CSE 167 (WI 2021) Homework 3

Download the code frameworks.

In this homework, we will explore the 2D vector graphics with spline curves. You will use the de Casteljau Algorithm, and variations of it, to draw Bézier curves and B-Splines.

If you choose to do the extra credit, you would write a vector graphics visualizer, which parses a text file with control point data, and draws the curve. Produce a 2D vector line art using your vector graphics program.

3.1 Assignment Overview

In the assignment, you can run the reference solution program (a ".exe" file for Windows, and "reference-solution-linux" and "reference-solution-osx" on Linux and Mac) to see how the final code should behave. Make sure your code behaves identically to the solution (checking that detail levels match). If your code works except for slightly different behavior at level 1, don't worry about it (though it may be an indication of other errors).

Your job is to fill in the sections of `curves2.cpp` that say `/*YOUR CODE HERE*/`.

Note

The solution as well as the skeleton code requires the user to press a number (0,1,2, or 3) selecting the type of curve, before editing points through mouse. The mouse interaction includes the left-click (add points), drag (move points), and right-click (delete points). Most of the code skeleton should be fairly self explanatory.

Restriction

As always, do not use any highlevel OpenGL calls or external libraries. The one exception is where it says "make sure the scene gets redrawn." The correct line to complete this operation is `glutPostRedisplay();`
Also, do not modify any files other than `curves2.cpp` for the required part of the assignment. (You may modify other files for the bonus part.)

Submission

Submit the `curves2.cpp` file for the required part of the homework.

Extra Credit (5–10 pts)

For the extra credit, you are likely to modify other parts of the homework. Make sure you have finished the required part. Then, make a copy of the code base, so that you have more freedom of developing your own program. For this extra credit part, your program needs to read a text file specifying the coordinates of the control points (in a style similar to HW2) so that it is able to draw multiple curves of various types specified in a `.txt` text file.

There is no restriction to what the file format should be. It can be as simple as

```
# This is a comment
polyline 0.3 0.5 -0.2 -0.1 0.0 0.1
bezier 10 0.3 0.2 -0.1 0.1 0.0 0.0
```

which draws a polygonal curve connecting $(0.3, 0.5)$, $(-0.2, -0.1)$, $(0.0, 0.1)$ and a Bézier curve with level of detail 10 with control points $(0.3, 0.2, -0.1, 0.1, 0.0, 0.0)$.

The standard (and much more elaborated) file format is the `SVG` format. You can look it up and gain inspiration from `SVG`. In a sense you are writing a simplified `SVG` renderer.

With such a vector graphics renderer, you can create some 2D vector art work. You might want to change the thickness and the color of the curve so that it looks better.

Submit your final artwork via a final screenshot, the plain text file specifying the control points, a "readme" file that briefly summarizes what you have done, and the zipped up source codes.

You will earn 5 extra credits if everything is done right. You will gain more credits if the work is impressive.

A few of the best artworks will be exhibited in a gallery on the course website.

> *For the remaining part of this assignment document, we will focus on the required part of the homework (rather than the extra credit part).*

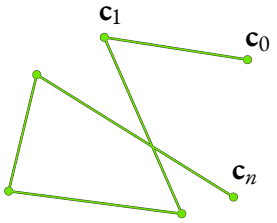
3.2 Background

In the program, the user selects $(n+1)$ points in the plane. Let us call the positions of the points $\mathbf{c}_0, \ldots, \mathbf{c}_n$, where $\mathbf{c}_i = (x_i, y_i)$. These points are called the **control points**. The program can display 4 types of curves:

0. Polygonal curve
1. Bézier curve
2. Cubic B-spline
3. Subdivision Bézier curve

3.2.0 Polygonal Curve

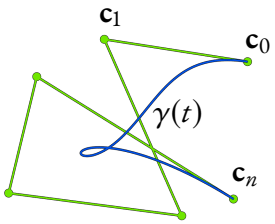
The polygonal curve connects the $(n+1)$ points $\mathbf{c}_0, \ldots, \mathbf{c}_n$ by n straight line segments. Mathematically, the curve is assembled by n parametric curves $\gamma_i : [0,1] \to \mathbb{R}^2$, $i = 0, \ldots, n-1$, where $\gamma_i(t) = (1-t)\mathbf{c}_i + t\mathbf{c}_{i+1}$. That is, $\gamma_i(t)$ linearly interpolates the consecutive points $\mathbf{c}_i$ and $\mathbf{c}_{i+1}$.



3.2.1 Bézier Curve

The Bézier curve controlled by $\mathbf{c}_0, \ldots, \mathbf{c}_n$ is a single curve (instead of a piecewise-defined curve)

$$\gamma : [0,1] \to \mathbb{R}^2, \quad \gamma(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}. \tag{1}$$



It has the following features:

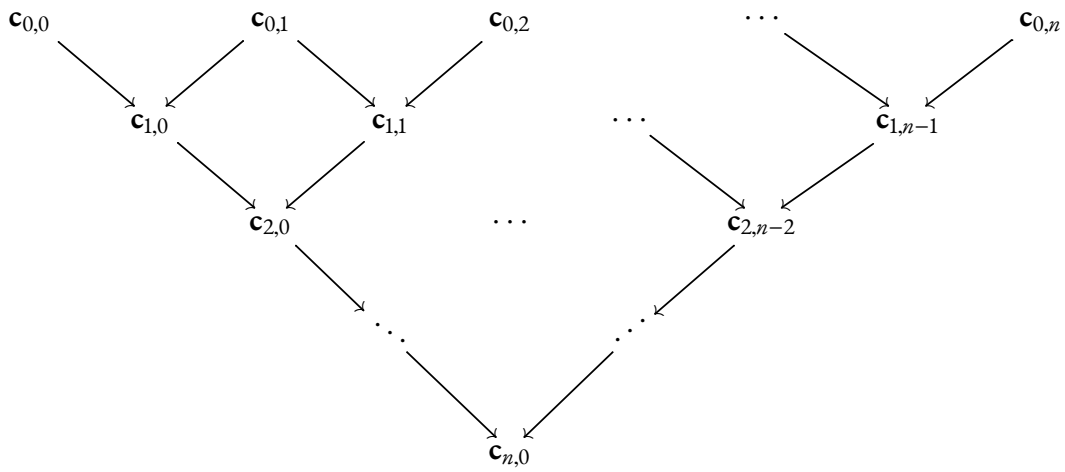
- Both $x(t)$ and $y(t)$ are n-th order polynomials in t.
- $\gamma(0) = \mathbf{c}_0$ and $\gamma(1) = \mathbf{c}_n$.
- $\gamma'(0)$ is parallel to $\mathbf{c}_1 - \mathbf{c}_0$ and $\gamma'(1)$ is parallel to $\mathbf{c}_n - \mathbf{c}_{n-1}$.

More precisely, $\gamma(t)$ is given in terms of the Bernstein polynomials by

$$\gamma(t) = \sum_{k=0}^{n} \binom{n}{k} t^k (1-t)^{n-k} \mathbf{c}_k \tag{2}$$

However, numerically evaluating the multiplications in this formula is expensive and unstable. The de Casteljau algorithm is a numerically stable and efficient way to evaluate (2). Let $\mathbf{c}_{0,k} = \mathbf{c}_k$,

$k = 0, \ldots, n$. Construct



(3)

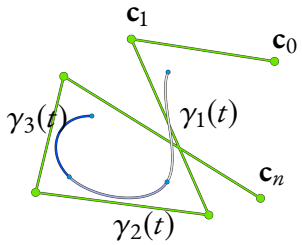
where each "funnel" is a linear (affine) interpolation

$$\mathbf{c}_{j,k} = (1-t)\mathbf{c}_{j-1,k} + t\mathbf{c}_{j-1,k+1}. \tag{4}$$

Then the final aggregated value $\mathbf{c}_{n,0}$ equals to the desired $\gamma(t)$. Note that one does not need to store all the $O(n^2)$ values of $\mathbf{c}_{j,k}$'s. One only need to use $O(n)$ memory for this calculation.

3.2.2 B-Spline

The B-spline is composed of $(n-1)$ curve segments $\gamma_1, \ldots, \gamma_{n-1}$, each of which is a cubic polynomial. The junction between γ_k and γ_{k+1} is G^2; that is, the curve is continuous with continuous tangent and continuous curvature. The curve segment γ_k is only a function of the four control points $\mathbf{c}_{k-1}, \mathbf{c}_k, \mathbf{c}_{k+1}, \mathbf{c}_{k+2}$.



For a *uniform B-spline*, the joint curve is parameterized as $\gamma : [1, n-1] \to \mathbb{R}^2$ with

$$\gamma(t) = \begin{cases} \gamma_1(t) & 1 \le t \le 2 \\ \gamma_2(t) & 2 \le t \le 3 \\ \vdots & \\ \gamma_{n-1}(t) & n-1 \le t \le n. \end{cases} = \sum_{k=0}^{n} \mathbf{c}_k B_{k,4}(t-k) \tag{5}$$

where $B_{k,4}$ is the basis piecewise cubic polynomial whose definition can be found here (https://en.wikipedia.org/wiki/B-spline#Definition). Instead of explicitly computing the rather complicated formula for B, we will use the de Casteljau algorithm and the cubic blossom. Consider $F(t_1, t_2, t_3)$ being a function on 3 scalar variables t_1, t_2, t_3 such that

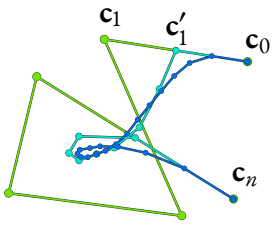
- F is symmetric: $F(t_1, t_2, t_3) = F(t_2, t_1, t_3) = F(t_1, t_3, t_2)$.
- F is tri-affine: Fixing any pair of the 3 variables, the remaining 1-variate function is affine $F(ax + by, t_2, t_3) = aF(x, t_2, t_3) + bF(y, t_2, t_3)$ $(a + b = 1)$.

Now, assign

$$F(-1,0,1) = \mathbf{c}_0, \quad F(0,1,2) = \mathbf{c}_1, \quad F(1,2,3) = \mathbf{c}_2, \quad F(2,3,4) = \mathbf{c}_3, \quad \text{etc.} \tag{6}$$

Our final desired $\gamma(t)$ is given by $F(t,t,t)$. Using the symmetry and tri-affinity of F, one can uniquely compute $\gamma(t) = F(t,t,t)$, say for $1 \le t \le 2$, using the values of $F(-1,0,1)$, $F(0,1,2)$, $F(1,2,3)$, $F(2,3,4)$. Concretely, from $F(-1,0,1) = F(0,1,-1)$ and $F(0,1,2)$ one computes $F(0,1,t)$ through affine combination. Similarly, one obtains $F(1,2,t)$ and $F(2,3,t)$ from other pairs. Then, from $F(0,1,t) = F(1,t,0)$ and $F(1,2,t) = F(1,t,2)$ one computes $F(1,t,t)$. Similarly, one gets $F(2,t,t)$. Finally, one computes $F(t,t,t)$.

3.2.3 Subdivision Bézier Curve



A spline, such as the Bézier curve, is determined from a set of control points. For many spline schemes (including Bézier) we observe that a spline from a coarse set of control points $\mathbf{c}_0, \ldots, \mathbf{c}_n$ happens to be the same as the spline coming from another finer set of control points $\mathbf{c}'_0, \ldots, \mathbf{c}'_n$ and $\mathbf{c}'_n, \ldots, \mathbf{c}'_{2n}$. The process of given $\mathbf{c}_0, \ldots, \mathbf{c}_n$ and compute $\mathbf{c}'_0, \ldots, \mathbf{c}'_{2n}$ is called subdivision. The idea of subdivision is that, instead of drawing the spline, we will just draw the control points after a few iteration of subdivisions. The subdivision points converge to the Bézier spline under multiple recursion of subdivisions.

Note that the Bézier curve can also be viewed in terms of the blossom. For $(n+1)$ control points, the function $F(t_1, \ldots, t_n)$ is a symmetric n-affine function. Letting $F(0,0,\ldots,0) = \mathbf{c}_0$, $F(0,0,\ldots,0,1) = \mathbf{c}_1$, $F(0,\ldots,0,1,1) = \mathbf{c}_2$,.., $F(1,\ldots,1) = \mathbf{c}_n$, we have the Bézier curve given by $\gamma(t) = F(t,t,\ldots,t)$ through repeated affine combinations (unfolding the de Castejau algorithm (3)). In this formalism, the subdivided points are

$$\mathbf{c}'_0 = F(0,\ldots,0,0), \quad \mathbf{c}'_1 = F(0,\ldots,0,\tfrac{1}{2}), \quad \mathbf{c}'_2 = F(0,\ldots,\tfrac{1}{2},\tfrac{1}{2}), \quad \ldots, \quad \mathbf{c}'_n = F(\tfrac{1}{2},\ldots,\tfrac{1}{2}),$$
$$\mathbf{c}'_{n+1} = F(\tfrac{1}{2},\ldots,\tfrac{1}{2},1), \quad \mathbf{c}'_{n+2} = F(\tfrac{1}{2},\ldots,1,1), \quad \ldots, \quad \mathbf{c}'_{2n} = F(1,\ldots,1).$$

Each of these $\frac{1}{2}$ points can be evaluated by a de Casteljau type averaging.

It might also be useful to compute the linear combination coefficients

$$\begin{bmatrix} \mathbf{c}'_0 \\ \mathbf{c}'_1 \\ \vdots \\ \mathbf{c}'_n \\ \vdots \\ \mathbf{c}'_{2n} \end{bmatrix} = \underbrace{\begin{bmatrix} s_{00} & \cdots & s_{0n} \\ s_{10} & \cdots & s_{1n} \\ \vdots & & \vdots \\ s_{n0} & \cdots & s_{nn} \\ \vdots & & \vdots \\ s_{2n,n} & \cdots & s_{2n,2n} \end{bmatrix}}_{\text{subdivision matrix}} \begin{bmatrix} \mathbf{c}_0 \\ \vdots \\ \mathbf{c}_n \end{bmatrix} \tag{7}$$

3.3 Specifications

In a step-by-step manner:

1. First, complete the `WorkingScene::` methods. Once you have filled in these functions correctly, you can draw polygonal curves (straight lines connecting points) by left clicking on the screen to add points. You should be able to delete points by right clicking on them. You can also drag existing points around.
2. Fill in code for `Bezier`, `Bspline` and `Bezier2`. The `draw` method for each of these curve subclasses takes an integer parameter called `levelOfDetail`. Call ℓ = `levelOfDetail` for short.

Bezier Devide the curve into ℓ segments. Hence, you need to evaluate the curve at $\ell + 1$ points, connecting these with line segments. The evaluation can be done by the deCasteljau algorithm. You can also evaluate the curve by the explicit Bernstein polynomials.

Bspline This is the cubic B-spline. The curve is composed of $(n - 1)$ pieces. Each piece γ_k is divided into ℓ straight line segments. Each piece γ_k is computed using only the 4 points $\mathbf{c}_{k-1}, \mathbf{c}_k, \mathbf{c}_{k+1}, \mathbf{c}_{k+2}$. You can either use the deCasteljau-type algorithm, or the explicit B-spline basis (e.g. in terms of the spline matrix and geometry matrix).

Bezier2 This is the subdivision Bézier curve. The level of detail ℓ represents the level of recursion. So, note that the number of points to draw grow exponentially in ℓ. Draw the curve by recursive subdivision, splitting it at its midpoint each time. You can use the deCasteljau algorithm for the subdivision.

3.4 Hints and Notes

- Note that the program (including the solution program) will not do anything if you click on control points after startup. You first need to enter the type of curve (say `0` for a basic polygonal curve). We encourage you to read the code to figure out how to switch to Bézier and Bspline curves. The `+` and `-` keys change the level of detail parameter.
- The class `Curve` has the function `drawLine()`, which you should use to draw straight lines.
- The class `Point` has a function `draw()`. For drawing knots in the B-Spline, you will want to create a point, and ask it to draw itself.
- Normalization. For `WorkingScene::drag` and `WorkingScene::mouse`, you will likely want to normalize the coordinates returned by x and y, by dividing by the window width.
- Functions in `Curve`. A number of functions of the `Curve` class will be helpful for `WorkingScene`. (Look at `Curve.cpp`.) In particular, `moveActivePoint` (useful for drag) simply updates the current active point.
- Adding and deleting points. `WorkingScene::mouse` will require code for adding and deleting points. A number of functions in the `Curve` class are helpful here. `addPoint` adds a point with specified x and y coordinates. `updateActivePoint` takes the x and y coordinates, picking the relevant point within some radius (it can thus be used for selection). `deleteActivePoint` deletes the currently active point. These functions can be useful.
- The assignment skeleton code uses the `stl::vector` class from the Standard Template Library (STL). It is a very powerful class, but may have a slight learning curve if you have not seen it before. It is worth investing the time to learn how to use this.[1] If you don't want to use the `stl::vector`, figure out just enough to convert the `vector` of points into a form you are comfortable with before you start manipulating those points.
- For `Bezier2`, you are provided with a more complete skeleton. The code uses `stl::vector`. You may choose to rewrite `draw()` and not use the skeleton at all, but we recommend that you study this code and fill in the provided structure. This will help you learn the `stl::vector` class as well as the recursive algorithm for Bézier curves.
- You can write other helper functions instead of using the given functions when imple-

[1] Microsoft's MSDN website is a good source of information. Nowadays Youtube is also a good source of information.

menting, say, `Bezier2`. You can change whatever you want in `curves2.cpp`, but don't change anything outside of that.