02561 Computer Graphics

Drawing curves

Author:
Grigoras, Miruna Nicoleta (s213288)

Contents

1. Introduction

"Vector representations are a resolution-independent means of specifying shape. They have the advantage that at any scale, content can be displayed without tessellation or sampling artifacts." [1]. Therefore, in the context of computer graphics, when trying to display different shapes and objects that should recreate real life objects, vectorial representations are the most useful ones as they don't depend on fixed measurements and are able to create the desired shape though the usage of certain mathematical algorithms.

One of the most basic and easy to use algorithms in order to represent various curves is given through the implementation of parametric curves through the Bézier approach. This is based on rendering triangles and triangular approximations in order to define and smooth an object's curves. The Bézier curves are based on a set of control points which will determine the shape of the future curve. The most common types of curves are the linear, quadratic and cubic ones which have two, three, respectively four control points. Even so, this project shows the implementation of Bezier curves with a varying number of control points, starting with a minimum of two (which are the basics for drawing a line) and going up to however many the user would like to input. The mathematical algorithm has been generalized and will be presented in the further chapters.

2. Method

Firstly, we should concentrate on what exactly is a parametric curve. "There are many situations where we don't know the exact position that an object will have at a given time, but we know an equation that describe its movement. These equations are known as parametric curves and are called like that because the position depends on one parameter: the time." [2]

The Bézier curves are just a subcategory of parametric curves. "A Bézier curve is defined by a set of control points $\mathbf{P}_0$ through $\mathbf{P}_n$, where n is called the order of the curve. The first and last control points are always the endpoints of the curve; however, the intermediate control points (if any) generally do not lie on the curve. The sums in the following sections are to be understood as affine combinations – that is, the coefficients sum to 1." [3]

As mentioned above, there are three most common Bézier curves: the linear, quadratic and cubic ones.

a. The linear curve

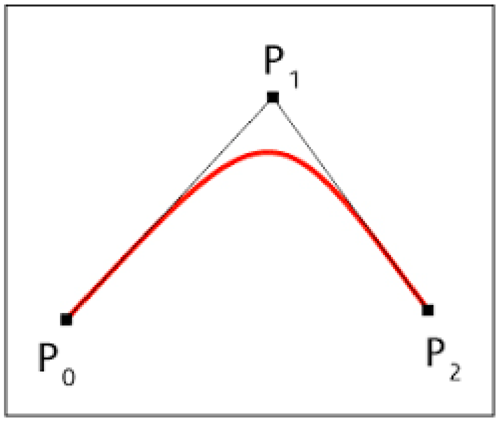
It requires only two control points: P_0 and P_1, practically describing a straight line between two points. The function for such a curve, which is equivalent to linear interpolation, can be described through the equation:

$$\mathbf{B}(t) = \mathbf{P}_0 + t(\mathbf{P}_1 - \mathbf{P}_0) = (1-t)\mathbf{P}_0 + t\mathbf{P}_1,\ 0 \leq t \leq 1$$

b. The quadratic curve

This curve requires three control points: P_0, P_1 and P_2 and can be interpreted as the linear interpolant of corresponding points on the linear Bézier curves from P_0 to P_1 and from P_1 to P_2 respectively. [3]

$$\mathbf{B}(t) = (1-t)^2\mathbf{P}_0 + 2(1-t)t\mathbf{P}_1 + t^2\mathbf{P}_2,\ 0 \le t \le 1.$$

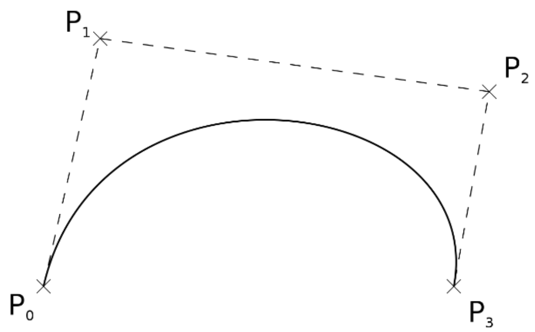


A.

c. The cubic curve

The cubic curve is defined through four control points, the equation corresponding to it being:

$$\mathbf{B}(t) = (1-t)^3\mathbf{P}_0 + 3(1-t)^2t\mathbf{P}_1 + 3(1-t)t^2\mathbf{P}_2 + t^3\mathbf{P}_3,\ 0 \le t \le 1.$$



B.

d. Generalized formula

Given the previously shown basic curves, a general formula can be deduced. Therefore, in case of n control points, the general equation which has been implemented in the project is:

$$\mathbf{B}(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i$$
$$= (1-t)^n \mathbf{P}_0 + \binom{n}{1} (1-t)^{n-1} t \mathbf{P}_1 + \cdots + \binom{n}{n-1} (1-t) t^{n-1} \mathbf{P}_{n-1} + t^n \mathbf{P}_n, \qquad 0 \leqslant t \leqslant 1$$

where $\binom{n}{i}$ are the binomial coefficients given by the formula: [4]

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

3. Implementation

The project's implementation evolves around the implementation of the generalised formula presented above. The first thing to achieve is calculating the binomial coefficients, which depend on the factorial function given below:

```
    function factorial(x){
        var fact = 1;
        for(var i=1; i <= x; i++)
            fact = fact * i;
        return fact;
    }
```

After gaining access to the factorial method, we can approach creating the curve, the implementation relies on the array called *pointsArray* which will store all the points that are to be drawn for the curve. In this implementation, each curve will be drawn with a fix number of points given by the variable:

```
    var maxNumCurvePoints = 300;
```

The *numControlPoints* parameter represents the number of control points that are to be given for this particular curve.

For every of the 300 points that are to be drawn, we need to apply the generalized formula, as each point out of the 300 stands for a certain "position in time" in the creation of the curve starting with the origin point and reaching the ending point.

```
    function getBezierCurve(pointsArray, numControlPoints){
        var curvePoints = [];
        for(var i=1; i <= maxNumCurvePoints; i++){
            var step = i / maxNumCurvePoints;
            var px = 0;
            var py = 0;
            for(var j=0; j < numControlPoints; j++){
                var binCoeff = factorial(numControlPoints-1) / (factorial(j) *
factorial(numControlPoints - j - 1));
                px += pointsArray[pointsArray.length-1-j][0]
                    * Math.pow(step, numControlPoints - j - 1)
                    * Math.pow((1-step), j)
                    * binCoeff;
                py += pointsArray[pointsArray.length-1-j][1]
                    * Math.pow(step, numControlPoints - j - 1)
                    * Math.pow((1-step), j)
                    * binCoeff;
            }
            curvePoints.push(vec2(px, py));
        }
        return curvePoints;
    }
```

One intriguing thing that can be seen is that the algorithm uses the *pointsArray*'s content starting with the last added point and going back to the *numControlPoints*-th point. The reason for this is that the *pointsArray* will keep stored all the points that have been drawn on the canvas, not just the points corresponding to one single curve.

Now, every time the user click on the canvas, there are two wanted actions to be performed: either draw a point or finalize the drawing of a curve. Therefore, the next lines of code have been written:

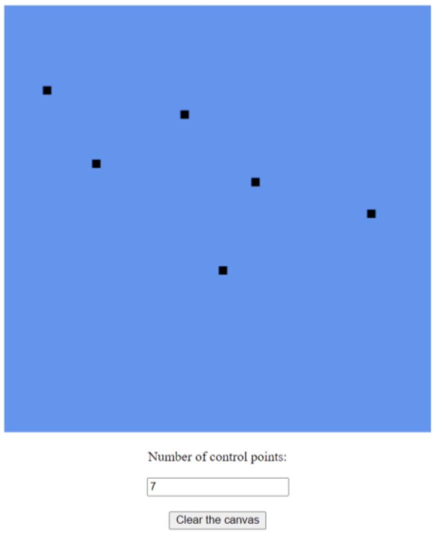
```
canvas.addEventListener("click", function () {
        var domThingy = event.target.getBoundingClientRect() ;
        var newPoint = vec2(-1 + 2 * (event.clientX -
domThingy.left)/canvas.width,
                    -1 + 2*(canvas.height - event.clientY + domThingy.top-
1)/canvas.height);
        numPointsCurve++;
        pointsArray.push(newPoint);

        if (numPointsCurve == numControlPoints) {
            numPointsCurve = 0;
            var curve = getBezierCurve(pointsArray, numControlPoints);
            for (var i = 0; i < numControlPoints; i++)
                pointsArray.pop();
            curve.forEach(p => {
                pointsArray.push(p);
            });
        }
    });
```

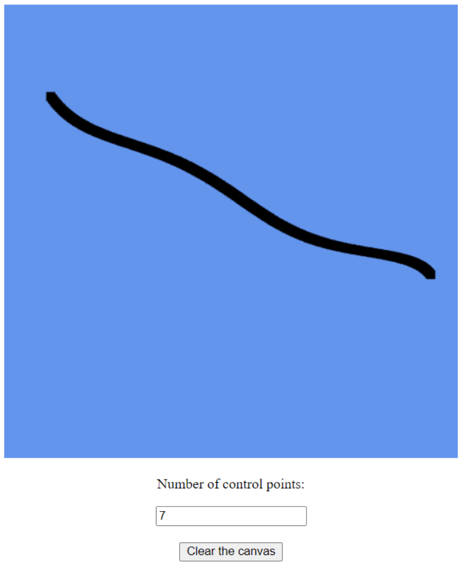
Therefore, after the *newPoint*'s coordinates have been saved, the *numPointsCurve* counter increases. This is a counter which is reinitialized every time a complete curve has been drawn, the user presses the "Clear Canvas" button or whenever the user changes the number of control points they which to have for their future curve. Its purpose is to keep count of how many points have been drawn up until the point in which the desired number of control points has been reached. When it has been reached, the curve points can be generated, the control points will be removed from the *pointsArray* as to not be displayed and the newly generated 300 points will be added afterwards.

4. Results

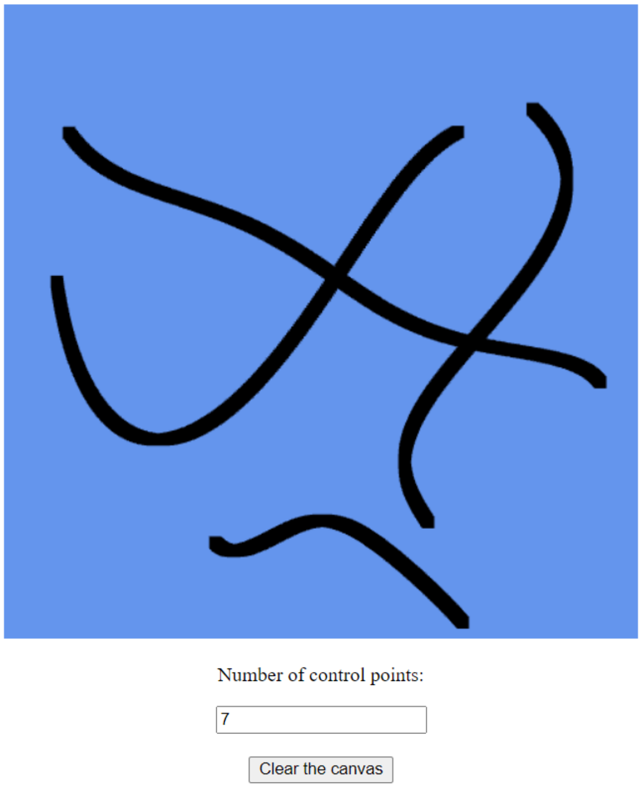
The process of drawing a 7-control-points curve inside the tool can be seen below. First, we draw the points:



After the first six points have been drawn, when the seventh one is put on canvas, the corresponding curve will appear:



After drawing the first curve, multiple curves can be drawn, like in the example below:



As it can be seen, the created curve might not have the wanted accuracy, case in which the Bézier curves might prove to be not the best existing solution.

Of course, there are a lot of improvements to be added to the project. One of them is related to how the curve is currently drawn.

The implementation approaches the drawing of the curve through the drawing of a fixed number of points. As expected, this can be modified and drawn with gl.TRIANGLES, gl.TRIANGLE_FAN, gl.TRIANGLE_STRIP, etc, which, in case of a higher resolution, would provide a better representation of the curve, would not risk displaying the space in-between the points in case the fixed number is not enough for a certain resolution (the effect of pixelation).

Another problem that might arise is the adding of too many control points. As this implementation relies on the factorial function being performed a number of times, for enlarged values, this might lead to a slow processing of the curve which is not a wanted result.

5. Discussion

Overall, this method of drawing curves is easy to implement and grasp in concept, but even so, it does have its disadvantages.

The Bézier -curve produced by the Bernstein basis function has limited flexibility.

- First, the number of specified polygon vertices fixes the order of the resulting polynomial which defines the curve.
- The second limiting characteristic is that the value of the blending function is nonzero for all parameter values over the entire curve. [5]

One possible alternative to the Bézier curves are the B-spile curves which would provide the following advantages:

- The degree of B-spline polynomial is independent on the number of vertices of defining polygon.
- B-spline allows the local control over the curve surface because each vertex affects the shape of a curve only over a range of parameter values where its associated basis function is nonzero.
- The curve exhibits the variation diminishing property.
- The curve generally follows the shape of defining polygon.
- Any affine transformation can be applied to the curve by applying it to the vertices of defining polygon.
- The curve line within the convex hull of its defining polygon. [5]

This whole project and all the worksheets can be accessed via this link.

6. Resources

[1] B. J. Diego Cantor, WebGL Beginner's Guide, Packt, 2012.

[2] J. B. Charles Loop, "GPU Gems 3," NVIDIA Developer, [Online]. Available: https://developer.nvidia.com/gpugems/gpugems3/part-iv-image-effects/chapter-25-rendering-vector-art-gpu. [Accessed 6 12 2021].

[3] Wikipedia, "Bézier curve," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/B%C3%A9zier_curve#Specific_cases. [Accessed 05 12 2021].

[4] "Binomial coefficient," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Binomial_coefficient. [Accessed 5 12 2021].

[5] "Computer Graphics Curves," TutorialsPoint, [Online]. Available: https://www.tutorialspoint.com/computer_graphics/computer_graphics_curves.htm. [Accessed 6 12 2021].