

Bartosz Galantowicz - Graphics and Programming (ACCA6021) - Parametric Curves

by Bartosz Galantowicz

FILE	2275_BARTOSZ_GALANTOWICZ_BARTOSZ_GALANTOWICZ_-_GRAPHICS_AND_PROGRAMMING_ACCA6021_-_PARAMETRIC_CURVES_311302_176857044.DOCX (1.24M)		
TIME SUBMITTED	10-DEC-2020 11:01PM (UTC+0000)	WORD COUNT	2254
SUBMISSION ID	139129418	CHARACTER COUNT	12274

Bartosz Galantowicz

1700994

Advanced Graphics

ACCA6011

Andrew Baker & Gordon Dickers

“Parametric Curves”

1 TABLE OF CONTENTS

2	Abstract.....	3
3	Introduction	3
4	Design.....	4
5	Implementation	4
5.1	B-Spline Curve.....	4
5.2	Curve Manipulation System.....	16
6	Conclusion.....	21
7	Bibliography	22
8	Appendix	23
8.1	ReadMe Bartosz Galantowicz	23

2 ABSTRACT

This work is a report describing functionalities and usage of the parametric curves in computer graphics. Moreover, the algorithms applied to calculate and draw Uniform Non-Rational Curves are discussed and described in more details. The examples of manipulation of the curve are documented due to using the features given by OpenGL Utility Toolkit.

3 INTRODUCTION

The aim of this assignment is to develop graphics procedures to generate Uniform Non-Rational Curves in OpenGL Utility Toolkit. Moreover, the functionalities to facilitate the real-time manipulation of the parametric curve using a set of control points are provided by the created application.

The lecturer has provided the framework to work with the program. However, the implementations of the algorithms and programming techniques to achieve the objectives of the assignment were in the responsibility of the student.

C++ programming language was used to develop the final project in Visual Studio 2015 IDE. Additionally, the provided OpenGL Utility Toolkit library was incorporated into the program to set up utilities for generating graphics and writing and reading inputs from a user.

4 DESIGN

The application is a console application. After starting the application, the menu is printed in the console window. User can interact with the program by providing an input to the console.

Firstly, a user is asked to input the control points for the curve. Accordingly, it is requested to enter the resolution of the curve. The higher the resolution of the curve is, the smoother it is displayed on the screen.

Secondly, the user can choose among the options basing on the printed menu. The rendering of the curve is begun by selecting the first option. Exiting the application is made by choosing the last option. The rest of the configurations are responsible for modifying the features of the curve, like changing coordinates of the controlling points or adjusting the resolution.

Finally, if the control points and the resolution are set up, the curve can be rendered in the window. The instruction of manipulating the control points is displayed to a user. It can control the position of each control point by input from the keyboard in the real-time.

5 IMPLEMENTATION

5.1 B-SPLINE CURVE

The designed mathematical library was implemented to the project to allow easy calculation of basic functions of computer graphics features. For example, the control points are stored in `cVector` class, which describes a point in 2D space.

```

1  #include "nMathLibrary.h"
2
3  namespace mathLibrary
4  {
5
6      // Vector class
7      ///////////////////////////////////////////////////
8
9      // cVecot's constructors
10     cVector::cVector(double xX = 0, double yY = 0)
11     {
12         // Set the vector's position
13         x = xX;
14         y = yY;
15         z = 1;
16         vector[0][0] = x;
17         vector[1][0] = y;
18         vector[2][0] = z;
19     }
20
21     cVector::cVector()
22     {
23         // Set the vector's position
24         x = 0;
25         y = 0;
26         z = 1;
27         vector[0][0] = x;
28         vector[1][0] = y;
29         vector[2][0] = z;
30     }
31
32     cVector::~cVector()
33     {
34     }
35

```

Fig 1. cVector class is made of variables which can be sued to describing a point in coordinates system.

The functions responsible for rendering graphics shapes are incorporated into the main file of the program. Namely, drawing a point, drawing a line or drawing a B-Spline curve are managed by these functions.

```

526 // Point clipping algorithm
527 void pointClipping(cVector &point)
528 {
529     int xMin = SCREEN_WIDTH - SCREEN_WIDTH;
530     int xMax = SCREEN_WIDTH;
531     int yMin = SCREEN_HEIGHT - SCREEN_HEIGHT;
532     int yMax = SCREEN_HEIGHT;
533
534     if ((point.x >= xMin && point.x <= xMax) && (point.y >= yMin && point.y <= yMax))
535     {
536         // Render a point
537         //std::cout << "The point is rendered!" << endl;
538         graphics.drawPixel(point.x, point.y);
539     }
540     else
541     {
542         //std::cout << "The point is clipped!" << endl;
543     }
544 }
545
546 // Utility function to draw a point
547 void drawPixel(cVector &point)
548 {
549     pointClipping(point);
550 }

```

Fig 2. The figure is showing functions responsible for drawing and clipping a pixel.

```

428 // Bresenham's line drawing algorithm
429 void drawLine(cVector &point1, cVector &point2)
430 {
431     double x1 = point1.x;
432     double x2 = point2.x;
433     double y1 = point1.y;
434     double y2 = point2.y;
435
436     // Determine steps for X and Y axis
437     int stepX, stepY = 0;
438     if (x1 <= x2)
439     {
440         stepX = 1;
441     }
442     else
443     {
444         stepX = -1;
445     }
446
447     if (y1 <= y2)
448     {
449         stepY = 1;
450     }
451     else
452     {
453         stepY = -1;
454     }
455
456     // Calculate dx and dy
457     int dx, dy = 0;
458
459     dx = abs(x2 - x1);
460     dy = abs(y2 - y1);
461
462     if (dx < 0)
463     {
464         dx = dx * -1;
465     }
466
467     if (dy < 0)
468     {
469         dy = dy * -1;
470     }
471

```

```

471 // Draw start pixel
472 drawPixel(cVector(x1, y1));
473
474 // Initialise error variable
475 int e = 0;
476
477 // Pick a Bresenham's algorithm version
478
479 if (dx >= dy)
480 {
481     // Basic version
482
483     e = dx / 2; // Calculating error
484
485     for (int i = 0; i < dx; i++)
486     {
487         x1 += stepX;
488         // Updating error value
489         e -= dy;
490         if (e < 0)
491         {
492             y1 += stepY;
493             e += dx;
494         }
495
496         // Draw a pixel
497         drawPixel(cVector(x1, y1)); // Using drawPixel function with clipping method is too slow for drawing a line
498     }
499 }
500
501

```

```

502 else
503 {
504     // Version with changed coordinates
505
506     e = dy / 2; // Calculating error
507
508     for (int i = 0; i < dy; i++)
509     {
510         y1 += stepY;
511         e -= dx;
512         if (e < 0)
513         {
514             x1 += stepX;
515             e += dy;
516         }
517
518         graphics.setColour(200, 200, 0);
519
520         drawPixel(cVector(x1, y1));
521     }
522 }
523
524

```

Fig 3. The algorithm of Bresenham is used to draw a line. The coordinates for each pixel are established by the algorithm due to they can be displayed.

```

553 // Draw B Spline Curve by taking for control points and the resolution
554 // The resolution is the number of points draw on the screen
555 void drawBSplineCurve(cVector &p0, cVector &p1, cVector &p2, cVector &p3, int resolution)
556 {
557     // Draw the line connecting all control points
558     graphics.setColour(200, 200, 0);
559     drawLine(p0, p1);
560     drawLine(p1, p2);
561     drawLine(p2, p3);
562
563     // Normalising the resolution by finding the step value
564     double u = 1.00 / resolution;
565
566     // Since the step value "u" is known we can adjust the weights for the given points
567     for (float i = 0; i <= resolution; i++)
568     {
569         if (i == 0)
570         {
571             uValues.push_back(0); // The very first value is always 0
572         }
573         else
574         {
575             uValues.push_back(uValues.at(i - 1) + u);
576
577             if (uValues.at(i) > 1)
578             {
579                 uValues.at(i) = 1; // The very last value is always 1
580             }
581         }
582
583         //cout << "u: " << uValues.at(i) << endl;
584     }
585
586     // Calculate the weights for each step
587     for (int i = 0; i < uValues.size(); i++)
588     {
589         // Equation for the first control point
590         float x1 = (p0.x * ((pow(1 - uValues.at(i), 3))) / 6);
591         float y1 = (p0.y * ((pow(1 - uValues.at(i), 3))) / 6);
592
593         // Equation for the second control point
594         float x2 = (p1.x * ((3 * (pow(uValues.at(i), 3))) - (6 * pow(uValues.at(i), 2)) + 4)) / 6);
595         float y2 = (p1.y * ((3 * (pow(uValues.at(i), 3))) - (6 * pow(uValues.at(i), 2)) + 4)) / 6);
596
597         // Equation for the third control point
598         float x3 = (p2.x * ((3 * (pow(uValues.at(i), 3))) + (3 * pow(uValues.at(i), 2)) + (3 * uValues.at(i) + 1)) / 6);
599         float y3 = (p2.y * ((3 * (pow(uValues.at(i), 3))) + (3 * pow(uValues.at(i), 2)) + (3 * uValues.at(i) + 1)) / 6);
600
601         // Equation for the fourth control point
602         float x4 = (p3.x * ((pow(uValues.at(i), 3))) / 6);
603         float y4 = (p3.y * ((pow(uValues.at(i), 3))) / 6);
604
605         // Get the result
606         float x = x1 + x2 + x3 + x4;
607         float y = y1 + y2 + y3 + y4;
608
609         // Draw the result
610         graphics.setColour(25, 0, 0);
611         drawPixel(cVector(x, y));
612     }
613 }

```

Fig 4. The drawBSplineCurve function is responsible for rendering a B-Spline curve basing on the given control points and the resolution.

The drawing of B-Spline curve is generated by providing the control points positions and the desired resolution. Control points are required to calculate the weights, which indicate how much one point on the curve is affected by the control points. Furthermore, the resolution is a number of steps – the points on the segment of the curve, needed to be found out and to be printed. Accordingly, the smoothness of the curve is dependent on the number of calculated points belonging to the segment.

The basis functions are multiplied by four control points as follows:

$$\begin{pmatrix} B_1(t) & B_2(t) & B_3(t) & B_4(t) \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{pmatrix}$$

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

Fig 5. This is the equation for the curve where t is in the range of 0 to 1.

```
// Equation for the first control point
float x1 = (p0.x * ((pow(1 - uValues->at(i), 3)))) / 6;
float y1 = (p0.y * ((pow(1 - uValues->at(i), 3)))) / 6;

// Equation for the second control point
float x2 = (p1.x * ((3 * (pow(uValues->at(i), 3))) - (6 * pow(uValues->at(i), 2)) + 4)) / 6;
float y2 = (p1.y * ((3 * (pow(uValues->at(i), 3))) - (6 * pow(uValues->at(i), 2)) + 4)) / 6;

// Equation for the third control point
float x3 = (p2.x * ((-3 * (pow(uValues->at(i), 3))) + (3 * pow(uValues->at(i), 2)) + (3 * uValues->at(i)) + 1)) / 6;
float y3 = (p2.y * ((-3 * (pow(uValues->at(i), 3))) + (3 * pow(uValues->at(i), 2)) + (3 * uValues->at(i)) + 1)) / 6;

// Equation for the fourth control point
float x4 = (p3.x * ((pow(uValues->at(i), 3)))) / 6;
float y4 = (p3.y * ((pow(uValues->at(i), 3)))) / 6;
```

Fig 6. The example of incorporating the equations into the program.

For instance, there are four control points and a resolution as follow:

P1(200, 200),

P2(350, 250),

P3(120, 80),

P4(480, 330)

resolution = 100.

The data is passed to the drawBSplineCurve function so that the calculation can proceed.

Firstly, the resolution is taken and normalised to get the value of the step, which is in range 0 to 1.

$$u = 1.00 / \text{resolution}$$

$$u = 1.00 / 100$$

$$u = 0.01$$

Since the step value “u” is known, it can be stored for each point on the segment of the curve, remembering that each point has a unique value between 0 and 1.

```
for (int i = 0; i <= resolution; i++)
{
    if (i == 0)
    {
        uValues->push_back(0); // The very first value is always 0
    }
    else
    {
        uValues->push_back(uValues->at(i - 1) + u);

        if (uValues->at(i) > 1)
        {
            uValues->at(i) = 1; // The very last value is always 1
        }
    }
}
```

Accordingly, obtaining the weights for each step regarding the control points is required.

For each step stored in uValues:

$$x1 = \frac{1}{6} \times ((1 - u)^3 \times P1.x)$$

$$y1 = \frac{1}{6} \times ((1 - u)^3 \times P1.y)$$

$$x2 = \frac{1}{6} \times ((3 \times u^3 - 6 \times u^2 + 4) \times P2.x)$$

$$y2 = \frac{1}{6} \times ((3 \times u^3 - 6 \times u^2 + 4) \times P2.y)$$

$$x3 = \frac{1}{6} \times ((-3 \times u^3 + 3 \times u^2 + 3 \times u + 1) \times P3.x)$$

$$y3 = \frac{1}{6} \times ((-3 \times u^3 + 3 \times u^2 + 3 \times u + 1) \times P3.y)$$

$$x4 = \frac{1}{6} \times ((u^3) \times P4.x)$$

$$x4 = \frac{1}{6} \times ((u^3) \times P4.y)$$

The calculations for the given control points and the step value of 0.5 are:

First basis function

$$x1 = \frac{1}{6} \times ((1 - 0.5)^3 \times 200)$$

$$x1 = \frac{1}{6} \times 0.125 \times 200$$

$$x1 = 4.166$$

$$y1 = \frac{1}{6} \times ((1 - 0.5)^3 \times 200)$$

$$y1 = \frac{1}{6} \times 0.125 \times 200$$

$$y1 = 4.166$$

Second basis function

$$x2 = \frac{1}{6} \times ((3 \times 0.5^3 - 6 \times 0.5^2 + 4) \times 350)$$

$$x2 = \frac{1}{6} \times ((3 \times 0.125 - 6 \times 0.25 + 4) \times 350)$$

$$x2 = \frac{1}{6} \times ((0.375 - 1.5 + 4) \times 350)$$

$$x_2 = \frac{1}{6} \times (2.875 \times 350)$$

$$x_2 = 167.7083$$

$$y_2 = \frac{1}{6} \times ((3 \times 0.5^3 - 6 \times 0.5^2 + 4) \times 250)$$

$$y_2 = \frac{1}{6} \times ((3 \times 0.125 - 6 \times 0.25 + 4) \times 250)$$

$$y_2 = \frac{1}{6} \times ((0.375 - 1.5 + 4) \times 250)$$

$$y_2 = \frac{1}{6} \times (2.875 \times 250)$$

$$y_2 = 119.7916$$

Third basis function

$$x_3 = \frac{1}{6} \times ((-3 \times 0.5^3 + 3 \times 0.5^2 + 3 \times 0.5 + 1) \times 120)$$

$$x_3 = \frac{1}{6} \times ((-0.375 + 0.75 + 1.5 + 1) \times 120)$$

$$x_3 = \frac{1}{6} \times (2.875 \times 120)$$

$$x_3 = 57.5$$

$$y_3 = \frac{1}{6} \times ((-3 \times 0.5^3 + 3 \times 0.5^2 + 3 \times 0.5 + 1) \times 80)$$

$$y_3 = \frac{1}{6} \times ((-0.375 + 0.75 + 1.5 + 1) \times 80)$$

$$y_3 = \frac{1}{6} \times (2.875 \times 80)$$

$$y_3 = 38.33$$

Forth basis function

$$x_4 = \frac{1}{6} \times ((0.5^3) \times 480)$$

$$x_4 = \frac{1}{6} \times (0.125 \times 480)$$

$$x_4 = 10$$

$$y_4 = \frac{1}{6} \times ((0.5^3) \times 330)$$

$$y_4 = \frac{1}{6} \times (0.125 \times 330)$$

$$y_4 = 6.875$$

The point on the segment of the curve is given by summing up the results:

$$x = x_1 + x_2 + x_3 + x_4$$

$$x = 4.166 + 167.7083 + 57.5 + 10$$

$$x = 239.3743$$

$$y = y_1 + y_2 + y_3 + y_4$$

$$y = 4.166 + 119.7916 + 38.33 + 6.875$$

$$y = 169.1626$$

The coordinates of the points are represented as integer values in the program. Hence, the values must be converted into integers.

$$x = 239$$

$$y = 169$$

Therefore, the point is printed at position (239, 169).

P(239, 169)

Print(P(238,169))

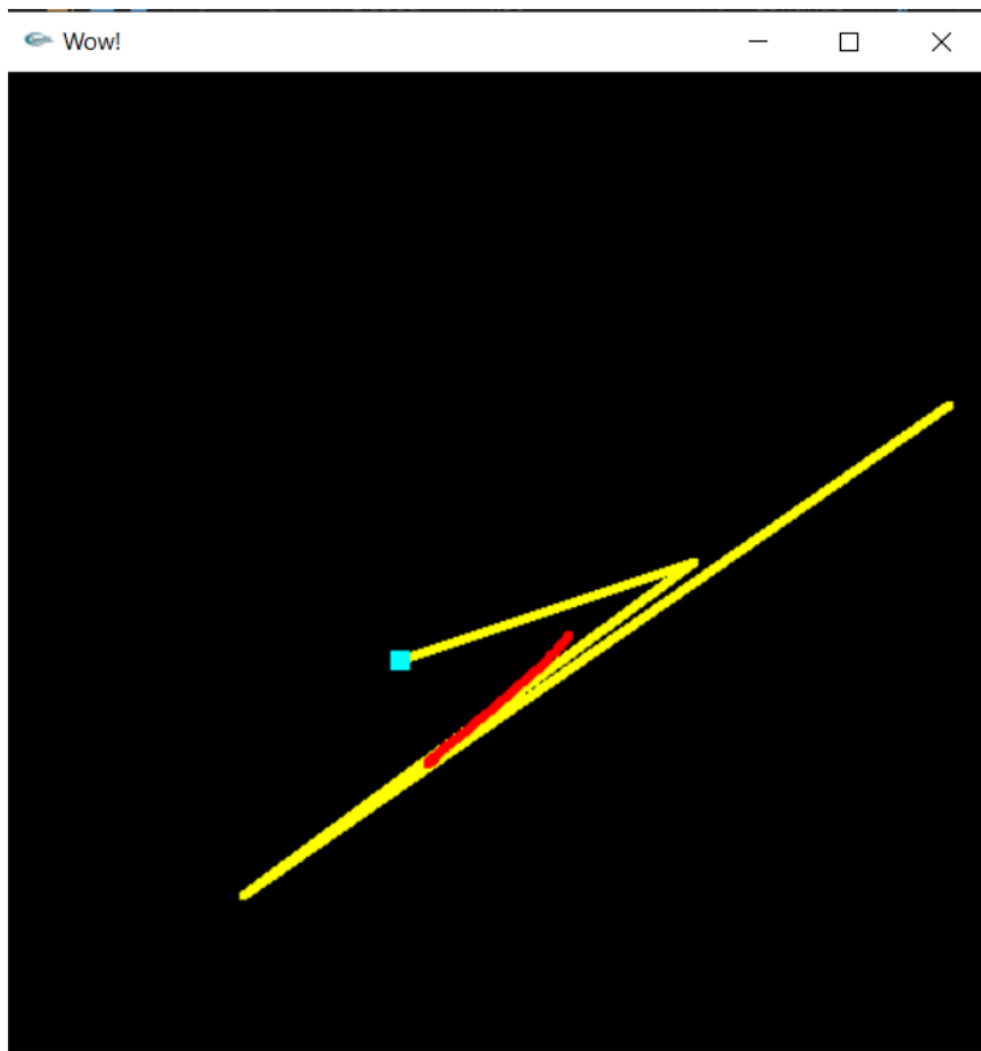


Fig 7. The created curve for the given control points and the resolution is displayed in red.

5.2 CURVE MANIPULATION SYSTEM

The designed curve manipulation system is based on changing the position of the control points during program run-time. This is due to using the functions included in the OpenGL library. The used functions are:

`void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y))` - a keyboard call back is generated after pressing an ASCII key.

`void glutSpecialFunc(void (*func)(int key, int x, int y))` - a call back is generated after pressing a non-keyboard ASCII key.

*func – the name of the function that will process the keyboards events.

```
30 void processNormalKeys(unsigned char key, int x, int y);
31 void processSpecialKeys(int key, int x, int y);

101 void update()
102 {
103     glutKeyboardFunc(processNormalKeys); // Process input from the keyboard
104     glutSpecialFunc(processSpecialKeys); // Process input from the keyboard
105 }
```

Fig 8. Declaration and calling the functions responsible for manipulating the control points.

```

139 // This function uses glut function to read input from the keyboard
140 // Proceed letters to set the control point
141 void processNormalKeys(unsigned char key, int x, int y) {
142
143     // Press "e" to exit
144     if (key == 'e')
145     {
146         exit(0);
147     }
148
149     // Press "w" to increment value of y by 1
150     if (key == 'w')
151     {
152         controlPoints[activeControlPointIndex].y += 1;
153         cout << "Control Point " << activeControlPointIndex << ": " << endl;
154         cout << "x" << activeControlPointIndex << " = " << controlPoints[activeControlPointIndex].x << endl;
155         cout << "y" << activeControlPointIndex << " = " << controlPoints[activeControlPointIndex].y << endl;
156
157         if (controlPoints[activeControlPointIndex].y > SCREEN_HEIGHT)
158         {
159             controlPoints[activeControlPointIndex].y = 500;
160         }
161     }
162 }

```

```

164 // Press "s" to decrement value of y by 1
165 if (key == 's')
166 {
167     controlPoints[activeControlPointIndex].y -= 1;
168     cout << "Control Point " << activeControlPointIndex << ": " << endl;
169     cout << "x" << activeControlPointIndex << " = " << controlPoints[activeControlPointIndex].x << endl;
170     cout << "y" << activeControlPointIndex << " = " << controlPoints[activeControlPointIndex].y << endl;
171     if (controlPoints[activeControlPointIndex].y < 0)
172     {
173         controlPoints[activeControlPointIndex].y = 0;
174     }
175 }
176
177 // Press "a" to decrement value of x by 1
178 if (key == 'a')
179 {
180     controlPoints[activeControlPointIndex].x -= 1;
181     cout << "Control Point " << activeControlPointIndex << ": " << endl;
182     cout << "x" << activeControlPointIndex << " = " << controlPoints[activeControlPointIndex].x << endl;
183     cout << "y" << activeControlPointIndex << " = " << controlPoints[activeControlPointIndex].y << endl;
184     if (controlPoints[activeControlPointIndex].x < 0)
185     {
186         controlPoints[activeControlPointIndex].x = 0;
187     }
188 }
189
190 //cout << activeControlPointIndex << endl;
191 }

```

```

193 // Press "d" to increment value of x by 1
194 if (key == 'd')
195 {
196     controlPoints[activeControlPointIndex].x += 1;
197     cout << "Control Point " << activeControlPointIndex << ": " << endl;
198     cout << "x" << activeControlPointIndex << " = " << controlPoints[activeControlPointIndex].x << endl;
199     cout << "y" << activeControlPointIndex << " = " << controlPoints[activeControlPointIndex].y << endl;
200     if (controlPoints[activeControlPointIndex].x > SCREEN_WIDTH)
201     {
202         controlPoints[activeControlPointIndex].x = 500;
203     }
204 }
205
206 //cout << activeControlPointIndex << endl;
207 }
208 }

```

Fig 9. processNormalKeys function implementation.

Changing a position of the control point is allowed due to using processNormalKeys function. Pressing “w” or “s” keys increment or decrements coordinates of a control point along the y-axis whereas pressing “a” or “d” adjust them along the x-axis.

```
210 // This function uses glut function to read input from the keyboard
211 // Proceed the special keys to choose the control point
212 void processSpecialKeys(int key, int x, int y)
213 {
214
215     switch (key) {
216     case GLUT_KEY_LEFT:
217         activeControlPointIndex -= 1;
218
219         if (activeControlPointIndex < 0)
220         {
221             activeControlPointIndex = controlPoints->size() - 1;
222         }
223         break;
224     case GLUT_KEY_RIGHT:
225         activeControlPointIndex += 1;
226         if (activeControlPointIndex > controlPoints->size() - 1)
227         {
228             activeControlPointIndex = 0;
229         }
230         break;
231     }
232 }
233 }
```

Fig 10. processSpecialKeys implementation.

Moreover, the currently setting control point can be swapped to another one by a user after pressing the left arrow or right arrow keys.

```
*****
*****
Run the program.

*****
*****

Instruction of use:
- Press page left arrow to choose the previous control point.
- Press page right arrow to choose the next control point.
- Press w to increment the value of y for the control point by 1.
- Press s to decrement the value of y for the control point by 1.
- Press a to decrement the value of x for the control point by 1.
- Press d to increment the value of x for the control point by 1.
- Press e to exit the application.
```

Fig 11. The instruction of the curve manipulation is displayed in the console window hen the program is started.

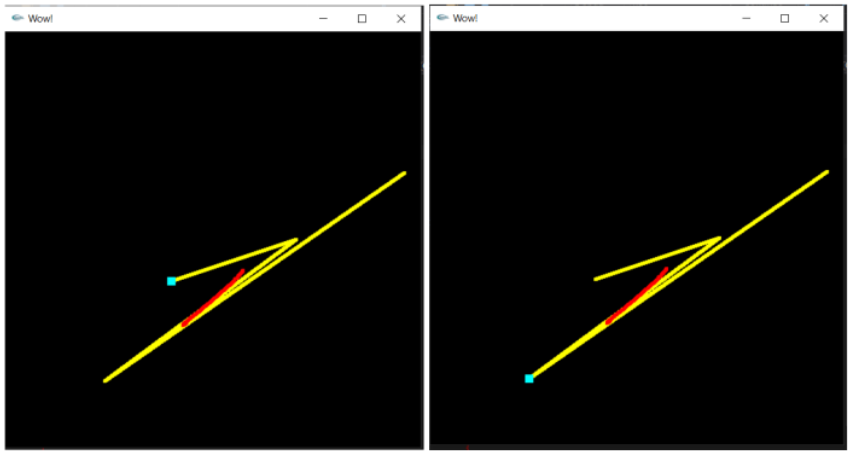


Fig 12. The currently manipulated control point is highlighted in the form of a blue square. These screenshots are showing a swapping process between the control points.

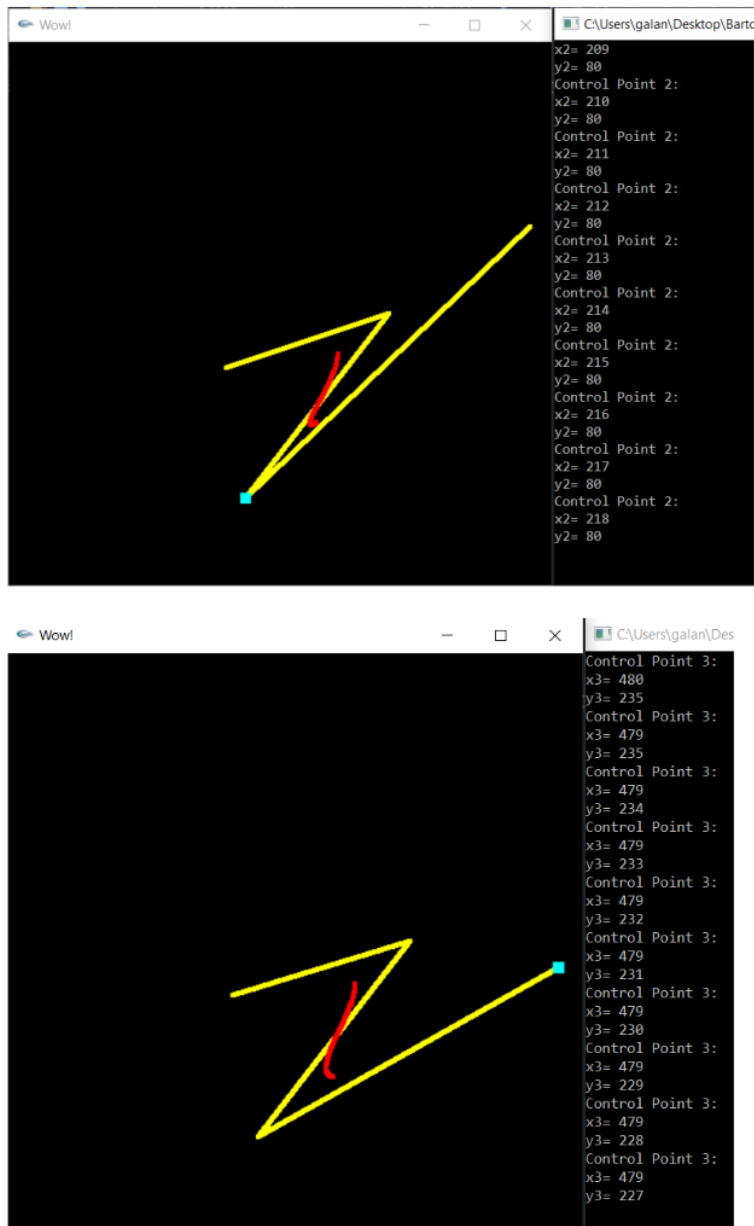


Fig 13. If the control point is moved by pressing “a”, “s”, “w” or “d” a user is informed about the results of the changes. The data regarding currently manipulated control point - its name and new coordinates - are printed in the console window.

6 CONCLUSION

The ability of the program to draw a Uniform Non-Rational Curve and the manipulation of its control points have been demonstrated. Furthermore, due to conducted research and studies, the knowledge about parametric curves, their characteristics and usage in computer graphics have been obtained. Nevertheless, the created design is limited to only four control points and a single curve segment.

7 BIBLIOGRAPHY

- [1] A. v. D. S. .. F. J. F. H. James D. Foley, Computer Graphics Principles and Practice, 2nd ed., P. S. Gordon, Ed., Addison-Wesley Publishing Company, 1996.
- [2] R. S. W. J. N. H. Graham Sellers, OpenGL Siperbible: Comprehensive Tutorial and Reference, 7th ed., Pearson Education Inc., 2016.
- [3] H. J. Hamilton, "Uniform Cubic B-Spline Curves," University of Regina, [Online]. Available: <http://www2.cs.uregina.ca/~anima/408/Notes/Interpolation/UniformBSpline.htm>. [Accessed 11 2020].

8 APPENDIX

8.1 README BARTOSZ GALANTOWICZ

The ReadMe Bartosz Galantowicz.txt document provides the most important function and instructions on how to use the program.

```
=====
=====
Parametric Curves - ASSIGNMENT PROJECT
=====
=====
```

This program uses its own designed graphics procedure to generate Uniform Non-Rational B-Spline Curves.

The application allows for rendering of the parametric curve by providing the values of control points.

Moreover, it is possible to manipulate the control points during run time.

```
=====
=====
HOW TO USE THE PROGRAM
=====
=====
```

1. The application is opened in the console window.
2. User is asked to provide values for 4 control points, respectively for x and y coordinates.
3. User is asked to enter the resolution of the curve. Large values make a curve more smooth.
4. The menu is displayed. User can choose one of these operations:
 - Chose 1 to start the simulation.
 - Chose 2 to change the coordinates of the all control points.
 - Chose 3 to change the coordinates of the one control point.
 - Chose 4 to change the resolution of the curve.
 - Chose 5 to exit the program.
5. After the simulation started the instruction of use is displayed. This help to manipulate the control points.
 - User can:
 - Press page left arrow to choose the previous control point.
 - Press page right arrow to choose the next control point.
 - Press "w" to increment the value of y for the control point by 1.

- Press "s" to decrement the value of y for the control point by 1.
- Press "a" to decrement the value of x for the control point by 1.
- Press "d" to increment the value of x for the control point by 1.
- Press "e" to exit the application.

```
=====
=====
Math Library namespace
=====
=====
```

```
nMathLibrary.h
nMathLibrary.cpp
```

This is the main mathematical library. It contains vector and matrix classes and relevant mathematical functions.

It is used to solve computations in computer graphics.

cVector class:

void setVector(double xX, double yY, double zZ = 1) - Setting the values for vector

void displayVector() - Display vector to console window

int returnDistanceToPoint(cVector &p2) - Return the distance between points

void multiplicationMatrixVector(cMatrix3D &m1) - Matrix * Vertex - return product of matrix and vector multiplication

void translatePoint2D(float translateX, float translateY) - Translate point using coordinates

void translatePoint2D(cVector &v) - Translate point using vector

void scalePoint2D(float scaleX, float scaleY) - Scale point using coordinates (Non-uniform scaling)

void scalePoint2D(cVector &v) - Scale point using vector (Non-uniform scaling)

void scaleUniformPoint2D(float factor) - Scale point using factor (Uniform scaling)

void rotatePoint2DAboutZ(double degrees) - Rotate point about z axis

`void rotatePoint2DAboutX(double degrees) - Rotate point about x axis`

`void rotatePoint2DAboutY(double degrees) - Rotate point about y axis`

`cMatrix class:`

`void setMatrixEntry(size_t row, size_t column, double value) - Setting the specific entry for the given matrix`

`void createIdentityMatrix() - Identity matrix - return identity matrix`

`void multiplicationMatrixMatrix(cMatrix3D & m2) - Matrix * matrix - return product of matrix and matrix multiplication`

`void displayMatrix() - Display matrix in the console window`

`Lookup Tables functions:`

`void fillLookupTables() - Filling lookup tables - the function is called at the beginning of the program in GraphicsTemplate.cpp file`

`double getSin(double degrees) - Return sinus of x degrees`

`double getCos(double degrees) - Return cosinus of x degrees`

`double getTan(double degrees) - Return tangent of x degrees`

`void displayLookupTable() - Draft function using to display the lookup tables`

`Conversion functions:`

`double toRadians(double degrees) - Convert degrees to radians`

`double toDegrees(double radians) - Convert radians to degrees`

```
=====
GraphicsTemplate
=====
```

This is the main application source file. It is responsible for performing the simulation of the prototype mechanics.
Some new functions are described below.

void displayMenu() - Display menu

void choseMenuOptions(int userChoice) - Chose menu options

void drawLine(cVector &point1, cVector &point2) - Bresenham's line drawing algorithm

void pointClipping(cVector &point) - Point clipping algorithm

void drawPixel(cVector &point) - Check if point is not clipped and draw a pixel

void drawBSplineCurve(cVector &p0, cVector &p1, cVector &p2, cVector &p3, int resolution) - Draw a B-Spline curve

int getResolution() - Return the resolution of the curve

Bartosz Galantowicz - Graphics and Programming (ACCA6021)

- Parametric Curves

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

Instructor

71 /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

GRADING FORM: SMU GENERIC_GJ

BARTOSZ GALANTOWICZ

71

RESEARCH-INFORMED LITERATURE



Refs included. Relevant and appropriate but please refer to them in the body of your report when you assert something or to support an argument for example.

6

KNOWLEDGE AND UNDERSTANDING



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

15

ANALYSIS



A good interpretation of the problem and synthesis of a working solution. Additional control points would have made this submission even better.

14

PRACTICAL APPLICATION AND DEP.



Some 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 even better.

28

SKILLS FOR PROFESSIONAL PRAC.



A very good report. See comments earlier regarding references. Report structured. Some good SE principles and practice in evidence here.

8