

Rapport du Projet PAP 2023 - 2024

Courbes de Bézier et Police de Caractères

CHEAM Richard - CHRIAA Younès

Table des matières

	Page
1 Introduction	2
2 Courbes de Bézier : Linéaire et Quadratique	2
3 Algorithm de Casteljau	2
4 Mise en œuvre en C++	3
4.1 Conception structurelle du programme	3
4.2 Diagramme UML	3
4.3 Classes et méthodes	4
4.3.1 Point2D	4
4.3.2 Bezier	4
4.3.3 Glyph	4
4.3.4 Font	4
4.3.5 Window	5
4.3.6 SDL	5
4.4 Police de caractères	6
4.5 Le remplissage	6
4.6 Les contours rouges	6
4.7 Problème compiler sur macOS	6

1 Introduction

L'avènement des polices TrueType dans les années 1980 a révolutionné la typographie en éliminant les contraintes des polices bitmap. Contrairement à ces dernières, qui nécessitaient un fichier distinct pour chaque taille de caractère, les polices TrueType, basées sur des courbes de Bézier linéaires et quadratiques, ont offert une solution plus efficace et esthétique. Ce rapport va explorer en détail ces polices, mettant en lumière l'impact des courbes de Bézier dans leur conception et son implémentation en C++.

2 Courbes de Bézier : Linéaire et Quadratique

Courbe de Bézier Linéaire (de degré 1): Cette courbe est définie par deux points, P_1 et P_2 , et se compose simplement du segment $[P_1; P_2]$. La linéarité de cette approche offre une simplicité élégante dans la représentation des caractères. C'est simplement une ligne droite formée par deux points P_1 et P_2 .

$$B(t) = P_1 + t(P_2 - P_1), \quad t \in [0, 1]$$

Courbe de Bézier Quadratique (de degré 2): Ici, la courbe est définie par trois points : P_1 , P_2 , et C . Les points P_1 et P_2 agissent comme les extrémités de la courbe, tandis que le point C , appelé point de contrôle, influence la forme de la courbe. Cette structure permet une flexibilité accrue dans la création de formes complexes tout en conservant une certaine simplicité.

$$B(t) = P_1(1 - t)^2 + 2P_2t(1 - t) + Ct^2, \quad t \in [0, 1]$$

3 Algorithm de Casteljau

L'algorithme de Casteljau, du nom de son inventeur Paul de Casteljau, est une méthode clé en infographie pour la construction et la manipulation de courbes de Bézier. Développé initialement pour répondre aux besoins de la modélisation dans l'industrie automobile, cet algorithme se distingue par sa simplicité et son efficacité. Il permet de calculer avec précision n'importe quel point sur une courbe de Bézier en se basant sur un processus itératif de subdivision linéaire. Cette méthode divise récursivement les points de contrôle de la courbe, convergeant progressivement vers un point spécifique sur la courbe.

Le pseudo-algorithme pour calculer un point sur une courbe de Bézier quadratique en utilisant l'Algorithme de Casteljau est présenté ci-dessous :

1. Entrées:

- Points de contrôle P_0 , P_1 , et C_{12} .
- Paramètre t où $0 \leq t \leq 1$.

2. Sortie: Point $B(t)$ sur la courbe de Bézier.

3. Calculer les points intermédiaires:

- $Q_0 = (1 - t) \cdot P_0 + t \cdot C_{12}$
- $Q_1 = (1 - t) \cdot C_{12} + t \cdot P_1$

4. Calculer le point sur la courbe:

- $B(t) = (1 - t) \cdot Q_0 + t \cdot Q_1$

4 Mise en œuvre en C++

4.1 Conception structurelle du programme

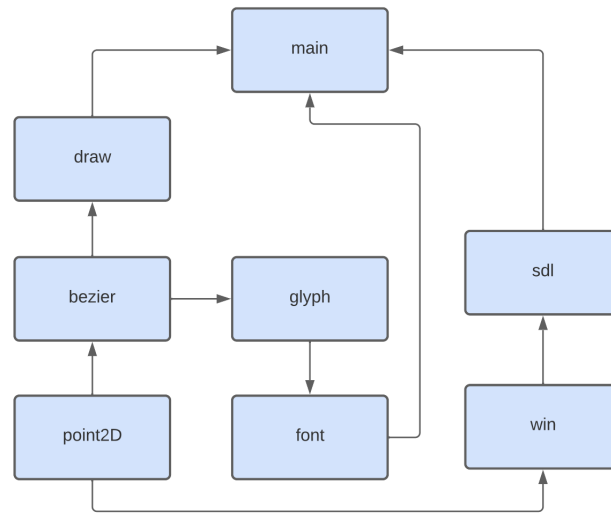


Figure 1: Dépendance des fichiers

4.2 Diagramme UML

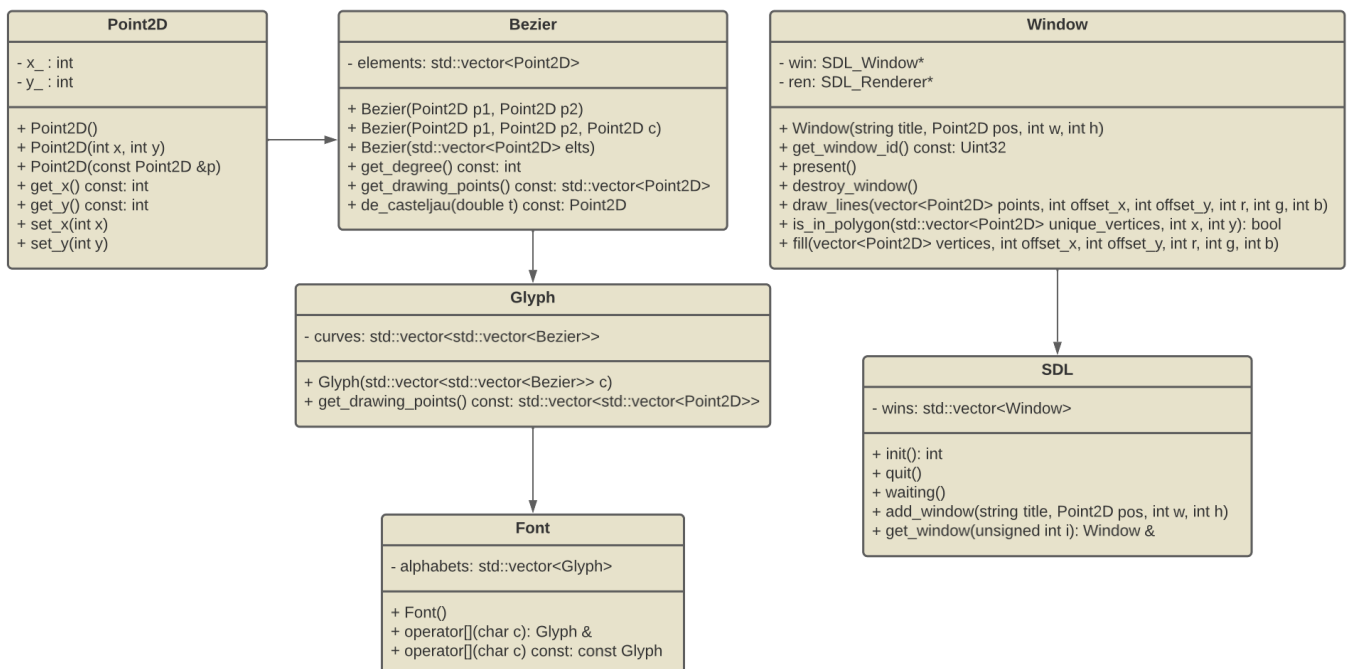


Figure 2: Diagramme UML des classes

4.3 Classes et méthodes

4.3.1 Point2D

Une instance de la classe `Point2D` représente un point avec des coordonnées dans un plan 2D. Les attributs de données de la classe sont la valeur de l'abscisse (`x`) et de l'ordonnée (`y`) du point, qui sont des nombres entiers. En ce qui concerne les méthodes, la classe dispose d'un constructeur par défaut, d'un constructeur avec des paramètres, d'un constructeur de copie, ainsi que des méthodes getter et setter pour chacune des coordonnées.

4.3.2 Bezier

Une instance de la classe `Bezier` représente une courbe de Bézier Linéaire ou bien Quadratique. Son attribut de données est un vecteur contenant 2 ou 3 objets de la classe `Point2D`, qui représentent les points de contrôle de la courbe. La classe dispose des méthodes suivantes :

- `Bezier(Point2D p1, Point2D p2)`: Constructeur pour une courbe de Bézier linéaire avec deux points de contrôle.
- `Bezier(Point2D p1, Point2D p2, Point2D c)`: Constructeur pour une courbe de Bézier quadratique avec trois points de contrôle.
- `Bezier(std::vector<Point2D> elts)`: Constructeur qui accepte un vecteur de points de contrôle.
- `int get_degree() const`: Retourne le degré de la courbe de Bézier (linéaire ou quadratique).
- `std::vector<Point2D> get_drawing_points() const`: Calcule et retourne les points qui composent la courbe de Bézier pour le dessin.
- `Point2D de_casteljau(double t) const`: Calcule un point sur la courbe de Bézier à un paramètre `t` donné en utilisant l'algorithme de De Casteljau.

4.3.3 Glyph

Une instance de la classe `Glyph` représente un ensemble de contours continus d'un caractère. Son attribut de données est un vecteur de vecteurs d'objets `Bezier`, où chaque vecteur interne d'objets `Bezier` représente un contour continu unique du caractère. Les méthodes de la classe sont les suivantes :

- `std::vector<std::vector<Bezier>> curves`: Attribut stockant les courbes de Bézier qui composent le glyph.
- `Glyph(std::vector<std::vector<Bezier>> c)`: Constructeur qui initialise un glyph avec un ensemble de courbes de Bézier.
- `std::vector<std::vector<Point2D>> get_drawing_points() const`: Calcule et retourne les points de dessin pour chaque contour du glyph.

4.3.4 Font

Une instance de la classe `Font` dans ce cas est constitué de 26 objets (alphabets) de `Glyph`, chacun représentant une lettre majuscule. Les méthodes de la classe sont les suivantes :

- `Font()`: Constructeur qui initialise la fonte avec les 26 lettres majuscules.
- `Glyph &operator[] (char c)`: Surcharge de l'opérateur `[]` pour accéder au glyph correspondant à un caractère donné.
- `const Glyph operator[] (char c) const`: Surcharge constante de l'opérateur `[]` qui retourne une référence constante au glyph correspondant à un caractère donné.

4.3.5 Window

Une instance de la classe `Window` représente une fenêtre à afficher. Il se compose d'un `SDL_Window*` et `SDL_Renderer*`. Les méthodes de la classe sont les suivantes :

- `Window(std::string title, Point2D pos, int w, int h)`: constructeur avec des paramètres.
- `get_window_id() const`: récupérer ID de la fenêtre.
- `present()`: mettre à jour la fenêtre avec tout rendu effectué lors de l'appel précédent.
- `destroy_window()`: détruire les membres de la classe `Window`.
- `draw_lines(std::vector<Point2D> points, int offset_x, int offset_y, int r, int g, int b)`: dessiner des lignes de points avec la couleur donnée (RGB). La valeur du déplacement est donnée par le décalage `offset_x` et le décalage `offset_y`.
- `is_in_polygon(std::vector<Point2D> unique_vertices, int x, int y)`: tester si un point donné se trouve ou non dans un polygone (fonction auxiliaire pour le remplissage ci-dessous).
- `fill(std::vector<Point2D> vertices, int offset_x, int offset_y, int r, int g, int b)`: remplir un polygone d'une couleur donnée sur la fenêtre.

4.3.6 SDL

L'instance de la classe `SDL` est responsable de l'amorçage de la bibliothèque `SDL` et de la libération de la mémoire attribuée par les composants de `SDL` avant la clôture de l'application. Son attribut de données étant un vecteur d'objets `Window`. Les méthodes de la classe sont les suivantes :

- `init()`: pour initialiser la bibliothèque `SDL`.
- `quit()`: pour nettoyer les éléments `SDL` initialisés.
- `waiting()`: pour attendre que l'événement appelle à la fermeture.
- `add_window(std::string title, Point2D pos, int w, int h)`: pour créer une nouvelle fenêtre et ajoutez-la au vecteur des fenêtres.
- `get_window(unsigned int i)`: pour retourner une fenêtre à partir du vecteur.

4.4 Police de caractères

Pour concevoir une police, nous commençons par son initialisation via le constructeur `Font`. Ensuite, nous accédons à chaque caractère de la police en utilisant l'opérateur `[]` pour intégrer des courbes qui définissent la forme du caractère (le contour). Chaque caractère est défini avec une dimension de 100 par 100 pixels. Par exemple, pour avoir une ligne droite, on appelle constructeur de Bézier avec deux points, start (P_1) et end (P_2), la connexion entre ces deux points forme une ligne droite. Plus précisément pour le caractère `A`, on commence par `(Point2D(10, 100), Point2D(40, 0))`, cela formera une ligne droite courbée vers la droite sur une distance de 30 (x). Par ailleurs, on appelle constructeur de Bézier avec trois points avec le point contrôle `C` lorsqu'il s'agit de courber la courbe, par exemple en forme du caractère `O`. De plus, à l'aide de la méthode `glyph` pour récupérer les points de dessin de chaque caractère, nous utilisons ensuite la méthode `draw_lines()` de la classe `Window` pour les afficher sur la fenêtre contenant 7 alphabets par ligne, au total 4 lignes dont la dernière ligne ayant 5 caractères.

4.5 Le remplissage

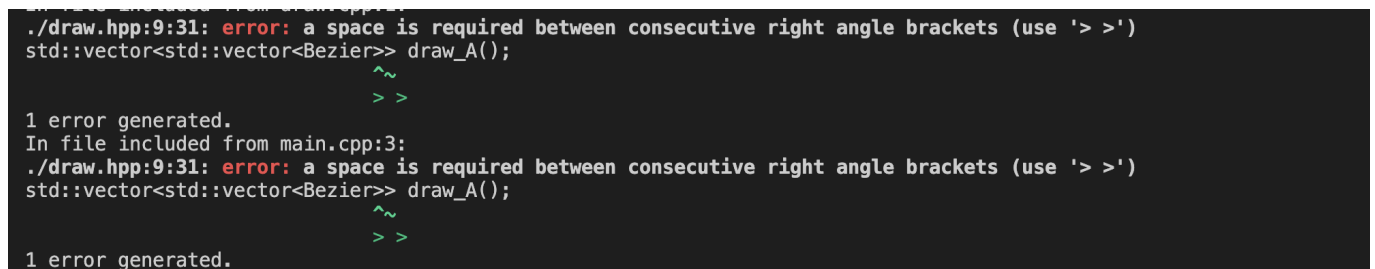
De plus, pour remplir l'intérieur du caractère en noir, d'abord vérifier les pixels à remplir (pas les pixels dehors d'un caractère). La fonction s'assure aussi de la validité du polygone en vérifiant s'il a au moins quatre sommets ; sinon, elle lève une exception parce qu'un polygone de moins de trois sommets ne peut pas former une forme fermée, et un polygone de moins de quatre sommets ne suffit pas à définir une forme convexe dans un espace à deux dimensions. Le remplissage compose de deux parties, il faut d'abord remplir tout l'alphabet avec de la couleur noire et ensuite remplir les trous des lettres comme `A`, `B`, `D`, `O`, `P`, `Q`, `R` avec de la couleur blanche.

4.6 Les contours rouges

Pour créer une bordure rouge d'un pixel autour d'un caractère, nous reproduisons le contour du caractère en rouge huit fois, puis chaque copie est déplacée d'un pixel dans l'une des huit directions. Afin d'obtenir une bordure rouge de 2 pixels, nous répétons le même processus une deuxième fois.

4.7 Problème compiler sur macOS

Le compilateur de macOS n'est pas satisfait de l'utilisation de `'>>'`, donc pour répondre à ce problème, `'>>'` a été remplacé par `'> >'` avec un espace entre les crochets, ce qui est un peu bizarre mais compilable.



```
./draw.hpp:9:31: error: a space is required between consecutive right angle brackets (use '> >')
std::vector<std::vector<Bezier>> draw_A();
                        ^~
                        > >
1 error generated.
In file included from main.cpp:3:
./draw.hpp:9:31: error: a space is required between consecutive right angle brackets (use '> >')
std::vector<std::vector<Bezier>> draw_A();
                        ^~
                        > >
1 error generated.
```

Figure 3: An error of right angle double brackets

De plus, lors de l'initialisation d'un glyph pour l'alphabet, le programme n'est pas satisfait de l'accolade après `'>>>'`. Donc, il n'est pas possible de dessiner ensemble des points (courbes) dans la parenthèse. Par exemple, `Glyph(std::vector<std::vector<Bezier>>{{ Bezier(Point2D(0,100), Point2D(38,0)) }})`

```
main.cpp:11:57: error: a space is required between consecutive right angle brackets (use '> >')
    font['A'] = Glyph(std::vector<std::vector<Bezier>>>{
                                     ^~
                                     > >
main.cpp:11:59: error: expected '(' for function-style cast or type construction
    font['A'] = Glyph(std::vector<std::vector<Bezier>>>{
                                     ~~~~~~^
2 errors generated.
```

Figure 4: An error of curly brackets in the initialization of Glyph

Par conséquent, dans le fichier `draw.cpp`, une fonction a été créée à la place en initialisant d'abord un vecteur de vecteurs contenant des courbes de Bézier définissant le glyphe lui-même et en utilisant `emplace.back()` pour établir la connexion entre deux points un par un. Cette approche est un peu naïve, mais elle fonctionne. Par exemple, `A[0].emplace_back(Point2D(10, 100), Point2D(40, 0));`.