current curve
        curveSegment.curveColour = mCurveSegments[mCurrentCurveSegment - 1].curveColour;

        // Control points 1,2,3 of new curve equal 2,3,4 of previous curve
        for (int i = 0; i < 3; i++)
        {
            curveSegment.controlPoints[i][0] = mCurveSegments[mCurrentCurveSegment - 1].controlPoints[i + 1][0];
            curveSegment.controlPoints[i][1] = mCurveSegments[mCurrentCurveSegment - 1].controlPoints[i + 1][1];
        }

        // The final control point to be set in this continuous curve is passed in as parameters X and Y
        curveSegment.controlPoints[3][0] = x;
        curveSegment.controlPoints[3][1] = 500 - y;

        mCurveSegments.insert(mCurveSegments.begin() + mCurrentCurveSegment, curveSegment);
        NextCurveSegment();
    }
}

```

Figure 12 PlaceControlPoint is called when a left mouse click is performed

The X and Y position of the users' mouse when this function is called will serve as the basis for one of two approaches. If mContinuity is false, we add this control point to next appropriate array position. Otherwise, this position, in addition to the three previous, will serve as control points 1 – 4 in a new curve to be added to our vector.

```

void cBSpline::UndoControlPoint() // Users have the ability to reset the position of control points that
{
    if (mControlPointStepTracker == 0)
    {
        if (mCurrentControlPoint > 0)
        {
            mCurrentControlPoint--;
        }
        else
        {
            if (mCurrentCurveSegment > 0)
            {
                mCurveSegments.pop_back();
                PreviousCurveSegment();
            }
        }

        // Setting control points to appear off-screen, so they don't get rendered due to clipping (otherwise
        mCurveSegments[mCurrentCurveSegment].controlPoints[mCurrentControlPoint][0] = SCREEN_WIDTH - 10;
        mCurveSegments[mCurrentCurveSegment].controlPoints[mCurrentControlPoint][1] = SCREEN_HEIGHT + 10;
    }
}

```

Figure 13 UndoControlPoint lets users reset a control points' position

Upon the user pressing right mouse button, the previously placed control point has its position set to outside the render window.

```

void cBSpline::ToggleContinuity()
{
    mContinuity = !mContinuity;
}

```

Figure 14 mContinuity interface method

ToggleContinuity lets the user shift between continuous and non-continuous curve placement by pressing 'c'.

```

void cBSpline::DeleteCurveSegment() // Allows the user to erase existing c
{
    mCurveSegments.erase(mCurveSegments.begin() + mCurrentCurveSegment);

    // Generally the user is placed at the previous curves' third control
    if (mCurrentCurveSegment > 0)
    {
        PreviousCurveSegment();
    }
    else
    {
        mCurrentControlPoint = 0;
        AddNewCurve(mCurrentCurveSegment % 7);
    }
}

```

Figure 15 DeleteCurveSegment allows users to select a previously placed curve and seamlessly erase it

When the user presses 'd', the curve currently selected is deleted, and the user is sent to the previous curve.

```

void cBSpline::ControlPointStepForward() // Allows us to track
{
    if (mControlPointStepTracker > 0) // Only allow steps forward
    {
        if (mCurrentControlPoint < 3)
        {
            mCurrentControlPoint++;
        }
        else
        {
            NextCurveSegment();
        }

        // Once this reaches zero, no more steps forward taken
        mControlPointStepTracker--;
    }
}

void cBSpline::ControlPointStepBack() // Lets the user select
{
    if (mCurrentCurveSegment != 0 || mCurrentControlPoint != 0)
    {
        if (mCurrentControlPoint > 0) // Step back along the curve
        {
            mCurrentControlPoint--;
        }
        else // Otherwise step to the previous curve
        {
            PreviousCurveSegment();
        }

        mControlPointStepTracker++; // Always increment the step tracker
    }
}

```

Figure 16 ControlPointStepForward and ControlPointStepBack allow users to navigate across control points

The step forward is called when the right key is pressed, while step back runs on left key press. They move the user forwards or backwards incrementally along all placed control points.

```

void cBSpline::IncreaseResolution() // Will increase pixels per
{
    float newResolution = mCurrentResolution + mResolutionMod;
    if (newResolution <= mMaxResolution)
    {
        mCurrentResolution = newResolution;
    }
}

void cBSpline::DecreaseResolution() // Decreases pixels per curve
{
    float newResolution = mCurrentResolution - mResolutionMod;
    if (newResolution >= mMinResolution)
    {
        mCurrentResolution = newResolution;
    }
}

```

Figure 17 Increase and Decrease resolution, or pixels per line

When the user presses the page up or page down key, the number of pixels rendered per line is increased or decreased respectively by a set modifier, defaulted to 10.

```
std::vector<SCurveSegment> cBSpline::GetCurveSegments()  
{  
    return mCurveSegments;  
}
```

Figure 18 GetCurveSegments returns the entirety of stored control points

When rendering control points, we want to pass said points to the render loop.

```
int cBSpline::GetCurrentResolution()  
{  
    return mCurrentResolution;  
}
```

Figure 19 GetCurrentResolution returns a value which can change at any point by the user

This lets the render loop know how many pixels to draw.

2. Point on Curve Calculation

To calculate the point we wish to render, we must look to the below:

$$\mathbf{p}(t) = (1-t)^3 \mathbf{p}_0 + 3t(1-t)^2 \mathbf{p}_1 + 3t^2(1-t) \mathbf{p}_2 + t^3 \mathbf{p}_3.$$

Figure 20 Point on a cubic curve formula [1, Eq. 17.8]

Which can be simplified to:

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

Figure 21 Simplified derivation of a point on a uniform non-rational b-spline [2]

Where $0 \leq t \leq 1$, where any number which lies in this range represents the knot we are calculating a pixel to draw for.

With regards to Figure 6, the results of a point on curve calculation will be compared to the result of calculating Figure 21, accurately determining X and Y coordinates of the pixel on our curve. The value of 't' evaluated will be 0.1 in both cases, the X and Y of our control points will be shown in Figure 23.

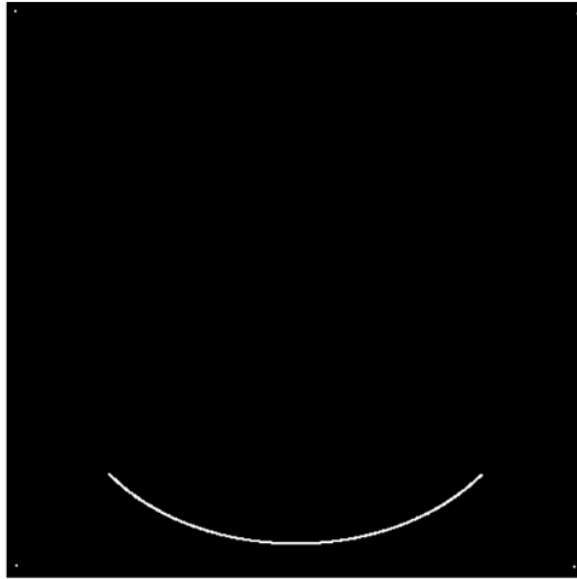


Figure 22 Control points placed on-screen for use in calculation testing

```
10, 489
11, 16
487, 15
490, 487
249.041, 35.1874
```

Figure 23 The four positions, of both X and Y coordinates, in addition to the calculated X and Y values

With regards to Figure 17, and $t = 0.1$, we calculate the basis functions using the formula in Figure 21:

$$(1 - 0.5)^3 \times 10 = 1.25, \quad 3(0.5)^3 - 6(0.5)^2 + 4 = 2.875$$

$$((-3(0.5)^3) + (3(0.5)^2) + 3(0.5) + 1) = 2.875, \quad (0.5)^3 = 0.125$$

Now that we have each basis weighting, we multiply each by both X and Y coordinates:

$$xPos = \frac{1}{6} \times ((1.25 \times 10) + (2.875 \times 11) + (2.875 \times 487) + (0.125 \times 490)) = 250.917$$

$$yPos = \frac{1}{6} \times ((1.25 \times 489) + (2.875 \times 16) + (2.875 \times 15) + (0.125 \times 487)) = 35.188$$

As seen above, the results are incredibly close to the computed values.

3. Functionality

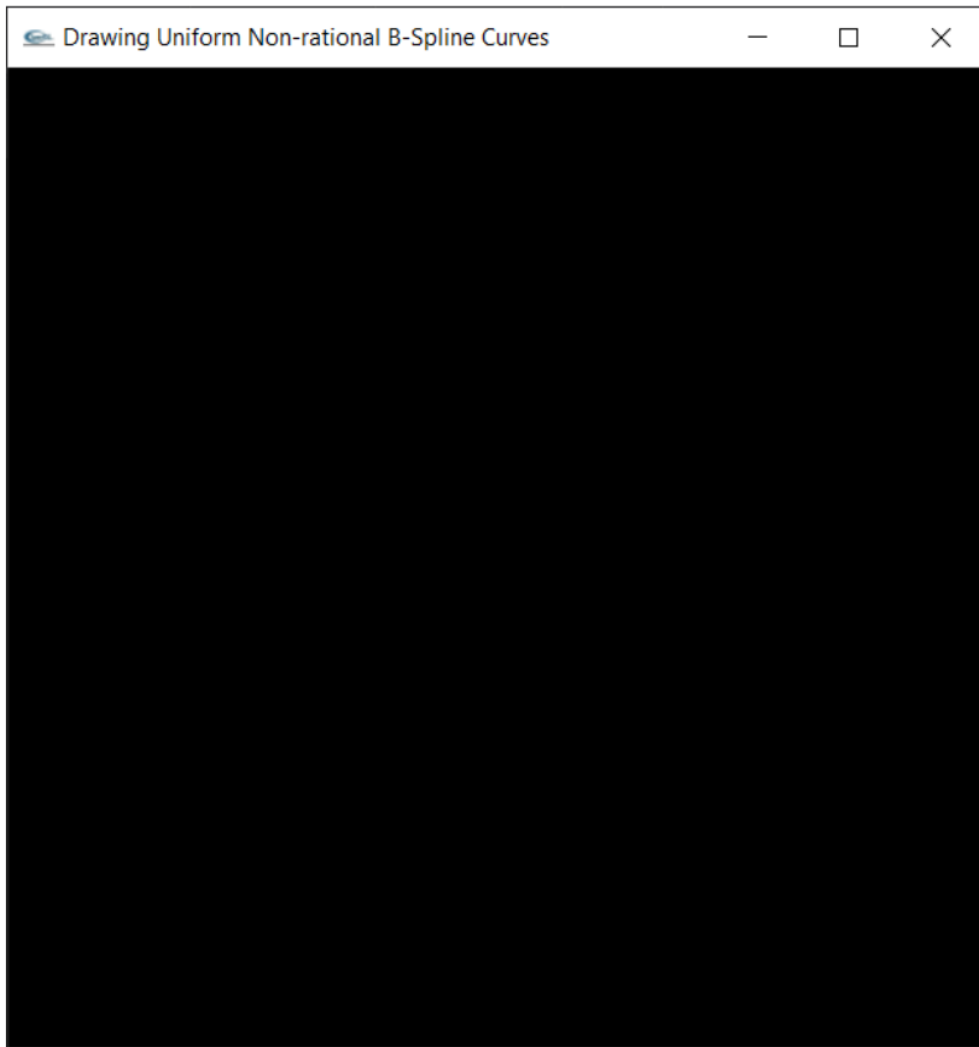


Figure 24 Empty graphics window

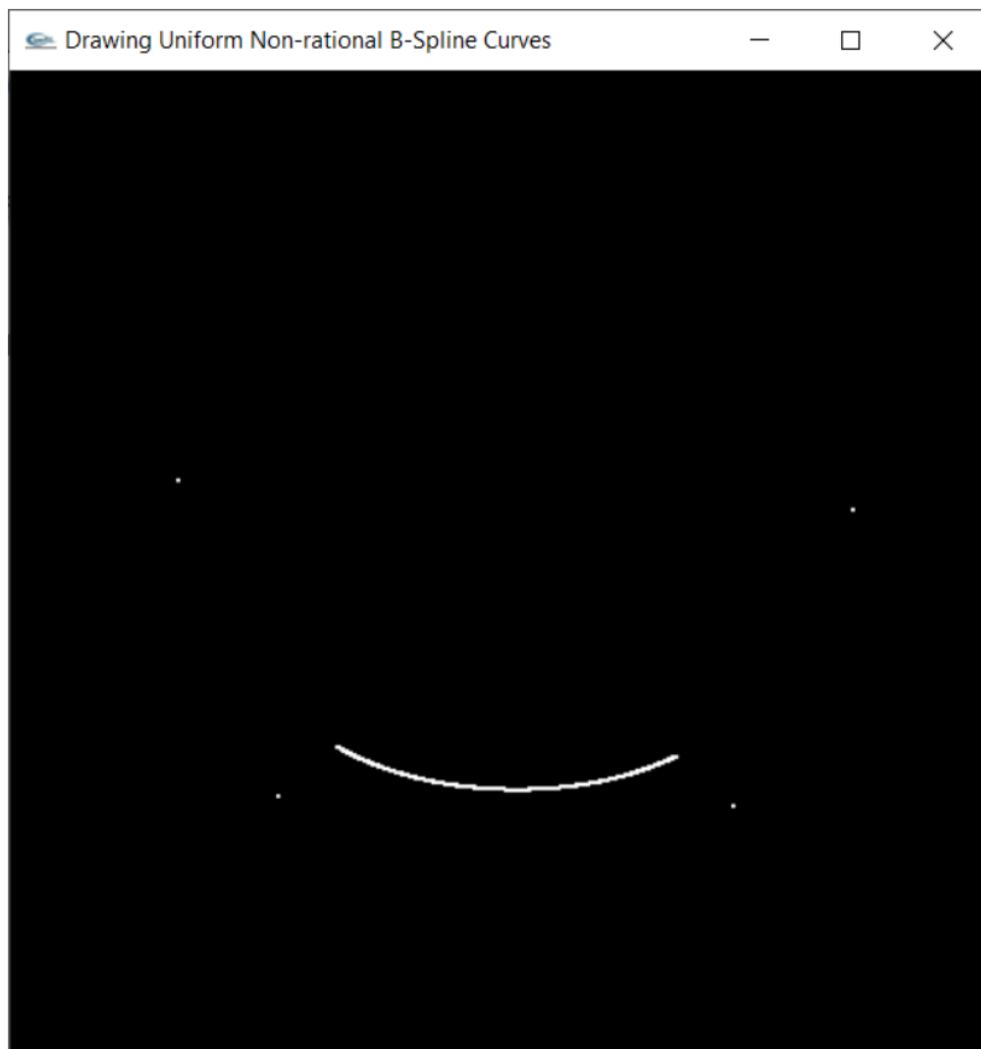


Figure 25 Four control points placed via mouse clicks, with curve rendered on screen pixel by pixel (no interpolation between pixels)

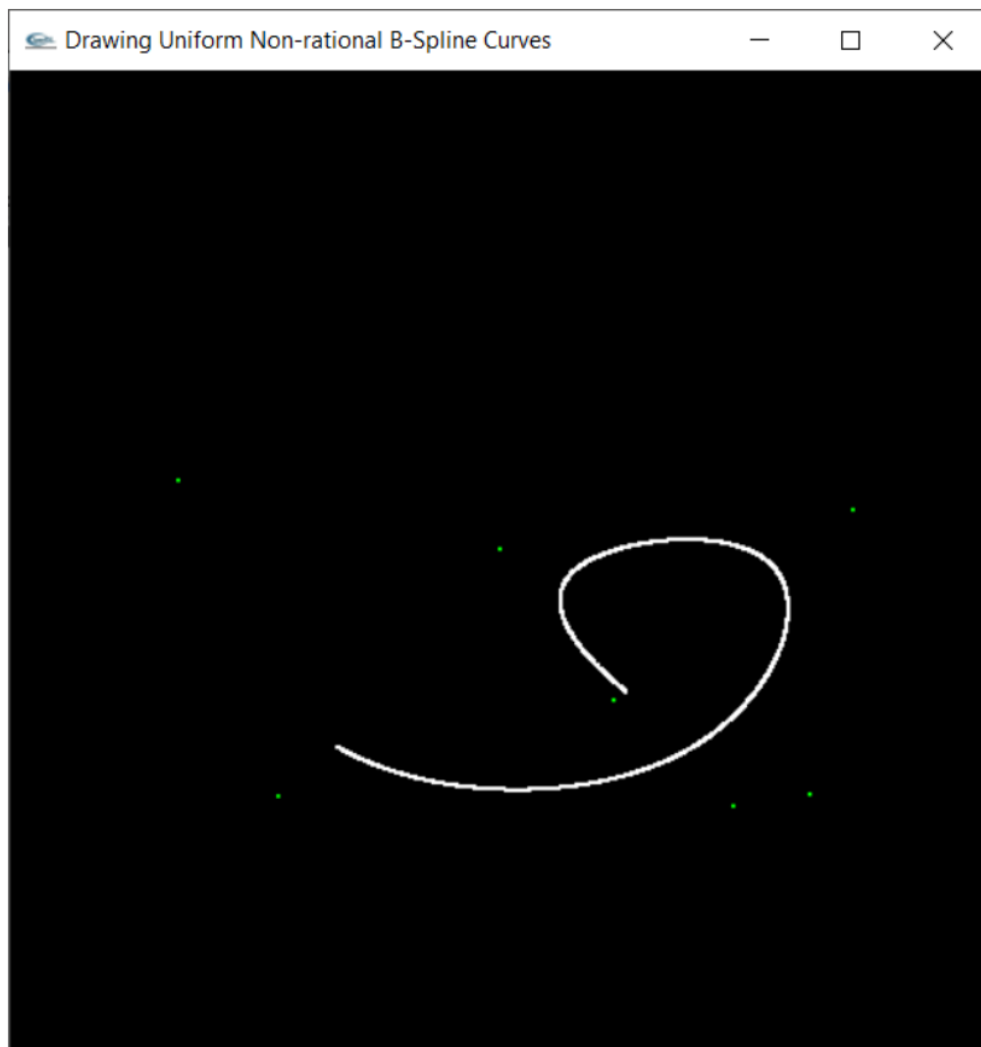


Figure 26 Continuous curve placement

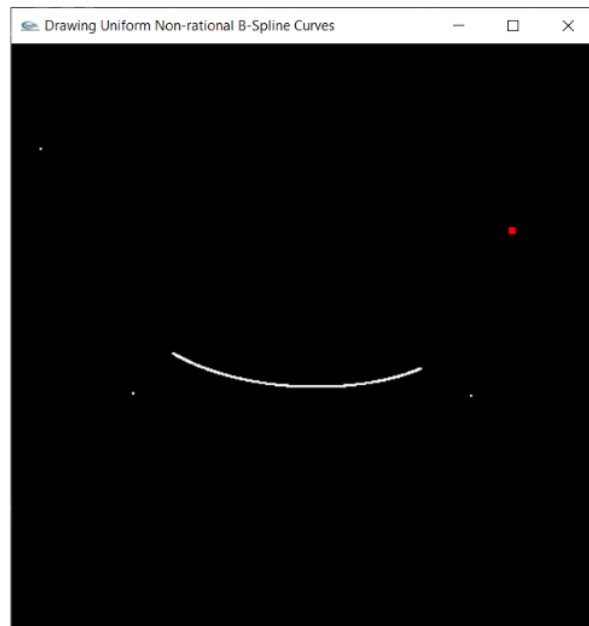


Figure 27 Selecting a previous control point

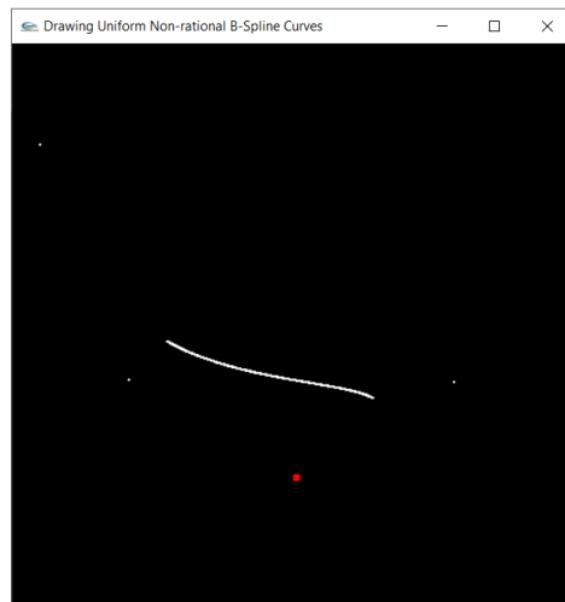


Figure 28 Moving previous control points selected with left mouse click

4. Bibliography

- [1] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki and S. Hillaire, Real-Time Rendering Fourth Edition, Taylor & Francis Group, LLC: Taylor & Francis, 2018.
- [2] A. Baker, "Parametric Curves," 2020.

Graphics and Games Programming Assignment 1 Report

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

86/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

GRADING FORM: SMU GENERIC_GJ

BEN KIDDIE**86****RESEARCH-INFORMED LITERATURE**

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

7**KNOWLEDGE AND UNDERSTANDING**

An excellent understanding of the B spline method demonstrated. Code works and shows a very good understanding of the graphics methods involved.

17**ANALYSIS**

A excellent interpretation of the problem and synthesis of a working solution. Additional control points made this submission even better. An excellent understanding of the B spline method demonstrated. Code works and shows a very good understanding of the graphics methods involved.

18

PRACTICAL APPLICATION AND DEP.

 Some excellent work here. Multiple splines possible. Interpolation between pixels would have been nice to see, however you make it possible to vary the pixel resolution. Nevertheless an excellent implementation.

37

SKILLS FOR PROFESSIONAL PRAC.

 You have a section for an abstract - it doesn't appear here though? However, in general, a very good report. Report structured. Some good SE principles and practice in evidence here. Refs included.

7