Courbes de Bézier et police de caractères

Mohamed-Reda Bouamoud / Mehdi Sahnoun

Table des matières

1 Introduction

Ce projet consiste à développer un programme en C++ permettant de créer trois polices de caractères basées sur des courbes de Bézier. Les polices TrueType, utilisées pour représenter les caractères de manière flexible et précise, seront ici générées à l'aide de courbes linéaires et quadratiques.

Trois variantes de police seront créées :
- La première avec uniquement les contours des caractères.
- La deuxième avec un remplissage intérieur des caractères.
- La troisième avec un contour rouge autour des caractères.

L'algorithme de de Casteljau, une méthode récursive pour générer des courbes de Bézier, sera utilisé pour dessiner ces glyphes, et la bibliothèque SDL servira à afficher les résultats graphiquement. Ce projet permettra d'explorer l'application des courbes de Bézier dans la conception de polices de caractères et leur affichage sur un écran.

2 Conception

L'architecture du projet est représentée dans le diagramme UML ci-dessous :

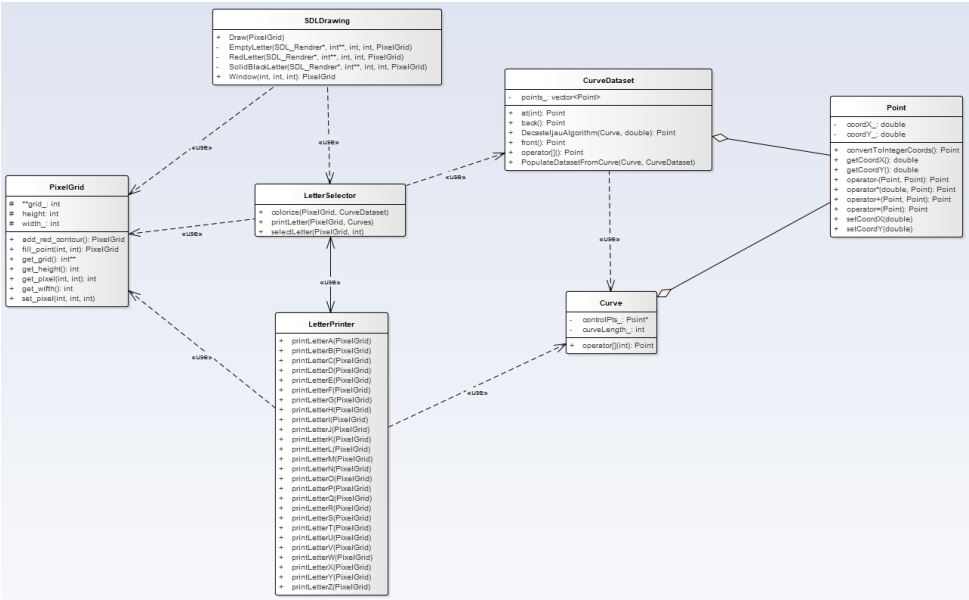

FIGURE 1 – Diagramme UML de l'architecture du projet

Le diagramme UML montre les différentes classes et leurs relations dans le projet. Voici une description détaillée de chaque composant :

- **PixelGrid** : Cette classe représente une grille de pixels où chaque caractère sera dessiné. Elle contient des méthodes pour ajouter des contours rouges, remplir des points, et obtenir ou définir des pixels spécifiques.

— **SDLDrawing** : Cette classe utilise la bibliothèque SDL pour dessiner les caractères sur l'écran. Elle contient des méthodes pour dessiner des lettres vides, des lettres rouges, des lettres noires solides, et pour créer une fenêtre SDL.
— **LetterPrinter** : Cette classe contient des méthodes pour imprimer chaque lettre de l'alphabet sur la grille de pixels.
— **LetterSelector** : Cette classe permet de sélectionner et de colorier des lettres spécifiques sur la grille de pixels.
— **CurveDataset** : Cette classe gère un ensemble de points qui définissent les courbes de Bézier. Elle contient des méthodes pour accéder aux points, appliquer l'algorithme de de Casteljau, et peupler le dataset à partir d'une courbe.
— **Curve** : Cette classe représente une courbe de Bézier avec ses points de contrôle et sa longueur. Elle permet d'accéder aux points de contrôle.
— **Point** : Cette classe représente un point dans l'espace avec des coordonnées X et Y. Elle contient des méthodes pour obtenir et définir les coordonnées, ainsi que pour convertir les coordonnées en entiers.

Les relations entre les classes sont illustrées dans le diagramme par des flèches et des symboles spécifiques. Par exemple, la classe *LetterSelector* interagit avec les classes *PixelGrid* et *CurveDataset*, tandis que la classe *SDLDrawing* exploite la classe *PixelGrid* pour représenter les caractères graphiquement. En outre, des relations d'agrégation sont présentes dans l'architecture. Par exemple, la classe *CurveDataset* est composée de plusieurs objets de la classe *Point*, indiquant que *CurveDataset* agrège des points sans posséder leur cycle de vie, c'est-à-dire que les points peuvent exister indépendamment. Cette relation est représentée dans le diagramme UML par une ligne terminée par un losange vide.

Cette architecture modulaire permet une gestion claire et efficace des différentes fonctionnalités nécessaires à la création et à l'affichage des polices de caractères basées sur des courbes de Bézier.

3 Algorithme de de Casteljau

Pour tracer une courbe de Bézier, l'algorithme récursif de de Casteljau a été implémenté. Cet algorithme permet de calculer les barycentres successifs des segments définis par les points de contrôle, jusqu'à obtenir un point sur la courbe correspondant à une valeur donnée de t.

L'implémentation en C++ est présentée ci-dessous :

```
Point DeCasteljauAlgorithm(const Curve& controlPoints, const double t) {
    if(t >= 0 && t <= 1){
        int length = controlPoints.getCurveLength();
        Curve newCurve(controlPoints);  // Create a copy of the control points

        Point point;
        for (int j = 1; j < length; ++j) {
```

```
            for (int i = 0; i < length - j; ++i) {
                // Perform linear interpolation between control points
                point = (1 - t) * newCurve[i] + t * newCurve[i + 1];
                newCurve[i] = point;  // Update the control point after interpolation
            }
        }

        // Return the computed point after all iterations
        return newCurve[0];
    }
    else throw std::out_of_range("Invalid value of t!");
}
```

Cette fonction calcule un point spécifique sur la courbe de Bézier en fonction de la valeur de $t \in [0, 1]$.

En outre, une autre méthode a été développée pour générer un ensemble de points à partir d'une courbe et peupler un *dataset* de points :

```
void PopulateDatasetFromCurve(Curve curve, CurveDataset& dataset) {
    int size = dataset.size();

    // Set the first and last points of the dataset directly
    dataset.front() = curve[0];
    dataset.back() = curve[1];

    // Compute and populate intermediate points
    for (int i = 1; i < size ; ++i) {
        double t = static_cast<double>(i) / (size - 1);  // Calculate t for the curre
        Point computedPoint = DeCasteljauAlgorithm(curve, t);
        dataset.at(i) = computedPoint;  // Use at() for safety in accessing elements
    }
}
```

La fonction `PopulateDatasetFromCurve` utilise l'algorithme de de Casteljau pour calculer et insérer les points intermédiaires dans un dataset. Ces points sont ensuite utilisés pour afficher la courbe de Bézier sur une fenêtre SDL.

Cet algorithme est fondamental dans la construction de courbes de Bézier et garantit leur précision, tout en étant adaptable à différentes résolutions.

4 Choix des Points pour les Courbes de Bézier

Pour définir les lettres avec des courbes de Bézier, nous avons choisi les points de contrôle manuellement en utilisant un outil interactif : GeoGebra. Cet outil nous a permis de tracer

les courbes des lettres et de déterminer les coordonnées des points de contrôle. Ces points ont ensuite été traduits en code pour dessiner les lettres dans le programme.

Voici un exemple du processus pour la lettre "A" :

```
void LetterPrinter::printLetterA(PixelGrid &table)
{
    // Define control points for Bezier curves
    Curve A1(3);
    Curve A2(3);
    Curve A3(2);

    // Left diagonal
    A1[0] = Point(0.2, 0.8);
    A1[1] = Point(0.35, 0.5);
    A1[2] = Point(0.5, 0.2);

    // Right diagonal
    A2[0] = Point(0.8, 0.8);
    A2[1] = Point(0.65, 0.5);
    A2[2] = Point(0.5, 0.2);

    // Crossbar
    A3[0] = Point(0.35, 0.5);
    A3[1] = Point(0.65, 0.5);

    // Print the letter A
    LetterSelector::printLetter(table, A1, A2, A3);
}
```

4.1 Processus de Création

1. **Visualisation des lettres** : Nous avons utilisé GeoGebra pour dessiner chaque lettre de l'alphabet en utilisant des courbes de Bézier. Cet outil nous a permis de visualiser les diagonales et les segments horizontaux, comme ceux nécessaires pour la lettre "A".
2. **Détermination des points de contrôle** : En traçant les courbes, nous avons sélectionné des points stratégiques pour définir les diagonales et les segments horizontaux. Par exemple :
   — Les points `(0.2, 0.8)`, `(0.35, 0.5)`, et `(0.5, 0.2)` définissent la diagonale gauche.
   — Les points `(0.35, 0.5)` et `(0.65, 0.5)` définissent la barre horizontale centrale.
3. **Traduction en code** : Les points déterminés ont été insérés dans des objets `Curve` dans le code. Chaque segment de la lettre est défini par une courbe distincte, qui est ensuite dessinée sur la grille de pixels via la méthode `LetterSelector::printLetter`.

4.2 Illustration

Une capture d'écran du processus dans GeoGebra est présentée ci-dessous, montrant la création de la lettre "A" et les points de contrôle choisis :

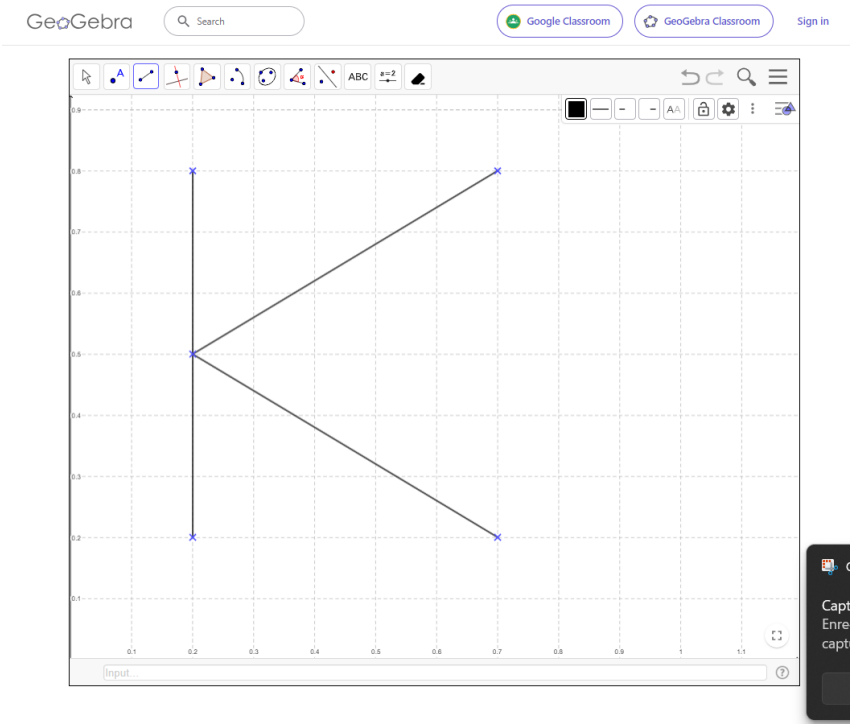

FIGURE 2 – Création des points de contrôle pour la lettre "K" dans GeoGebra

Cette méthode garantit que les courbes reflètent précisément les formes des lettres tout en utilisant des approximations efficaces.

5 Gestion des Différences de Complexité entre les Lettres

L'un des principaux défis de ce projet réside dans la variabilité du nombre de courbes nécessaires pour dessiner chaque lettre. Par exemple :

- La lettre **T** nécessite seulement deux courbes.
- La lettre **Q**, quant à elle, exige cinq courbes différentes.

Cette disparité rendait le code initialement complexe et difficile à maintenir. Pour réduire cette complexité, nous avons décidé de déléguer la création des courbes à des fonctions spécifiques de la classe `LetterPrinter`.

5.1 Solution : Délégation aux Fonctions de `LetterPrinter`

La classe `LetterPrinter` contient une fonction dédiée pour chaque lettre, par exemple `printLetterA`, `printLetterB`, etc. Ces fonctions créent les courbes nécessaires pour leur lettre respective et délèguent l'impression à la méthode générique `LetterSelector::printLetter`. Voici une vue d'ensemble de cette organisation :

```
class LetterPrinter {
public:
    // Functions to print individual letters
    static void printLetterA(PixelGrid &grid);
    static void printLetterB(PixelGrid &grid);
    static void printLetterC(PixelGrid &grid);
    // ...
};
```

Chaque fonction de `LetterPrinter` utilise une approche adaptée pour sa lettre spécifique. Par exemple, `printLetterA` configure trois courbes pour la lettre "A", tandis que `printLetterQ` configure cinq courbes pour la lettre "Q".

5.2 Impression Générique avec `LetterSelector`

Pour éviter la répétition de code, nous avons conçu une méthode générique dans la classe `LetterSelector` qui gère les courbes indépendamment de leur nombre. La méthode `LetterSelector::printLetter` utilise des *fold expressions* pour traiter un nombre variable de courbes de manière concise et efficace. Voici son implémentation :

```
template <typename... Curves>
void LetterSelector::printLetter(PixelGrid &grid, Curves&... curves) {
    int size = std::max(grid.get_width(), grid.get_height());
    CurveDataset points(size);

    // Fold expression to process each curve
    (void)std::initializer_list<int>{(
        (curves.getCurveLength() > 0 ?
         (PopulateDatasetFromCurve(curves, points), colorize(grid, points)) : void())
        0)...};

    grid.add_red_contour();
}
```

Cette méthode réalise les étapes suivantes :

1. **Initialisation** : Un `CurveDataset` est créé pour stocker les points calculés à partir des courbes.
2. **Traitement des courbes** : Chaque courbe est transformée en un ensemble de points via la fonction `PopulateDatasetFromCurve`, puis colorisée sur la grille avec `colorize`.
3. **Ajout du contour rouge** : Enfin, un contour rouge est ajouté autour de la lettre.

5.3 Avantages de cette Approche

— **Modularité** : Chaque lettre est gérée par une fonction indépendante dans `LetterPrinter`, rendant le code facile à lire et à modifier.
— **Réutilisation** : La méthode générique `LetterSelector::printLetter` centralise le traitement des courbes, réduisant les répétitions dans le code.
— **Scalabilité** : Cette structure facilite l'ajout de nouvelles lettres ou de styles de polices à l'avenir.

6 Résultats du Programme

Le programme offre deux modes d'affichage distincts, chacun mettant en avant les polices de caractères générées à partir des courbes de Bézier. Ces modes sont configurables selon les besoins de l'utilisateur en modifiant les appels de fonctions dans le code.

6.1 Mode 1 : Affichage Parallèle

Dans ce mode, la lettre sélectionnée est affichée simultanément dans trois fenêtres SDL distinctes, chacune utilisant un style de police différent :
— **Contour Noir** : Affiche uniquement les contours de la lettre.
— **Lettre Noire Solide** : Remplit l'intérieur de la lettre en noir.
— **Contour Rouge** : Ajoute un contour rouge de deux pixels autour de la lettre pour une meilleure lisibilité.
Voici un exemple d'appel correspondant dans le code :

```
// Draw the letter in three separate windows using threads
lettreSDL.Draw('Q', 40, 50);
```

Ce mode permet de comparer les différents styles visuellement et de les évaluer dans des fenêtres séparées.

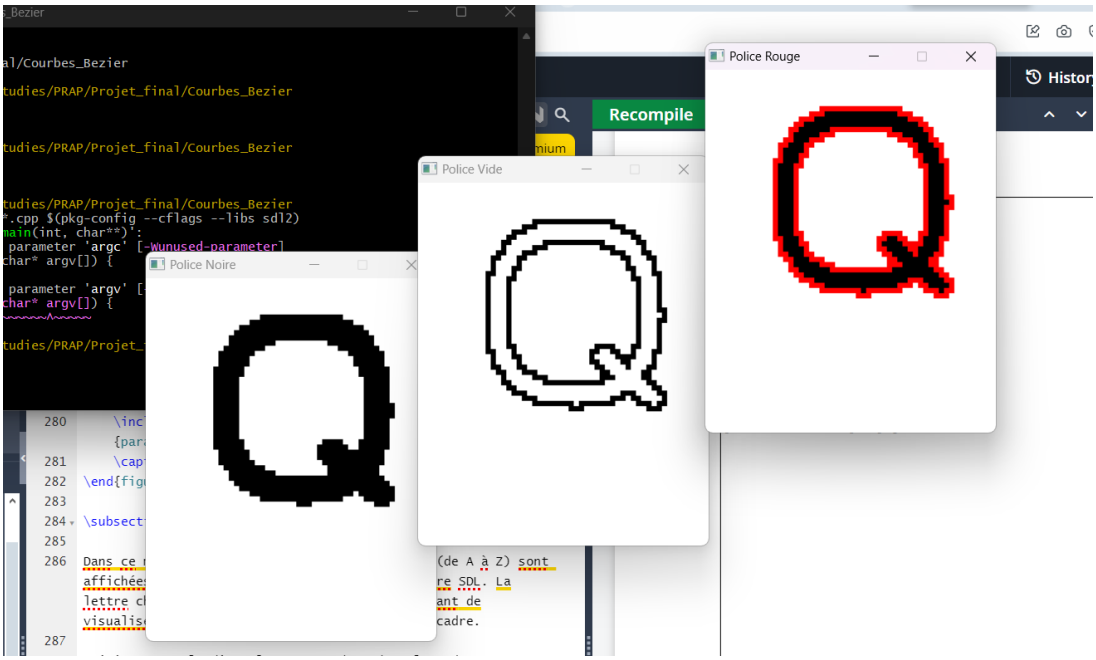

FIGURE 3 – Exemple du mode d'affichage parallèle

6.2 Mode 2 : Affichage Séquentiel

Dans ce mode, toutes les lettres de l'alphabet (de A à Z) sont affichées séquentiellement dans une seule fenêtre SDL. La lettre change toutes les deux secondes, permettant de visualiser l'ensemble des lettres dans un seul cadre.

Voici un exemple d'appel correspondant dans le code :

```
// Draw all the letters one after another in one window
lettreSDL.Draw2(40, 50);
```

Ce mode est particulièrement utile pour examiner l'ensemble des polices dans un contexte linéaire et pour vérifier leur cohérence.

Pour visionner la vidéo illustrant le **Mode Séquentiel**, veuillez ouvrir ce fichier PDF avec **Adobe Reader** (et non un lecteur PDF quelconque), car ce dernier prend en charge les vidéos intégrées.

Cliquez ici pour lire la vidéo.

6.3 Analyse des Modes

Les deux modes offrent des perspectives différentes pour l'évaluation des polices :

— **Mode Parallèle** : Idéal pour comparer directement les styles et explorer leurs utilisations possibles.
— **Mode Séquentiel** : Plus adapté pour examiner chaque lettre dans son contexte alphabétique complet.