Report: Bezier Curves and Font

Antonella Mele
Sofia-Zoi Sotiriou

Introduction

This project aims to develop a program that creates and displays three policies for uppercase sans-serif letters using Bézier curves. Bézier curves are implemented with the de Casteljau algorithm, a recursive algorithm which performs a series of interpolations between a number of points to produce a straight line or a curve. To visualize the result we used the SDL library of C++ to create a graphical window containing the three policies we created.

Policies:

- Policy1: Drawing the outline of each letter in black on a white background on a bitmap.
- Policy2: Filling the letter in black.
- Policy3: Adding a red border to the previous version.

Approach

We started by implementing the basis of the project: the visualization (using the SDL library of C++) and creation (using the DeCasteljau algorithm) of lines. We created a basic window to visualize the results of every experiment we would consequently make and we implemented the DeCasteljau algorithm for linear and quadratic Bezier curves (since we didn't know if we would need to utilize more complex Bezier curves of degree three or higher). Then, we separated the workload into two main categories: tracing the letters and filling them.

- **Tracing**

  We started by breaking down each letter into its components, with a component being either a straight line or a curve. Each component is represented by a vector

- **Filling**

  In tracing the letters, we decided that each letter would have a width of 15px to find the balance between showcasing what was asked in the project and fitting our plans for the layout of the display window. Then, for filling the letters we thought the best policy would be to

To get a more readable result, we decided on drawing the letters in two lines for each policy, with each line containing 13 letters. We also added two dotted lines separating the different policies. And finally, for uniformity reasons, we decides that each letter should have a thickness of 15px between the inner and outer lines. We took into account also an offset in both the horizontal (20px) and vertical (90px) direction so the policies would not start exactly at the top and left edge of the window, but they would have a bit of space for better readability and uniformity.

Project overview and structure

- **main.cpp**
  The main.cpp file acts as the central entry point for the program. All the objects are created here, whether that is to create an alphabet or render it using SDL. For every alphabet the points for all the letters are created and then the class Sdl is called to handle the drawing.

- **DeCasteljau.cpp and DeCasteljau.h**
  The DeCasteljau class is a key component of the system, focused on applying the DeCasteljau algorithm to generate Bezier curves. It includes an `interpolation` function and `linearBezier`, `quadraticBezier` and `cubicBezier` functions based on the different needs of the program.

- **Sdl.cpp and Sdl.h**
  The Sdl.cpp and Sdl.h files are responsible for managing the rendering of the alphabets. It contains methods for initializing the window and renderer, drawing the alphabets and lines to separate them, as well as methods for event handling and cleanup.

- **Alphabet.h**
  The alphabet.h file defines the Alphabet class, which serves as the foundational structure for all alphabets used in the program. It is a purely virtual class, containing all the members that both Trace and Filling use, to avoid unnecessary re-initializations of variables (ex. the variable results is used every time a letter is formed and is common to both the Filling and Trace classes). Each alphabet is represented by a collection of points and lines stored in a public three-dimensional vector (letters) that the main function uses to draw each alphabet.

  The vector contains two-dimensional vectors which each represent a letter to be drawn and each "letter" contains all the points it needs. In other parts of the project we further separate the letters in the lines and curves that comprise it, but for this part we judged that it would be an addition that would serve only to burden the memory and not offer any clarity at all, so each letter is just a collection of points. The Alphabet class includes a pure virtual function, `generateAlphabet(int x)`, which is implemented in the derived classes Trace and Filling in different ways based on the specific needs of each class. One thing that is common to both implementations is that the parameter x defines which alphabet is going to be drawn (for trace that is between the first and third alphabet and for filling it is between the second and third).

- **Trace.cpp and Trace.h**
  The Trace class, declared in trace.h and defined in trace.cpp, extends the Alphabet class and provides a specific implementation for representing the alphabet with just the outlines. It does so by using arrays of three-dimensional vectors, `vector<vector<vector<double>>>`. Each of these

vectors represents a letter of the alphabet (either from the first alphabet or the third) and contains two dimensional vectors which represent each line or curve that the letter contains. Each two dimensional vector (line/curve) contains vectors of doubles (which all have just two elements, the double representing the X coordinate and the double representing the Y coordinate of a point), which are the points that the we want to implement the DeCasteljau algorithm on to create that line or curve.

The `generateAlphabet(int x)` function uses a vector of pointers to go through all the letter vectors of the alphabet we want to draw (which is determined by the parameter the function has when called by the main function), calling the `generateLetter` function and adjusting the offsets as necessary for each letter.

Then, for each letter the
`generateLetter(double offsetX, double offsetY)` function is called. In it, the program iterates through all the lines of the letter (the two dimensional vectors) and based on the length of each vector (2 or 4 for linear and cubic Bezier respectively) it performs the DeCasteljau algorithm and stores the resulting points of the line or curve produced in a vector called letters (a member of the class Alphabet, since it is common to both Trace and Filling classes, which is emptied after each alphabet to avoid unnecessary data in the memory).

- **Filling.cpp and Filling.h**
  The Filling.h and Filling.cpp files define the Filling class, which is a subclass of the Alphabet class. This class produces the filled-in version of the alphabet for the second and third font.

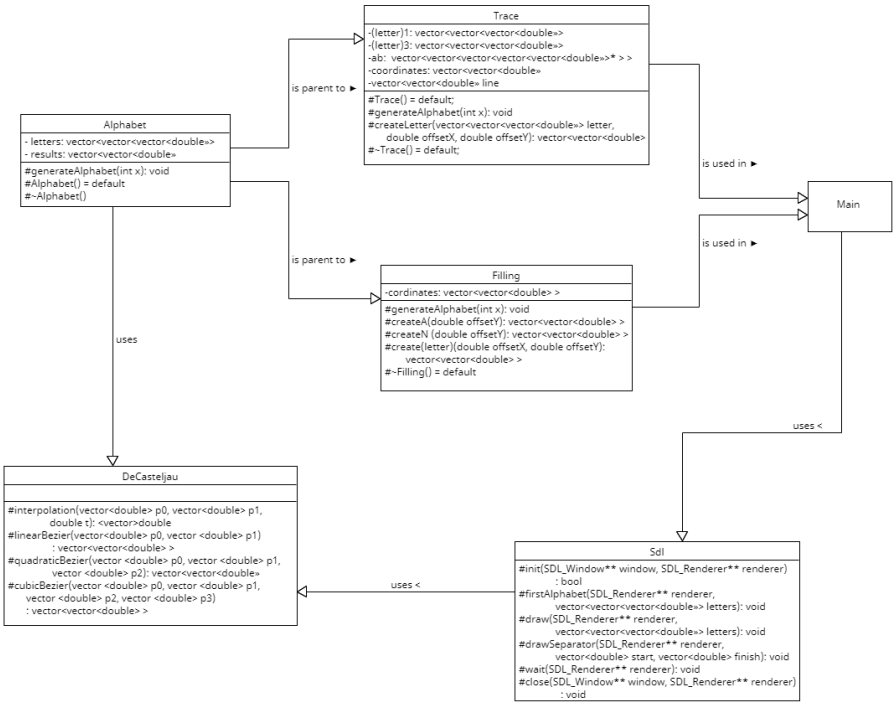
  Since for this part of the project we needed to draw the exact same alphabet twice (as opposed to tracing the first and third alphabet where the lines and curves had different lengths and curvatures), the logic of this class is completely different from Trace. Here, we have a method for each letter (`createA(), createB(), createC()` etc.) where we give different values to the X and Y offsets we utilize, to control where the alphabet is drawn. So the product is the same in any case, but with the offset parameters it is possible to control where it is created. The Filling class contains all the create functions for each letter, the `generateAlphabet(int x)` function and a vector of two sets of coordinates which determine where the alphabet is going to start (so which alphabet is going to be drawn, the second or third).

  In this class, the `generateAlphabet(int x)` function calls all the create functions one after the other, while incrementing the starting offsets that the parameter x of the function determined. Each create function follows the same logic, but the implementation is unique for each letter so we could not aggregate them in a single `createLetter` function like we did for the class Trace.

The logic is that since each letter has a length of 15px, we draw 15 lines or curves between the inner and outer layers of each letter incrementing or decreasing each point as we go. For example, for the letter C, we drew 15 curves, with the first being the outer curve of the traced version, and the last being the inner curve of the traced version. So we utilize a for loop of 15 iterations each, utilizing a bezier line or curve (whichever one is needed) with the points for it being the points of the outer curve incremented or decreased by the counter of the for loop, so that in the final run of the for loop, the inner curve of the traced version is drawn.

We tried to work with the connecting lines (meaning the lines that connect the inner and outer lines of a letter like for example the small horizontal line on top of the letter L) so that we could use the least amout of filling lines possible (so for example to fill the vertical line of L, we drew 15 vertical lines, instead of small horizontal lines for every point of the vertical line).

UML Diagram



A few notes on the diagram

- In the class Trace instead of putting all 26 vectors for the first alphabet and all 26 for the third thus reducing the readability of the diagram, we put (letter)1 and (letter)3 as placeholders for all those vectors
- In a similar fashion, in the class Filling, instead of putting all the function that create a letter (createA, createB createC etc.), we only put `createA` and `createN` which have a single offset as a parameter, and for the rest of the letters that have the same function signature we put `create(letter)`.

Technical Problems and Solutions

Locating the coordinates
We could not find a way to produce the coordinates of the points that we needed to use as input for the DeCasteljau algorithm, so we manually designed each letter and adjusted the coordinates as needed. For the third alphabet we thought we would just need to add two pixels of border to the already existing coordinates of the first alphabet, but that was not always the case, so that alphabet needed to be manually adjusted as well.

Quadratic Bezier Curve insufficiency
After tracing the first few letters, we noticed that the quadratic version of the algorithm was not performing as well as we wanted, (ex. for the outside curve of the B we could not make the quadratic version work, so we figured that for smaller curves, like the inside curves of B, P, R it would be even worse) so a cubic version with four points per curve instead of three was implemented. For us, that was sufficient so we did not proceed with a curve created by more points.

Tracing Q
We broke down Q into three main components: two half-circles (curves) and a diagonal line cutting through the right half-circle. Then, to trace it, we figured that we would need to cut the right half-circle into two smaller curves, one on each side of the diagonal line. This presented some problems in the second policy, since that way the curves were not continuous and so when filled, there were obvious gaps that were hard to fill with our method. So, in order to have a better results both aesthetically and functionally for the filling part, we traced a full circle like O, then we traced the diagonal lines to complete the Q, and then we drew over the parts that overlapped with a thick (so it would not need to be completely precise) line matching the color of the background, essentially erasing the parts we didn't want shown.

This was achieved by adding an empty line (a vector representing a line with no elements) in the vector for Q before adding the eraser line. Then, in the main function, we used a boolean variable and an if statement. When the iterator reached the empty vector, it changed the boolean variable, changing also the

color of the renderer. Then, at the next iteration, after drawing the white line (so when the last element of the vector of point to draw was reached), the color is changed back to black to continue with drawing the rest of the letters, and the boolean variable is changed back to its original value to avoid redoing the same process.

That is why in the class Sdl the first alphabet has its own function, to accommodate these changes.

Tracing and then filling curves

We decided early on in the project that each letter would have a thickness of 15px. So, when designing the letters we thought it would be enough to leave a 15px difference between the inner and outer layers of each letter. But in curves, the result wasn't always aesthetically pleasing, so we had to change some coordinates manually which resulted in coordinates with more than 15px of distance.

This created a problem for us in the filling process of the letters, since our whole method was based on that 15px distance, and by using it on every letter it created some significant gaps in between the border and the filing. In those cases, we created helper lines, and instead of creating 15 lines like the ones in the border, we created small straight lines between the inner and outer lines.

For example, the first time we faced this problem when tracing B, and when creating curves that started for the outmost top curve and went towards the innermost top curve, there were glaring gaps in the filling. So, we created again the outmost and innermost top curves and for each point of the outer curve, we created a straight line using linear Bezier, between that point and the corresponding point of the inner helper curve.

Compilation on different softwares

We worked on the project on our personal computers that work with different softwares (one with Windows11 and one with MacOS) and we found that no matter what terminal we used to run the project, we could not make it work on both. After a bit of research we found that the entrypoint for GUI applications for windows is `WinMain` instead of the traditional `main` in the main.cpp file, so we had both lines in the file (because we created a gitlab project so we both had access and worked on the same code) and we each commented and un-commented according to what worked for each of us.