Jonathan Bailleul

11 juin 2001

Rendu rapide de splines rationnelles (NURBS) dans une bibliothèque graphique (CLIM) par Common Lisp

DEA Informatique
LaBRI, Université Bordeaux I

Sujet proposé par Robert Strandh

Résumé

CLIM (*Common Lisp Interface Manager*) est une bibliothèque ANSI Common Lisp permettant le développement d'interfaces graphiques dans le cadre du système X. Pour délimiter l'affichage de fenêtres X, CLIM utilise des objets appelés *régions*, définies par des formes géométriques construites par composition d'ellipses et de rectangles.

En premier lieu, une étude du modèle des régions nous permet de mettre en évidence certaines de ses lacunes, et nous incite à rechercher un modèle géométrique éventuellement plus expressif et facile à manipuler, mais qui permette surtout d'éviter les incohérences constatées.

Le modèle le plus simple et le plus fréquemment cité qui corresponde à ces premiers critères est celui des courbes B-Splines . Son étude détaillée vérifie que ses propriétés géométriques répondent à la plupart de nos attentes, exceptée la capacité à représenter exactement cercles et ellipses. Nous optons alors pour une version rationnelle de ce modèle : celui des courbes NURBS, véritable standard en modélisation géométrique.

Le contexte d'affichage nous imposant de tracer les NURBS sous forme d'approximations polygonales, nous survolons alors les principales familles de méthodes permettant de les déterminer. Puis, afin de sélectionner certaines méthodes, nous procédons à une étude comparative de versions optimisées de leurs algorithmes qui privilégie la vitesse d'exécution. Cela implique l'établissement préalable d'un protocole d'optimisation basé sur la *spécialisation de programmes.*

Cette technique, qui consiste à adapter le code des programmes à leur contexte d'exécution, n'est apparemment pas évoquée dans la littérature relative à la modélisation géométrique : elle nous permet pourtant d'économiser de 10 à 90% du temps de calcul selon les cas étudiés.

Table des matières

Chapitre 1

Objectif : régions CLIM

Ce chapitre résume [MY92] Chapitre 3, et [Har98] Chapitre 2.5.1.

1.1 Introduction au système FreeCLIM

Common Lisp Interface Manager (CLIM) est une bibliothèque ANSI Common Lisp proposant des fonctionnalités pour construire des interfaces utilisateur. CLIM permet au développeur de concevoir son interface en faisant abstraction du système d'exploitation, du système de fenêtrage et même des toolkits disponibles : seuls des objets CLIM (fenêtres, gadgets) sont manipulés par le biais d'un système ANSI Common Lisp. Le développeur peut alors concevoir son application en employant des fonctionnalités et des spécifications de haut niveau, qui seront ensuite automatiquement traduites pour les systèmes de fenêtrage et les toolkits disponibles, auxquels CLIM délègue finalement les opérations de bas niveau. De ce fait, la portabilité des applications construites avec CLIM est automatiquement assurée.

La norme de CLIM ([MY92]) étant définie de manière très précise et explicite, de nombreuses implémentations de cette interface de programmation ont vu le jour. Cependant, toutes sont exclusivement livrées avec des systèmes Lisp commerciaux, et le plus souvent contribuent à départager ces systèmes de leurs homologues gratuits.

C'est ce constat qui a motivé le projet FreeCLIM, dont l'objectif est de fournir une implémentation gratuite de CLIM utilisable sur tous les systèmes ANSI Common Lisp, commerciaux ou non. Notre projet vise à enrichir FreeCLIM de nouvelles fonctionnalités concernant les régions.

1.2 Définition des régions CLIM

Une *région* CLIM est une partition (au sens mathématique du terme) du plan cartésien 2D. De ce fait, la résolution d'une région est infinie. En outre, elles sont définies dans un repère abstrait.

- Par convention, les régions sont *fermées*, c'est-à-dire qu'elles comprennent leur *frontière*.
- Tout d'abord, on distingue les *régions bornées* (*bounded region*), dans lesquelles deux points quelconques sont forcément distants d'une longueur finie, des *régions non bornées* (*unbounded region*) qui possèdent au moins un couple de points enfreignant cette règle.

Parmi les régions bornées, on distingue les *chemins* (*path*) des *zones* (*area*) :

– Les chemins sont définis à partir d'une suite de segments de droite ou d'arcs elliptiques. Les points de ces régions se situent exclusivement sur le chemin. Ils peuvent indifféremment être ouverts ou fermés.

– Les zones délimitent une partition du plan cartésien 2D. Elles comprennent non seulement les points de leurs frontières mais aussi l'ensemble des points situés à l'intérieur de ces dernières. Les zones peuvent être définies à partir de primitives comme `rectangle` et `ellipse`.

Une région sous forme de chemin fermé ne représente donc pas forcément une zone : c'est pourquoi il existe des primitives pour créer une zone à partir d'un chemin fermé.

- L'ensemble des régions CLIM est *clos par transformation affine* :

  – Une *transformation affine* est une transformation qui s'applique à un polygone et préserve ses lignes droites. En d'autres termes, tout triplet de points alignés dans le polygone de départ sera transformé en un autre triplet de points alignés dans le polygone d'arrivée.

  – L'ensemble des primitives de régions CLIM est *clos* par transformation affine : toute transformée affine d'une région définie par composition de primitives CLIM peut être exprimée par une composition de primitives CLIM (pas nécessairement les mêmes).

Remarque : CLIM est capable de gérer des ellipses quelconques, i.e. il ne se limite pas aux *ellipses orthogonales* (ellipses dont les axes sont parallèles à ceux du repère cartésien). Les cercles sont un cas particulier des ellipses.

1.3 Rôle des régions CLIM

Les régions CLIM, représentées par des instances de la classe `region`, servent à définir des zones de *clipping*, c'est-à-dire des zones délimitant l'affichage d'objets graphiques.

Par exemple, si l'on veut dessiner dans une fenêtre une forme géométrique pleine, il faut d'abord modéliser une région de type zone qui reproduise cette forme. Ensuite, il faudra afficher en fond d'écran un rectangle suffisamment grand et définir la région comme zone de clipping de ce rectangle. Ainsi, seuls les points du rectangle qui appartiennent à la région sont effectivement affichés à l'écran (zone grise de la figure 1.1).

Nous pouvons remarquer que le rectangle de taille minimale à afficher pour représenter la forme est le rectangle englobant de notre région, que l'on peut obtenir automatiquement par application d'une fonction primitive sur la région.

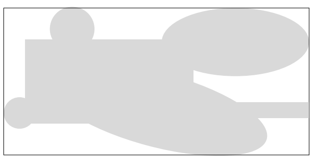

FIG. 1.1 – Clipping d'un rectangle englobant par une région.

Toute application X possède une fenêtre d'affichage et une série de fenêtres filles, qui recouvrent par exemple ses boutons ou ses menus. De même, une entrée de menu est une fenêtre fille de son menu. Chaque fenêtre X possède donc sa propre zone de clipping délimitant son affichage, zone qui sera incluse dans celle de sa fenêtre mère.

Par défaut, la zone de clipping d'une fenêtre X est réglée sur les dimensions rectangulaires de la fenêtre. Dans le cadre de CLIM, les régions servent à représenter ces zones de clipping, et offrent la possibilité d'en définir des versions plus élaborées.

1.4 Opérations sur les régions

Pour opérer sur les régions, la norme CLIM propose les primitives suivantes :

- Appartenance d'un point ou d'une région, égalité ou incidence entre régions.
- Détermination du rectangle englobant.
- Les transformations affines suivantes :
  - translation
  - rotation
  - zoom (*scaling*)
  - réflexion (image miroir)
- Conversion des coordonnées abstraites des régions dans le repère discret d'affichage.
- Composition : union, intersection, différence.

Les opérations de composition opèrent sur une collection de régions et essaient d'exprimer la région issue de la composition de manière aussi concise que possible. Par exemple, l'intersection de deux régions dont l'une englobe l'autre retournera la région incluse. Aussi, l'union de deux rectangles qui s'intersectent sera exprimée sous forme *normalisée*, c'est-à-dire sous une forme minimale d'unions de rectangles disjoints (cf figure 1.2).

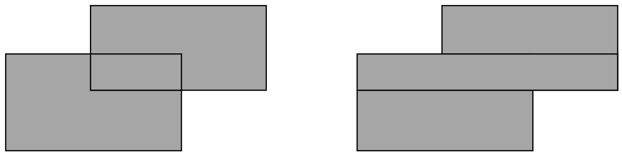

Fig. 1.2 – Normalisation en x de l'union de rectangles intersectants

La différence entre deux régions $reg1, reg2$ en CLIM est définie sous la forme d'une nouvelle région comprenant tous les points de $reg1$ qui n'appartiennent pas à $reg2$. La frontière de la région résultante ne sera pas confondue avec la frontière de $reg2$, et sera éventuellement recomposée pour délimiter une région fermée.

En outre, la composition de régions définies à partir de primitives CLIM n'est pas forcément directement exprimable sous forme d'une autre primitive CLIM. Par exemple, l'union des régions de la figure 1.3 forme une nouvelle région qui ne peut être décrite à partir d'une seule des primitives disponibles, mais seulement par la collection de certaines d'entre elles.

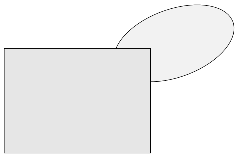

FIG. 1.3 – Union d'un rectangle et d'une ellipse

Dans ce genre de cas, nous sommes forcés de produire un objet regroupant la collection des régions composées qui définissent la composition, et précisant la nature de cette composition . D'où l'existence des classes `standard-region-union`, `standard-region- intersection` et `standard-region-difference`, qui sont des classes filles de `region-set`).

1.5 Limitations du modèle et améliorations envisageables

La figure 1.4 liste toutes les classes définissant des régions CLIM[1].

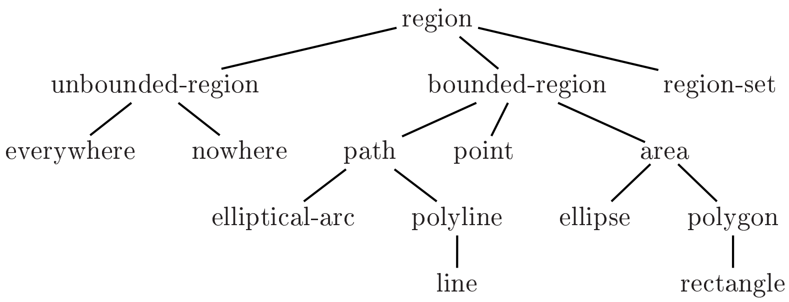

FIG. 1.4 – Hiérarchie des classes relatives aux régions

Nous pouvons remarquer qu'il n'est pas simple de modéliser des formes non-triviales par composition de primitives. Il parait en effet raisonnable de supposer qu'un utilisateur préfère plutôt définir une forme géométrique par une trajectoire de points (dessin à “main levée”) que par une séquence d'unions, intersections et différences de rectangles et ellipses.

Ensuite, la modélisation par composition peut poser des problèmes quand l'utilisateur veut modifier une partie restreinte de la forme courante. Par exemple, supposons que l'on veuille “étirer” uniquement le morceau parabolique située au dessus de la figure 1.5 : on ne peut y parvenir dans ce cas qu'en modifiant la définition de l'ellipse sous-jacente, et donc le morceau parabolique située du dessous. Par conséquent, on constate que la forme originale à été altérée au-delà de la zone désirée.

Cet exemple est sans doute trivial, mais il suffit à montrer les difficultés que peut rencontrer un utilisateur procédant à la modélisation interactive de régions de formes élaborées.

Plus grave encore : il existe des régions définies par composition pour lesquelles certaines opérations ne sont plus valides. Par exemple, soit sur la figure gauche de 1.6 les régions $R = R_1 \cup R_2$ et r. Avec le modèle actuel, il est impossible de définir si r est incluse dans R. En effet, même si l'on peut trouver des

[1]les classes `standard-region-*` sont omises

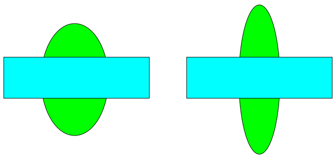

FIG. 1.5 – Difficultés de la modification locale de formes composées

points de r appartenant à R_1, d'autres à R_2, et même d'autres encore à $R_1 \cap R_2$, on ne peut prouver l'inclusion que si l'on s'est assuré qu'aucun point de r est défini en dehors de R. En effet, les régions sont de résolution infinie et ne définissent que des partitions de l'espace 2D. En conséquence, l'opération d'inclusion s'avère calculable partielle sous le jour de cet exemple.

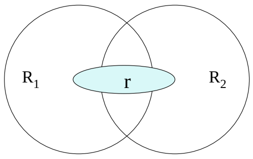

FIG. 1.6 – Exemple invalidant l'opération de test d'inclusion d'une région

Un modèle permettant de calculer la forme de la composition de R_1 et R_2 en déterminant les points d'intersections de ces régions nous tirerait d'embarras dans ce cas (cf figure droite de 1.6). En effet, il suffirait de déterminer si les contours de R et r s'intersectent, puis de vérifier si un point quelconque de r appartient bien à R.

Les limitations du modèle actuel des régions nous encouragent à rechercher un modèle géométrique unique qui devra répondre aux exigences suivantes :

- Exprimer toutes les formes géométriques déjà proposées par CLIM.
- Etendre le domaine de ces formes à des formes plus élaborées. De plus, il serait judicieux d'employer un modèle permettant de définir des formes à partir de trajectoires arbitraires, puis d'autoriser des modifications locales de ces formes.
- Aussi souvent que possible, exprimer toute composition de régions sous forme d'une seule autre région.

En effet, il y aura toujours des cas où il sera impossible d'exprimer la composition de formes sans employer la notion de collection (représentée dans CLIM par les instances de la classe `region-set`). Par exemple, l'union de deux régions disjointes ne peut être exprimée par une seule instance d'un modèle géométrique, quand bien même ce dernier s'avérerait capable de déduire le contour d'un amas de régions mutuellement intersectantes sans conserver la définition d'aucune d'entre elles.

En conséquence, nous aurons toujours besoin de la classe `standard-region-union` et donc de sa classe mère `region-set`. Cependant, nous pourrons nous affranchir définitivement des classes `standard-region-intersection` et `standard-region-difference`.

Nous allons donc parcourir la littérature relative à la modélisation géométrique afin de déterminer quel type de modèle pourrait répondre à nos exigences.

Chapitre 2

Le choix des NURBS

Ce chapitre résume le cours de Modélisation Géométrique de Carole Blanc (DEA LaBRI, 2000/2001) [Bla94] Chapitres 2 & 3, et [PT95] Chapitres 2, 3 & 4. Toutes les figures proviennent de [Bla94].

Les régions CLIM sont des objets de résolution infinie et de forme arbitrairement complexe. Nous souhaitons les représenter via un modèle unique, qui permette notamment une modélisation pratique et dont le tracé n'exigerait que peu de ressources machine.

Dans la première section, nous expliquerons pourquoi les caractéristiques attendues induisent le choix d'un modèle polynômial, paramétrique et défini par morceaux.

Dans la deuxième section, nous étudions le cas des courbes B-Splines : le plus simple et le plus répandu des modèles géométriques correspondant aux trois critères précédemment énoncés.

Bien que la plupart des propriétés des B-Splines font de ce modèle un excellent choix pour la représentation des régions, ce modèle ne permet pas d'exprimer les sections coniques (cf annexe 6.1). Comme nous devons être capables de les représenter, nous étudierons dans la troisième section le modèle des courbes NURBS, défini à partir des B-Splines , et permettant de résoudre ce problème.

Nous pouvons noter que les régions et les courbes sont des formes en 2 dimensions. Pourtant, nous prendrons le repère Euclidien $\mathbb{R}^3$ comme base pour définir nos courbes. En effet, dans la littérature relative à la modélisation géométrique, les courbes sont toujours définies en 3 dimensions et nous souhaitons respecter cette convention.

2.1 Pourquoi un tel modèle ?

Nécessité d'un modèle polynômial défini par morceaux :

Intéressons-nous au problème de l'interpolation numérique. Il s'agit d'obtenir à partir d'un échantillon de points une courbe interpolante de résolution infinie. Par exemple, à partir des points prélevés sur une maquette, on voudrait retrouver une courbe fidèle au modèle original, mais surtout définie pour toute autre valeur que celle de l'échantillonage.

Une courbe approximant un échantillon de n points peut toujours être représentée par un polynôme de degré (minimum) $n-1$ [MGLV]. Cette première approche de la modélisation géométrique par polynôme unique pose déjà certains problèmes :

- Des formes évoluées approximant un grand nombre de points vont requérir des polynômes de haut degré, par conséquent très coûteux à évaluer.

- Lorsqu'un utilisateur va souhaiter modifier localement la forme de la courbe en déplaçant un point à approximer (par exemple), la forme de la courbe s'avérera affectée en totalité, et qui plus est de manière peu prévisible. Cela rend le travail de modélisation très ardu, surtout en mode interactif.

En conséquence, nous allons nous tourner vers des *courbes par morceaux* (piecewise curves), permettant de définir des formes arbitrairement complexes à partir d'une série illimitée de *points de contrôle* que la courbe va *suivre* (interpoler, approximer, voire les deux par alternance). La courbe globale sera alors formée par une série de sections de courbes connectées, chaque section étant représentée par un polynômes de faible *degré p* fixé dès la phase de modélisation.

Nécessité d'un modèle paramétrique : [1]

Nous avons arrêté notre choix sur un modèle polynômial défini par morceaux. Il reste maintenant à fixer la définition de ces polynômes :

- **Définition fonctionnelle :** Un point $P_c : \{a, b, c\} \in \mathbb{R}^3$ de la courbe est le résultat d'une fonction explicite $F(x) = \{x, f(x), g(x)\}$. Cependant, cette définition interdit à plusieurs points de partager la même abcisse, ce qui complique notamment la représentation de sections coniques : par exemple, il faudrait définir deux fonctions différentes pour représenter un cercle.

  Le modèle que nous attendons devrait permettre de suivre les points de contrôle indépendamment de leur position, ce qui est impossible ici.

- **Définition implicite :** Soit une forme volumique V définie par un ensemble de points $P_V : \{x, y, z\} \in \mathbb{R}^3$ satisfaisant l'inégalité $F(x, y, z) \leq 0$. La fonction $F(P), P \in \mathbb{R}^3$, est strictement positive pour P à l'extérieur de V, strictement négative pour P à l'intérieur, et nulle pour P sur sa frontière. Par exemple, un cercle peut être défini par $\{P_c : \{x, y, 0\} \in \mathbb{R}^3 \setminus x^2 + y^2 = 1\}$.

  Cependant, il est a priori impossible de trouver un point sur la courbe sans tester auparavant plusieurs valeurs de points de $\mathbb{R}^3$. Cela exige un temps de calcul qui peut convenir aux applications procédant au lancer de rayons mais qui sera prohibitif dans notre cas. D'autres arguments sont développés dans [FvDF+95] et soulignent le caractère inadéquat de cette définition.

- **Définition paramétrique :** Un point $P_t : \{x, y, z\} \in \mathbb{R}^3$ de la courbe est le résultat d'une fonction paramétrique $F(t) = \{X(t), Y(t), Z(t)\}$. En faisant varier t sur son intervalle de définition, on peut calculer une série de points qui décrivent la courbe, par analogie avec la position d'un mobile en mouvement qui évolue sur sa trajectoire en fonction du temps.

  Toute valeur de t permet de calculer un point sur la courbe, et plusieurs de ces points peuvent partager la même abscisse : par exemple, une des paramétrisations du cercle sera $\{x(t) = cos(t); y(t) = sin(t); z(t) = 0\}$.

[1] d'après [FvDF+95] pp340–341

Les recherches bibliographiques entreprises montrent que parmi les modèles polynômiaux, définis par morceaux et paramétriques, celui des courbes B-Splines est le plus représenté et aussi le plus simple. Tirant parti des documents collectés, nous allons donc procéder à une étude approfondie de ce modèle pour dégager ses propriétés géométriques et déterminer dans quelle mesure il peut répondre à nos attentes.

2.2 Courbes B-Splines

2.2.1 Définition

Les courbes B-Spline forment un modèle géométrique continu, donc de résolution infinie, et défini par morceaux.

Un point sur une B-Spline de *degré* p (d'*ordre* p+1) est donc défini comme suit :

$$C(t) = \sum_{k=0}^{n} N_{k,p}(t) P_k \tag{2.1}$$

- t étant la valeur choisie du paramètre, que nous prendrons par convention dans l'intervalle $[0, 1]$
- P_k étant un des $n+1$ points de contrôle orientant la forme de la courbe. Le chemin formé par les points de contrôles est aussi appelé *réseau de contrôle*.
- $N_{k,p}(t)$ étant une *fonction d'influence* de la B-Spline , aussi nommée *fonction de base*. Remarquons que B-Spline signifie "Spline Basis".

Les fonctions de base sont connues sous le nom de *fonctions de Schoenberg*. Ce dernier proposa une définition de ces fonctions sous forme de monômes tronqués [Sch46], alors qu'il travaillait sur des techniques d'interpolation numérique.

Une forme récursive de ces fonctions a été développée simultanément par Cox [Cox72] et De Boor [Boo72]. C'est elle qui est actuellement utilisée dans la mesure où son évaluation est plus stable et performante.

$N_{i,p}(t)$ est alors définie comme suit :

- Soit $U_m = \{u_0, \cdots, u_m\}$ un vecteur de $m+1$ nombres réels tels que :

  $$\forall i \in [0, m], \begin{cases} u_i < u_{i+1} \\ u_i \in [0, 1] \end{cases}$$

  On appellera les u_i *noeuds* (knots) et U_m le *vecteur de noeuds* ou *vecteur nodal* (knot vector).
- Alors :

$$\begin{aligned} N_{i,0}(t) &= \begin{cases} 1 & \text{si } u_i \le t < u_{i+1} \\ 0 & \text{sinon} \end{cases} \\ N_{i,p}(t) &= \frac{t - u_i}{u_{i+p} - u_i} N_{i,p-1}(t) + \frac{u_{i+p+1} - t}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(t) \end{aligned} \tag{2.2}$$

Pour introduire plus naturellement le modèle des NURBS, conformément à [PT95], nous restreindrons notre étude aux courbes B-Splines *extrémales*. Leur vecteur nodal doit alors respecter le format suivant :

$$U_m = \{\underbrace{a, \cdots, a}_{p+1}, \underbrace{u_{p+1}, \cdots, u_{m-(p+1)}}_{n-2}, \underbrace{b, \cdots, b}_{p+1}\} \text{ , avec } a = 0 \text{ et } b = 1.$$

- Le vecteur contient $m+1$ éléments, avec $m = n + p + 1$.

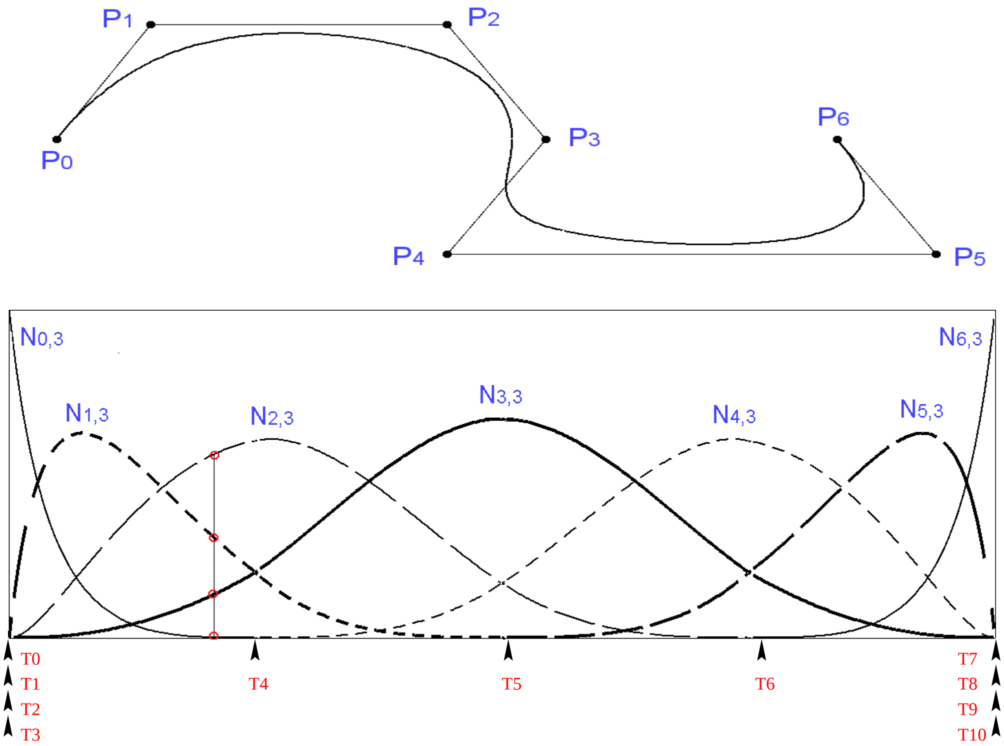

FIG. 2.1 – Exemple de courbe B-Spline extrémale et de ses fonctions d'influence ($p = 3$, et $U_m = \{0, 0, 0, 0, 0.25, 0.50, 0.75, 1, 1, 1, 1\}$)

- Les noeuds a et b sont de *multiplicité* $\mu_a = \mu_b = p + 1$, c'est-à-dire que les valeurs de a et b ont $p + 1$ occurrences dans le vecteur de noeuds.
- Ce vecteur nodal est de type *non-uniforme* : l'intervalle entre deux noeuds consécutifs n'est pas nécessairement constant.
- La multiplicité $p + 1$ des noeuds a et b force la courbe à interpoler le premier et le dernier point du réseau de contrôle (cf [Bla94] pp51–52). De ce fait, $C(0) = P_0$ et $C(1) = P_n$: cette propriété définit le modèle des courbes B-Splines extrémales, dont l'intérêt est que l'on connaît la position des points extrêmes dont le calcul est alors inutile.

Remarque : Dans la mesure où toutes les B-Splines reposent sur les fonctions de Schoenberg, et où les propriétés énoncées par la suite découlent directement de la définition de ces fonctions, le domaine de validité des propriétés qui suivent recouvre tous les modèles de courbes B-Spline .

2.2.2 Propriétés géométriques

- **Localité L^{p+1} :**

  Une fonction d'influence $N_{i,p}(t)$ n'est non-nulle que sur l'intervalle $[u_i, u_{i+p+1}]$ (appelé *portée* de la fonction). Réciproquement, sur tout intervalle $[u_i, u_{i+1}]$, seules $p+1$ fonctions d'influence sont non-nulles : $\{N_{i-p,p}, \ldots, N_{i,p}\}$ (cf figure 2.1.

La modification d'un point de contrôle n'affectera donc qu'une partie de la courbe, dans la mesure où la valeur de la fonction d'influence associée est maximale pour ce point, puis décroît pour devenir nulle en dehors de sa portée. Réciproquement, un point de la courbe est déterminé par $p+1$ points de contrôle et les fonctions d'influence associées.

Cette propriété facilite la modélisation de formes complexes dans la mesure où des ajustements locaux du modèle sont possibles. De plus, nous verrons qu'il est possible d'insérer à la demande de nouveaux points de contrôle (ou des noeuds) afin d'avoir un plus grand degré de liberté lors de la modification d'une section précise de la courbe.

- **Invariance affine :**

  On peut montrer ([PT95] p57) que les fonctions d'influence respectent de par leur définition la propriété de *normalité* :

$$\forall t \in [0,1], \sum_{k=0}^{n} N_{k,p}(t) = 1 \tag{2.3}$$

  De cette propriété découle la propriété d'*invariance par transformation affine* (ou *invariance affine*) : pour appliquer une transformation affine à une courbe B-Spline , il suffit de l'appliquer à son réseau de contrôle puis de calculer la nouvelle B-Spline qui en découle.

  Cette propriété peut être vérifiée de deux manières différentes :

  - Dans [Bla94], de par la normalité, les valeurs des fonctions d'influence sont définies comme coordonnées barycentriques des points de contrôle. La courbe étant une représentation barycentrique du réseau de contrôle, la propriété d'invariance affine découle des propriétés des coordonnées barycentriques.

  - Dans [Pie91], on définit une transformation affine générique A comme combinaison d'une transformation linéaire L (e.g. rotation, zoom) et d'une translation T. Une transformation affine, associant un point P' à tout point P du plan, se caractérise alors par : $P' = A[P] = L[P] + T$.

    En développant cette égalité, on parvient (cf 6.2 pour démonstration) à vérifier la propriété d'invariance affine.

- **Enveloppe Convexe :**

  - Une courbe B-Spline est une *représentation barycentrique* de son réseau de contrôle :

    Le barycentre $B_{P_0,\ldots,P_n}$ des points $\{P_0, \ldots, P_n\}$ pondérés par les coefficients $\{c_0, \ldots, c_n\}$ est défini comme suit :

$$B_{P_0,\ldots,P_n} = \frac{\sum_{k=0}^{n} c_k P_k}{\sum_{k=0}^{n} c_k} \tag{2.4}$$

    Pour une valeur de t, déterminons le barycentre des points de contrôle d'une courbe B-Spline pondérés par les valeurs pour t des fonctions d'influence associées :

$$B_{P_0,\dots,P_n} = \frac{\sum_{k=0}^{n} N_{k,p}(t)P_k}{\underbrace{\sum_{k=0}^{n} N_{k,p}(t)}_{=1 \text{ par normalité}}} = C(t) \tag{2.5}$$

– Les fonctions d'influence vérifient la propriété de *positivité* :

$$\forall t \in [0,1], N_{k,p}(t) \geq 0 \tag{2.6}$$

En effet, de par la définition récursive des fonctions d'influence (équation 2.2), $N_{i,p}(t)$ s'écrit sous forme de combinaison linéaire de termes $N_{j,0}(t)$ positifs (0 ou 1) dont les coefficients rationnels s'avèrent tous positifs (cf [PT95] pp56–57 pour preuve détaillée).

Nous savons que le barycentre de points pondérés par des coefficients strictement positifs est confiné dans l'enveloppe convexe de ces points. De par la positivité des fonctions d'influence, on peut donc affirmer qu'une courbe B-Spline est toujours comprise dans l'enveloppe convexe formée par les points de contrôle (i.e. le *polygone de contrôle*).

Soit une *section de courbe* : $\{C(t)\} \setminus t \in [t_k, t_{k+1}]$ avec $\{t_k, t_{k+1}\} \in U_m$. Du fait de la localité des fonctions d'influence, seules les fonctions $\{N_{k-p,p}, \dots, N_{k,p}\}$ sont non-nulles sur cet intervalle. Sachant qu'un barycentre n'est pas affecté par des points de coefficients nuls, on peut donc affirmer que la section de courbe est une représentation barycentrique des points de contrôles associés aux fonctions d'influence de la portée.

D'où la propriété d'*enveloppe convexe stricte* :
– Toute section de courbe B-Spline est comprise dans l'enveloppe convexe des $p+1$ points de contrôle associés aux fonctions d'influence de sa portée.
– En conséquence, une courbe B-Spline est comprise dans l'union des enveloppes convexes de ses sections.

- **Régularisation des oscillations :**

Les fonctions d'influence vérifient la propriété de *régularité* :
– Toute fonction d'influence $N_{i,p}(t)$ admet un maximum unique pour une valeur $t_i \in [0,1]$.
– Lorsque deux fonctions d'influence distinctes $\{N_{i,p}, N_{j,p}\}$ (avec $i < j$) sont égales pour une valeur de $t \in [0,1]$, alors $t \in [t_i, t_j]$.

Farin [Far90] montre que la régularité des fonctions d'influence est une condition suffisante pour obtenir la propriété de *régularisation des oscillations* (variation-diminishing property) :
Dans l'espace $\mathbb{R}^3$, le nombre d'intersections entre un plan et la courbe ne peut excéder celui entre ce plan et le réseau de contrôle de la courbe.
Cette propriété est aussi valide dans le plan $\mathbb{R}^2$ pour une droite de ce plan et la courbe. Une preuve en est fournie dans [LR83].

En conséquence, le réseau de contrôle représente une approximation correcte de la courbe dans la mesure où cette dernière sera exempte d'oscillations parasites à l'intérieur du réseau de contrôle. Cette approximation est déjà nettement plus précise que la boîte englobante d'une région CLIM. Nous verrons par la suite qu'il existe une grande variété de méthodes pour améliorer cette approximation jusqu'à lui faire "coller" à la courbe.

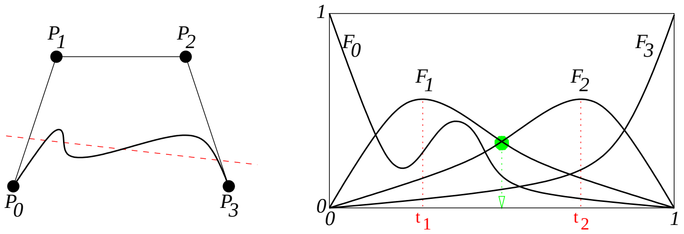

FIG. 2.2 – Exemple de courbe Spline planaire non-régulière.

Sur la figure 2.2, on peut constater que F_1, F_2 et F_3 sont des fonctions d'influence régulières. Ce n'est pas le cas de F_0 : elle possède deux maximums, le deuxième étant "parasite". En effet, il attire une nouvelle fois la courbe vers P_0, entraînant l'oscillation constatée. La propriété de régularisation des oscillations n'est donc pas vérifiée pour cette courbe. Nous pouvons aussi le constater géométriquement en trouvant une droite du plan telle celle indiquée sur la figure.

- **Continuité** $\mathcal{C}^{p-\mu_k}$:

Pour toute valeur k d'un noeud, une courbe B-Spline d'ordre p (de degré $n = p - 1$) est dérivable $\mathcal{C}^{p-\mu_k}$ fois. Elle est indéfiniment dérivable en toute valeur du paramètre $t \in [0, 1]$ qui n'est pas valeur d'un noeud.

De ce fait, avec des splines de degré 3, on peut obtenir une continuité paramétrique $\mathcal{C}^2$ aux points de jonction des morceaux de courbe. Cette continuité est souhaitable dans la mesure où elle nous épargne des changements brusques des vecteurs vitesse et accélération lors du parcours de la courbe avec un pas de t régulier. Cependant, elle ne nous renseigne en rien sur la forme de la courbe, puisqu'elle se base sur les valeurs du paramètre t.

La qualité visuelle de la courbe est en réalité assurée par la continuité géométrique G^n, déterminée à partir de l'abcisse curviligne de la courbe. Les courbes B-Splines assurent une continuité G^2 dans le cas général, c'est-à-dire une continuité des vecteurs de tangence et de courbure. Cependant, des courbes à continuité géométrique posent certains problèmes de modélisation ou d'utilisation.

Nous ne souhaitons pas développer plus avant cette section (cf [Bla94] pp23–25 pour une présentation exhaustive du domaine). Le point qui nous intéresse est que le modèle B-Spline représente un bon compromis entre coût de calcul, qualité visuelle et viabilité en modélisation géométrique. La continuité du modèle suffit à nous assurer que dans le cas général les courbes ont un aspect "doux" aux points de jonction, et de ce fait ne ressemblent pas à la concaténation de morceaux de courbes hétérogènes.

Remarque :
Nous avons donc décidé de travailler avec des B-Splines de degré $p = 3$ (B-Splines *cubiques*) : cette valeur est la plus petite qui satisfasse nos exigences de continuité, sachant qu'un degré supérieur augmenterait inutilement le temps de calcul.

2.2.3 Restrictions

Le modèle B-Spline permet de modéliser une grande variété de courbes dans des conditions satisfaisantes. Ce modèle repose sur des fonctions de base définies à partir de polynômes de degré n. Cette définition implique cependant qu'une famille de courbes ne sera pas représentable par ce modèle : celle des *coniques*, qui comprend notamment les cercles et les ellipses.

En effet, Les Piegl démontre ([PT95] p25) qu'il est impossible de représenter l'équation paramétrique d'un point sur un cercle à partir d'un polynôme de degré quelconque (cf Annexe 6.3 pour preuve). En conséquence, même s'il est possible de représenter des formes proches des coniques à partir de courbes B-Spline fermées, on n'obtiendra jamais qu'une approximation dont la qualité dépend du degré de la courbe.

De par la nécessité de représenter des coniques sous CLIM, et ce de manière exacte, le modèle B-Spline est insuffisant.

Cependant, il est bien connu en géométrie classique que les courbes coniques peuvent être représentées à partir de quotients de polynômes. Reconsidérons en effet le cas du cercle (cf [Bla94] p 65) :

$$\begin{cases} x(\varphi) = cos(\varphi) \\ y(\varphi) = sin(\varphi) \end{cases} \Leftrightarrow \begin{cases} x(t) = \dfrac{1-t^2}{1+t^2} \\ y(t) = \dfrac{2t}{1+t^2} \end{cases} \text{avec } t = tan(\frac{\varphi}{2}) \tag{2.7}$$

Des polynômes rationnels suffisent à définir l'équation paramétrique du cercle. Nous pouvons remarquer que :

- d'une part il est connu en mathématiques que les fonctions *sinus* et *cosinus* ne peuvent être représentées sous formes de polynômes.
- d'autre part l'évaluation de telles fonctions a un coût prohibitif, ce qui n'est pas le cas des polynômes en général, et surtout pas le cas des fonctions de base.

Pour répondre à l'impératif de représentation des courbes coniques, nous pourrions faire coexister le modèle B-Spline avec de nouvelles primitives définies à partir d'équations paramétriques exactes telles 2.7. Cependant, un tel système hybride est difficile à gérer et à implémenter (dixit [Sil90]). Nous allons donc plutôt adopter un modèle à base de polynômes rationnels, construit à partir des B-Splines et impliquant des calculs de même ordre de grandeur : le modèle des NURBS.

2.3 Courbes NURBS

2.3.1 Définition

Soit une courbe B-Spline non-uniforme et extrémale (généralisation non-uniforme du modèle étudié dans la section précédente) définie dans l'*espace homogène* $\mathbb{R}^4$: on appelle courbe *NURBS* (Non-Uniform Rational B-Spline) la projection de cette courbe dans le repère Euclidien $\mathbb{R}^3$.

Les point de contrôle que l'on doit définir sont les points P_k de l'espace homogène $\mathbb{R}^4$: ils auront pour forme $P_k : (x_k w_k, y_k w_k, z_k w_k, w_k)$, où w_k est le poids du point P_k.

Nous pouvons illustrer la relation entre repère homogène $\mathbb{R}^4$ et repère associé $\mathbb{R}^3$ par les remarques suivantes :

Soient $P_E : (x, y, z) \in \mathbb{R}^3$ et $P_H : (xw, yw, zw, w) \in \mathbb{R}^4$.

- Tout point P_H : (xw, yw, zw, w) de l'espace homogène évolue sur la droite (O, P_{H_1}), avec O : $(0, 0, 0, 0)$ et $P_{H_1} : (x, y, z, 1)$.
- L'espace homogène $\mathbb{R}^4$ coïncide avec l'espace Euclidien $\mathbb{R}^3$ associé lorsque la coordonnée w de P_H est fixée à 1.
- En conséquence, $P_E : (x, y, z)$ est la projection Euclidienne de $P_H : (x, y, z, 1)$.
- Dans le cas général, $P_H : (xw, yw, zw, w)$ se projette alors en $P_E : (x, y, z)$.

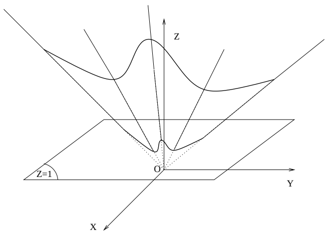

FIG. 2.3 – Repère Euclidien $\mathbb{R}^2$ associé à l'espace homogène $\mathbb{R}^3$

La figure 2.3 illustre ces notions avec une courbe B-Spline définie dans l'espace homogène $\mathbb{R}^3$, se projetant dans le repère Euclidien $\mathbb{R}^2$ associé sous forme d'une courbe NURBS.

Dans l'espace homogène $\mathbb{R}^4$, un point sur une courbe B-Spline de *degré* p (d'*ordre* p+1) est défini comme suit :

$$C(t) = \sum_{k=0}^{n} N_{k,p}(t)\widehat{P_k} \tag{2.8}$$

- $\widehat{P}$ étant le point de contrôle de $\mathbb{R}^4$ de coordonnées (xw, yw, zw, w).
- P étant le point de contrôle associé dans $\mathbb{R}^3$ par (x, y, z).
- les $N_{k,p}(t)$ étant les fonctions de Schoenberg des courbes B-Spline , définies sur un vecteur nodal non-uniforme de même format que celui d'une courbe B-Spline extrémale.

De ce fait, le point $C(t)$ aura pour coordonnées dans $\mathbb{R}^4$:

$$\left(\sum_{k=0}^{n} N_{k,p}(t) x_k w_k ; \sum_{k=0}^{n} N_{k,p}(t) y_k w_k ; \sum_{k=0}^{n} N_{k,p}(t) z_k w_k ; \sum_{k=0}^{n} N_{k,p}(t) w_k \right) \tag{2.9}$$

Sa projection dans $\mathbb{R}^3$ sera donc :

$$\left(\frac{\sum_{k=0}^{n} N_{k,p}(t)x_k w_k}{\sum_{k=0}^{n} N_{k,p}(t)w_k};\frac{\sum_{k=0}^{n} N_{k,p}(t)y_k w_k}{\sum_{k=0}^{n} N_{k,p}(t)w_k};\frac{\sum_{k=0}^{n} N_{k,p}(t)z_k w_k}{\sum_{k=0}^{n} N_{k,p}(t)w_k}\right) \tag{2.10}$$

Dès lors, on peut définir l'équation d'un point sur une courbe NURBS dans $\mathbb{R}^3$ comme suit :

$$C(t) = \frac{\sum_{k=0}^{n} N_{k,p}(t)w_k P_k}{\sum_{k=0}^{n} N_{k,p}(t)w_k} \tag{2.11}$$

On en déduit les fonctions d'influence des NURBS :

$$R_{k,p}(t) = \frac{w_k}{\sum_{k=0}^{n} N_{k,p}(t)w_k} N_{k,p}(t) \tag{2.12}$$

La présence des fonctions de Schoenberg dans la définition de ces fonctions d'influence permet aux courbes NURBS d'hériter de toutes les propriétés des courbes B-Spline . Mais la spécificité du modèle rationnel nous offre des propriétés additionnelles que nous allons détailler.

2.3.2 Propriétés additionnelles

- **Représentation exacte des sections coniques :**

  Celle-ci est enfin rendue possible grâce au caractère rationnel des fonctions d'influence.

- **Invariance par projection parallèle ou perspective :**

  Pour projeter une courbe NURBS, il suffit de projeter les points de son réseau de contrôle et de tracer la courbe à partir des points obtenus (cf [Pie91] p. 59 pour preuve).

  Les projections parallèle et perspective interviennent dans le cadre du rendu en 2D de modèles 3D. L'intérêt pour CLIM n'est donc pas évident, celui-ci ne se préoccupant a priori que de modèles 2D. Cependant, cette propriété est capitale pour la viabilité du modèle dans le cadre de la modélisation d'objets 3D, et en conséquence contribua grandement à établir la popularité des NURBS dans ce domaine.

2.3.3 Atouts du modèle

- **Un degré de liberté supplémentaire :**

  Pour modifier la forme d'une courbe B-Spline , nous avons le choix entre le déplacement de points de contrôle, qui donne des résultats visuellement prévisibles et satisfaisants, et la modification du vecteur nodal, qui affecte la courbe sans rapport avec un quelconque repère visuel. Une excellente compréhension du modèle est indispensable pour employer cette dernière méthode.

  Le fait de disposer d'un poids à affecter aux points de contrôle va nous fournir un degré de liberté supplémentaire pour la modélisation. De plus, ce facteur est assez visuel puisqu'il est en rapport avec les points de contrôle : on peut déjà supposer que la valeur du poids va attirer ou repousser la courbe du point affecté. Etudions précisément l'influence du poids d'un point de contrôle sur la courbe :

Comparons l'équation 2.11 d'un point d'une courbe NURBS avec l'équation 2.5 qui définit un point d'une courbe B-Spline comme barycentre de son réseau de contrôle. Nous pouvons constater qu'un point d'une NURBS est aussi une représentation barycentrique de son réseau de contrôle. Mais dans ce cas, le coefficient de chaque point de contrôle est le produit de la valeur de la fonction d'influence associée au point par le poids du point. Comme précisé auparavant, l'un des intérêts du modèle NURBS est que l'utilisateur peut modifier la valeur du poids, et donc du coefficient barycentrique du point de contrôle.

En conséquence, de par les propriétés des barycentres, plus on augmente le poids d'un point de contrôle, plus la courbe se rapprochera de ce point. Inversement, diminuer la valeur du poids va repousser la courbe de ce point (cf figure 2.4). Aussi, la présence de poids négatifs va rendre les coefficients associés négatifs et repousser la courbe hors de l'enveloppe convexe des points de contrôle. Pour plus de détails, c.f [Bla94] pp49–51 et [Pie89].

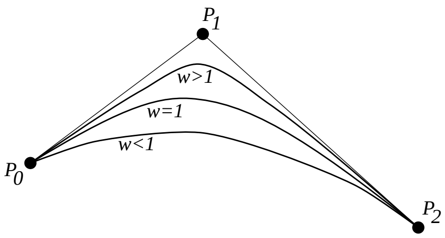

FIG. 2.4 – Influence du poids du point de contrôle P_1 sur une courbe NURBS de degré 2

Par exemple, une application intéressante est décrite dans [Pie89], présentant un algorithme qui calcule le nouveau poids d'un point de contrôle telle que la courbe se rapproche ou s'éloigne du réseau de contrôle d'une distance fixée en suivant un axe déterminé par l'utilisateur (l'axe reliant un point de la courbe à un point du réseau).

- **Un standard en modélisation géométrique :**

La grande variété des formes (2D/3D) exprimables par les NURBS et son caractère fédérateur en ont fait un modèle standard d'échanges de données géométriques (International Graphics Exchange Standard, 1986). En effet, la quasi totalité des courbes et surfaces employées en modélisation géométrique peuvent être représentées par ce même modèle.
De ce fait, le modèle NURBS est devenu en 1992 le seul standard ISO pour l'échange de données destinées à la production industrielle.

- **Un modèle disposant d'une littérature fournie :**

Du fait de la popularité des NURBS et de leur parenté avec les B-Splines , nous disposons d'un vaste choix de méthodes éprouvées, notamment pour la modélisation, la modification, le tracé et le calcul d'intersections de nos courbes.

2.4 Conclusion

Il s'avère donc que parmi les modèles géométriques répondant à nos critères[2], celui des NURBS est le seul qui permette d'unifier sous une même représentation la famille des formes géométriques que les régions CLIM peuvent déjà décrire.

Un atout majeur de ce modèle est qu'il représente le standard dans le domaine de la modélisation géométrique, ce qui implique l'existence d'une base documentaire conséquente qui pourra guider notre recherche. Nous avons d'ailleurs pris la précaution de vérifier que la détermination d'intersections entre courbes (ou surfaces) NURBS était traitée de manière exhaustive dans la littérature : nous avons en effet absolument besoin de ces opérations afin de représenter les compositions de régions NURBS sous forme de NURBS.

En conséquence, nous allons maintenant pouvoir nous concentrer sur notre objectif principal : le tracé du contours de régions.

[2] modèle par morceau, polynômial, paramétrique

Chapitre 3

Méthodes de tracé des NURBS

Maintenant que nous avons arrêté notre choix sur les NURBS comme modèle géométrique pour représenter les régions CLIM, nous pouvons entreprendre une vaste recherche documentaire afin d'avoir une idée globale des méthodes qui s'y appliquent. Dans la mesure où le modèle NURBS dérive du modèle B-Spline et peut représenter par exemple des courbes de Bézier, le champ de nos recherches doit aussi recouvrir ces deux modèles.

Notre principale préoccupation concerne maintenant le tracé des contours de régions CLIM, voire même d'autres objets, tels les caractères de polices, qui tireraient parti du modèle géométrique que nous venons d'adopter. Nous allons donc entreprendre un inventaire des méthodes de tracé de NURBS évoquées dans la littérature. Malheureusement, une étude exhaustive de toutes les techniques employées demanderait trop de temps dans le cadre d'un mémoire de DEA, c'est pourquoi nous nous en tiendrons essentiellement à la description des principales familles d'algorithmes, quitte à en détailler ultérieurement quelques uns parmi les plus utilisés.

Auparavant, il est nécessaire d'examiner le contexte dans lequel ces algorithmes vont devoir opérer : de ce contexte vont découler des premières contraintes que ces algorithmes vont devoir respecter, ce qui va nous permettre de restreindre le champ de nos investigations.

Pourtant, le but de ce chapitre n'est pas d'élire les méthodes de tracé à employer dans notre contexte.

Il paraîtrait en effet raisonnable de chercher en priorité les méthodes les plus efficaces. Cependant, il faudrait d'abord étudier avec précision la quantité d'opérations effectuées par chaque algorithme, tâche d'autant plus ardue qu'il en existe souvent plusieurs variantes pour chaque méthode. Ensuite, il conviendrait d'optimiser ces algorithmes supposés critiques, ce qui compliquerait encore notre étude tant que des modalités communes n'auraient pas été fixées. Enfin, rien ne nous indique a priori que nous aurons tout le temps besoin d'algorithmes de tracé rapides en temps réel[1].

Toutes ces incertitudes indiquent que nous devrions nous contenter de présenter, dans la mesure du possible, les différentes méthodes de tracé adaptées à notre contexte sans chercher à les comparer dans l'immédiat.

[1]référence à des courbes invariantes dont on pourrait pré-calculer à l'avance une approximation.

3.1 Contexte de l'affichage

Nous souhaitons étendre la bibliothèque FreeCLIM, application cliente de X, d'un module de tracé de courbes NURBS. Nous travaillons donc dans le cadre d'une session X en mode client/serveur. Dans le cas général, l'application cliente et le serveur X ne sont pas sur la même machine et doivent donc communiquer par le réseau : nous devrons porter notre attention autant sur la rapidité de calcul de notre module NURBS que sur la rapidité du transfert des paquets X décrivant la courbe et destinés au serveur.

Les distributions X standard sont capables d'afficher avec une efficacité optimale des segments de droite, des polygones, des ellipses "orthogonales", des arcs elliptiques qui en découlent, et par restriction des cercles et arcs de cercle. Elles savent utiliser de manière transparente les fonctions de tracé accélérées éventuellement proposées par la carte graphique si son pilote, intégré au serveur X, le permet. Notons au passage l'absence de courbes elliptiques quelconques, qui va restreindre le champ des formes modélisables et approximables.

Nous pouvons dès lors remarquer qu'un objet défini à partir d'un nombre raisonnable de ces primitives sera non seulement affiché efficacement, mais aussi transmis à travers le réseau en peu de temps, même sur un réseau à faible débit. De ce fait, un grand nombre d'applications simples (i.e. sans images) peuvent être programmées afin de consommer peu de ressources machine et réseau.

Le modèle géométrique des NURBS n'étant pas supporté par X, nous allons dans un premier temps envisager différentes possibilités d'implémenter un module de NURBS qui viendrait compléter les primitives de tracé du système X, ce qui nous éviterait alors de transmettre une version discrète (et donc volumineuse) de la courbe par le réseau :

- **Extension du protocole X :**

  Si le serveur X pouvait être complété d'un module d'affichage de NURBS, il suffirait simplement à FreeCLIM de transmettre la définition de la courbe (réseau de points de contrôle et vecteur de noeuds) au serveur de la machine d'affichage pour effectuer le tracé. On pourrait alors envisager d'en paramétrer la précision en fonction de la puissance de la machine hébergeant le serveur X.

  Il s'avère qu'il existe un protocole normalisé d'extension de X, permettant de développer des modules étendant les primitives de tracé disponibles. Malheureusement, le chargement de modules dynamiques d'extensions dans le serveur X sur demande du client n'est pas normalisé. En conséquence, nous ne pourrons pas assurer le chargement de notre module NURBS sur tous les serveurs X, ce chargement dépendant de l'implémentation du serveur utilisé.

- **Usage de primitives *OpenGL* :**

  OpenGL et ses extensions forment une bibliothèque de construction, de manipulation et d'affichage d'objets 3D. Développée par SGI, c'est le standard du domaine ; nous pouvons remarquer qu'une implémentation gratuite de cette bibliothèque est disponible pour Linux sous le nom de MesaGL.

  Lors de la procédure `open-display` initialisant l'affichage du serveur X, nous pouvons interroger ce dernier sur les extensions dont il dispose, afin éventuellement d'adapter notre stratégie de tracé en conséquence. Plus particulièrement, la version 4.0 de Xfree86 dispose d'extensions OpenGL, ce qui est susceptible de nous intéresser dans la mesure où son extension "GLU Toolkit" propose des fonctions de tracé de NURBS dont la qualité et l'efficacité sont encore à vérifier.

Pour les mêmes raisons que l'hypothèse précédente, une telle démarche n'est pas valide dans le cas général. En effet, nous souhaitons pouvoir faire fonctionner FreeCLIM sur toutes les versions de serveurs X. Nous ne l'envisagerons donc pas dans un premier temps, mais peut-être sera-t-il judicieux de tirer parti de cette fonctionnalité lorsqu'elle sera disponible.

Il s'avère que, dans le cas général, on ne peut pas compter sur des fonctionnalités auxiliaires mises à disposition du serveur X. En conséquence, nous allons devoir tracer nos courbes NURBS à partir des primitives X, et transmettre sa version discrète à travers le réseau. Envisageons alors les deux scénarios possibles :

- **Transmission des courbes pixel par pixel :**

  Cette hypothèse requiert a priori un important volume de données, dans la mesure où il faut transmettre le contour discret de la courbe. Elle pourrait cependant être viable si le serveur X et l'application cliente se trouvaient sur la même machine. Dans ce cas, les paquets de données X circulent par le bus interne, rendant leur temps de transport négligeable. De même, il existe des protocoles permettant à l'application cliente de communiquer avec le serveur via une mémoire graphique partagée.

  Mais dans le cas général, l'application cliente pourra aussi bien se situer sur une machine distante de celle hébergeant le serveur X, comme c'est le toujours le cas pour les terminaux X. Il faudrait alors faire transiter un important volume de données par le réseau, ce qui ne serait envisageable que si l'on se confinait à des réseaux à haut débit.

- **Transmission d'une approximation de courbe :**

  Une courbe NURBS peut être approximée par une série de segments de droite : nous verrons par la suite plusieurs méthodes pour réaliser ce processus. Les segments de droite étant des primitives X optimales en temps de tracé et de transmission par le réseau, cette hypothèse est viable dans tous les cas.

  Nous n'obtiendrons au final qu'une approximation polygonale de la courbe. Cependant, nous aurons la liberté de définir la qualité de l'approximation, dépendant notamment du nombre de segments utilisés. Nous pouvons adapter cette qualité en fonction de nos besoins ou de la capacité du réseau.

  Remarque : Les arcs elliptiques pourraient fournir des approximations de NURBS plus fidèles que celles réalisées à partir de segments de droite. Cependant, dans la mesure où X ne gère que des ellipses orthogonales, la mise en oeuvre de cette suggestion n'est pas triviale. En effet, nous pouvons remarquer qu'il est possible d'obtenir une approximation satisfaisante d'un arc elliptique quelconque à partir de sections d'arcs d'ellipses orthogonales. Cependant, il reste encore à déterminer quels fragments de quelles ellipses il faut employer.

Actuellement, il semble que seule l'option d'une approximation de courbe par segments de droite soit envisageable pour répondre à toutes nos contraintes.

3.2 Objets à afficher

Dans le cadre de FreeCLIM, nous projetons d'employer les NURBS pour modéliser les objets suivants :

- **Régions CLIM :**

  Elles représentent des zones délimitant l'affichage du contenu d'une fenêtre X. Remarquons que ces fenêtres peuvent être déplacées, redimensionnées, voire combinées en cours de session X. En effet, quand deux fenêtres se chevauchent, il est nécessaire d'effectuer une différence de leurs régions pour déterminer leurs zones à afficher. En conséquence, les représentations discrètes des régions vont probablement devoir être calculées en temps réel.

  Mais dans la mesure où la forme des fenêtres n'est pas tellement susceptible d'évoluer en cours de session, il est aussi possible d'en pré-calculer une représentation discrète et de lui faire subir ces transformations en temps réel. Ainsi on pourrait réduire considérablement la quantité de calcul effectuée en temps réel tout en maintenant une qualité de tracé acceptable.

- **Caractères de polices :**

  Ces objets sont définis de manière fixe en taille et en forme avant le lancement d'une session X sous une résolution donnée. En supposant que cette résolution ne soit jamais ou peu modifiée en cours de session, on peut parfaitement envisager de pré-calculer et de stocker une représentation discrète de chaque caractère. C'est d'autant plus pertinent dans la mesure où la plupart d'entre eux seront affichés à plusieurs reprises : il serait alors inadéquat de les recalculer à chaque fois.

  Cependant, on ne peut pas exclure l'hypothèse selon laquelle on aurait besoin de modifier la taille ou l'orientation d'une police de caractères au cours de la session X, ce genre d'opérations étant d'ailleurs relativement courant dans le cadre de l'utilisation d'un traitement de texte (par exemple). Il paraîtrait alors assez déraisonnable voire impossible de stocker des représentations discrètes des caractères pour toutes les valeurs pertinentes de ces paramètres.

A ce stade, tout ce que nous savons est que nous sommes aussi bien susceptibles de tracer des représentations discrètes de courbes NURBS avec des contraintes de temps réel que d'en pré-calculer à l'avance. Il faudrait que le projet arrive au stade de l'implémentation de méthodes de tracé dans FreeCLIM pour confronter ces conjectures à la réalité et nous permettre de poser des hypothèses plus tranchées.

Ces dernières informations ne peuvent pas dans l'immédiat nous aider à sélectionner une méthode de tracé plutôt qu'une autre.

Nous allons donc pouvoir présenter la plupart des méthodes susceptibles de calculer des approximations polygonales de courbes NURBS. Nous ne devrions pas écarter toutes les méthodes de tracé point à point, dans la mesure où un échantillon de points espacés détermine implicitement une approximation polygonale. Cependant, nous pouvons déjà écarter (provisoirement) les méthodes basées exclusivement sur un tracé pixel par pixel (dont [AP92], entre autres).

Nous pouvons aussi écarter les méthodes de tracé qui s'efforcent de travailler avec des nombres entiers plutôt qu'avec les flottants, dont nous avons naturellement besoin dans nos calculs (valeurs du paramètre, du vecteur nodal, du réseau de contrôle). Ces méthodes (dont [AP92]) ont pour la plupart été publiées il y a plus de 8 ans, c'est à dire à une époque ou les opérations flottantes coûtaient plus de dix fois plus cher que les opérations entières. Depuis, les unités à virgule flottante des processeurs ont beaucoup progressé, et l'écart s'est considérablement réduit. C'est pourquoi les astuces employées pour rester dans le domaine entier risquent aujourd'hui de rendre ces méthodes plus lentes que leurs implémentations flottantes directes (c'est d'ailleurs certainement le cas de [AP92]).

Cependant, Les critères de qualité de ces approximations, éventuellement variables selon le contexte d'exécution (dont la configuration réseau), restent encore à définir.

3.3 Méthodes d'évaluation d'un point de la courbe

Etant donnée une valeur du paramètre t, ces méthodes nous permettent de calculer directement la position du point $C(t)$ de la courbe. On distingue les méthodes algébriques, qui déterminent la position du point en calculant explicitement les valeurs des fonctions d'influence, des méthodes géométriques qui se basent sur une relation de récurrence entre les points de contrôle.

Remarquons cependant que l'opération inverse n'est pas triviale (cf [PT95], pp229–234). En effet, quand cela est possible, elle produit essentiellement une approximation de t après une recherche dichotomique.

3.3.1 Méthodes algébriques

Elles se basent sur l'application littérale de la définition des courbes NURBS, et donc évaluent explicitement les fonctions d'influence pour une valeur donnée du paramètre t.

3.3.1.1 Schéma général :

- Trouvons à quel intervalle de noeuds $[u_i, u_{i+1}]$ appartient la valeur spécifiée du paramètre t.

- Ceci fait, évaluons en t les fonctions d'influence $\{N_{i-p,p}, \ldots, N_{i,p}\}$ valides sur la portée de cet intervalle (remarquons qu'il n'y a que 4 fonctions à évaluer pour des NURBS cubiques).

- Puis, par application des équations 2.8 et 2.10, calculons ensuite le point de l'espace homogène $\mathbb{R}^4$ par moyenne pondérée des $p+1$ points de contrôle associés aux fonctions d'influence, puis projetons-le dans l'espace Euclidien $\mathbb{R}^3$.

3.3.1.2 Etude d'une implémentation efficace :

Une méthode efficace est détaillée dans [PT95] (pp68–70) : elle procède à une recherche binaire ($\theta(log(N))$ pour trouver l'intervalle $[u_i, u_i + 1]$, puis évalue les fonctions de Schoenberg de manière économe (cf section 4.3.1 pour une version du code). Voici un aperçu des méthodes d'optimisations employées (cf [PT95] pour plus de détails) :

- Chaque terme $N_{k,p-x}(t)$ n'est développé qu'une fois, quand bien même il est requis par le calcul d'une autre fonction d'influence. Il est alors mémorisé le temps nécessaire dans un tableau.

- Les coefficients des fonctions d'influence sont des nombres rationnels. On remarque qu'en appliquant la relation de récurrence sur deux termes qui, en se développant, font apparaître des fonctions d'influence identiques, les dénominateurs de leurs coefficients sont toujours identiques. C'est pourquoi ces derniers sont aussi mémorisés.

- Nous savons déjà que : $\forall k \neq i, N_{k,0}(t) = 0$. Posons alors :

  $$N_{k,1}(u) = A \cdot N_{k,0}(u) + B \cdot N_{k+1,0}(u), \text{ avec } A, B \in \mathbb{R}$$

  Nous pouvons constater que si $k \in \{i-1, i\}$, seul un des deux termes de l'expression sera nul. En effet :

  $$N_{i,1}(u) = A \cdot N_{i,0}(u) \text{ , et } N_{i-1,1}(u) = B \cdot N_{i,0}(u)$$

  Par conséquent, $N_{k,1}(u)$ est nul pour toute valeur de k hors de l'intervalle $[i-1, i]$

L'algorithme tient compte de toutes ces observations pour proposer une implémentation (non-spécialisée) optimale de cette méthode (cf section 4.3.1 pour une version du code).

Nous pouvons remarquer que l'algorithme d'évaluation des fonctions d'influence dérive directement de celui indiqué par Carl De Boor dans l'article [Boo72] (pp56–58) qui présentait, en concurrence avec [Cox72], la définition "moderne" des B-Splines .

3.3.2 Méthodes géométriques

Elles procèdent par *corner-cutting* pour évaluer un point de la courbe pour une valeur donnée du paramètre t : on calcule les moyennes successives de couples de points issus du polygone de contrôle jusqu'à ce qu'il ne reste plus qu'un point, alors situé sur la courbe. Comme il n'est pas nécessaire d'évaluer les fonctions d'influence, le calcul est sensiblement plus simple et probablement plus rapide que celui des méthodes algébriques.

3.3.2.1 Courbes de Bézier : algorithme de *de Casteljau*

Selon [PT95] (p24), Boehm [BFJ84] et De Casteljau [dC86, dC93] ont défini un algorithme simple pour déterminer un point sur une courbe de Bézier de degré p (cf section 4.3.2 pour une version du code) :

Soit la valeur du paramètre t donnée :

- On place sur chacun des p segments du polygone de contrôle $\{\bigcup_{j=0}^{p-1}[P_{0,j}, P_{0,j+1}]\}$ un point $P_{1,j}$ à distance $Longueur([P_{0,j}, P_{0,j+1}]) \cdot t$.
- On forme un polygone de $p-1$ segments à partir des points $\bigcup_{j=0}^{p-2} P_{1,j}$ précédemment calculés, et on l'utilise pour poursuivre récursivement le processus jusqu'à ce que le polygone résultant soit réduit à un seul segment $[P_{p-1,0}, P_{p-1,1}]$.
- Le point de la courbe est $P_{p,0}$ (cf figure3.1).

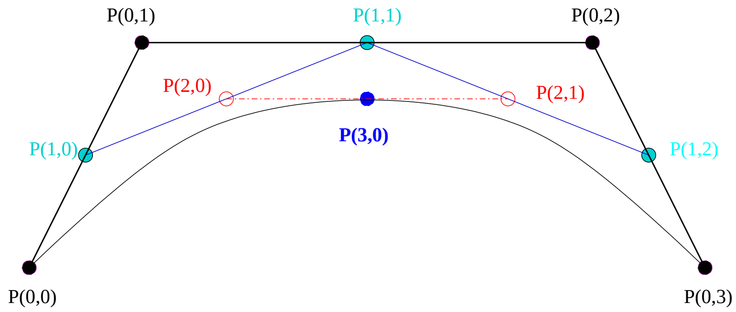

FIG. 3.1 – Corner-cutting par l'algorithme de de Casteljau ($t = 0.5$).

On peut remarquer sur la figure 3.1 que les points calculés forment une enveloppe convexe décrite par les points $\{\{\bigcup_{i=0}^{p} P_{i,0}\} \bigcup \{\bigcup_{j=p-1}^{0} P_{j,p-j}\}\}$ plus proche de la courbe que le polygone de contrôle original.

3.3.2.2 Courbes B-Splines et NURBS : algorithme de *de Boor*

Carl de Boor a défini une variante de cette méthode pour traiter le cas des B-Splines. L'extension aux NURBS est immédiate : il suffit d'opérer le corner-cutting avec les points de contrôle dans l'espace homogène $\mathbb{R}^4$ puis de projeter le point obtenu en $\mathbb{R}^3$.

Soit une valeur de t donnée, il faut d'abord déterminer l'intervalle du vecteur nodal auquel t appartient, puis considérer les $p+1$ fonctions d'influence valides sur la portée de cette intervalle. Nous utilisons alors un algorithme similaire au précédent qui opère sur le polygone de contrôle associé aux fonctions d'influence de la portée. La différence par rapport à l'algorithme de Casteljau vient du fait que l'on coupe chaque segment suivant un rapport toujours différent qui dépend du vecteur nodal.

D'après [Sil90] (p340) citant [dB78], une combinaison linéaire de B-Splines peut être calculée à partir d'une relation de récurrence sur ses points de contrôle, et ce sans avoir recours aux fonctions d'influence associées :

$$\forall k \in [2, p], \text{ Soit } \alpha = \frac{t - u_i}{u_{i+p-1} - u_i}, \text{ alors } P_i^{[k]} = \alpha P_i^{[k-1]} + (1-\alpha) P_{i-1}^{[k-1]} \tag{3.1}$$

u_i étant le i-ième élément du vecteur nodal de la B-Spline .

Une instance de l'algorithme figure dans [PT89] (p55), reproduite dans la section 4.3.3 : Nous pouvons constater qu'il se base sur cette propriété.

3.4 Méthodes de tracé d'une courbe complète

Ces méthodes calculent une représentation discrète de la totalité de la courbe qui, selon le nombre d'itérations effectué, peut aussi bien prendre la forme d'une suite de pixels connexes que d'une série de points espacés déterminant une approximation polygonale (notre hypothèse prioritaire).

3.4.1 Méthodes directes

Leur principe est de calculer un échantillon de points de la courbe à différentes valeurs du paramètre t, chaque point étant calculé indépendamment par une méthode d'évaluation de points de la courbe. Dans le cas général où on peut supposer que la courbe ne comporte pas de fortes variations locales, il paraît raisonnable d'exiger que les points calculés soient uniformément espacés sur la courbe afin d'en déterminer une approximation fidèle.

3.4.1.1 Schéma général

- On se fixe un nombre N de points à calculer dépendant généralement de la résolution d'affichage courante ou bien de la longueur d'arc estimée de la courbe. Par exemple, pour les courbes de Bézier, une méthode consiste à fixer ce nombre à la diagonale du carré de côté correspondant au plus grand segment du polygone de contrôle (cf [AP92] p152).

- On subdivise ensuite l'intervalle paramétrique $[0, 1]$ en N valeurs intermédiaires.

- On itère enfin un algorithme d'évaluation algébrique (Schoenberg) ou géométrique (de Boor) sur les valeurs du paramètre t obtenues.

Cependant, l'espacement sur la courbe des points calculés ne reflète généralement pas l'écart entre les valeurs de t correspondantes. Dans la mesure où nous souhaitons obtenir des points à distance (à peu près) fixe pour déterminer un échantillonage représentatif de la courbe, cela nous pose un problème.

Une première solution consiste à *sur-échantillonner* la courbe, c'est-à-dire à adopter un pas de calcul suffisamment petit pour être certain que dans tous les cas la distance entre deux points calculés n'excédera pas une certaine valeur (taille d'un pixel, ou une longueur de segment maximale). L'inconvénient évident de cette méthode est qu'à supposer qu'on puisse trouver une valeur optimale du pas (suffisamment petite, mais pas trop) pour satisfaire ce critère, on va forcément calculer un grand nombre de points redondants ou trop proches dans le cas général.

Pour éviter ce gaspillage de ressources, d'autant plus gênant que les méthodes d'évaluation directes de points sont assez coûteuses en nombre d'opérations, nous allons examiner des méthodes permettant d'adapter les valeurs de t au profil de la courbe.

3.4.1.2 Techniques de reparamétrisation

Leur but est de trouver une paramétrisation basée sur la longueur d'arc de la courbe, pour laquelle une série de valeurs de t uniformément espacées correspondraient à une série de points de la courbe dont les abcisses curvilignes seraient séparées d'une distance fixe.

Pour ce faire, il faut trouver une fonction de reparamétrisation, associant aux anciennes valeurs du paramètre de nouvelles valeurs. La courbe reparamétrée décrit le même tracé que la courbe originale, mais pour les mêmes valeurs de t, on n'obtient plus les mêmes points sur la courbe.

Un certain nombre de documents traitent de ce problème pour les courbes NURBS (cf [Blo99], [CM96], [Alt93] entre autres). Il est a priori impossible de définir une reparamétrisation basée sur la longueur d'arc (cf [CM96]), mais certaines techniques permettent cependant d'obtenir des approximations intéressantes.

3.4.1.3 Une technique de tracé adaptative

Leslie Piegl décrit dans [Pie83] un algorithme déterminant pour chaque point la valeur du pas de calcul à adopter pour s'assurer que la distance entre deux points soit toujours bornée. Voici une présentation résumée ce cette méthode :

- On fixe un intervalle de tolérance $[ex1, ex2]$ bornant la distance D qui séparera deux points consécutifs. Remarquons que la borne inférieure ne devrait logiquement pas aller en deçà de la taille d'un pixel.
- On fixe aussi une estimation du pas de calcul dt initial à soumettre à l'algorithme.
- Soit le point $C(i)$ déjà calculé : on détermine la valeur du nouveau point $C(i + dt)$ par une méthode d'évaluation directe, de même que la distance D qui le sépare de son prédécesseur.
  - Si $D < ex1$, alors on augmente le pas de calcul courant de $\frac{dt}{2^n}$, et on le soumet de nouveau à l'algorithme (n comptant le nombre total de propositions soumises).
  - Si $D > ex2$, alors on diminue le pas de calcul courant de la même valeur.
- Quand le pas de calcul est finalement accepté, on le soumet de nouveau à l'algorithme jusqu'à ce que la valeur de t atteigne 1.

D'après les tests réalisés par l'auteur, il faut en moyenne $1.4N$ itérations de l'algorithme pour tracer une courbe de N points. Le nombre de corrections de dt se maintient donc autour de 28%. Ce résultat semble très intéressant a priori, car le nombre d'itérations "inutiles" est probablement très inférieur à celui nécessaire au tracé d'une courbe sur-échantillonnée.

Notons que le coût des opérations requises par l'algorithme est très faible par rapport au coût de l'évaluation d'un point de la courbe : il suffit de quelques additions et soustractions, d'au plus deux comparaisons et d'une division par 2^n en cas de correction, cette dernière opération s'effectuant efficacement par un décalage de n bits.

Cependant, la validité du résultat de cet algorithme dépendra en grande partie de l'opportunité du choix de l'intervalle de tolérance. Celui-ci doit en effet être suffisamment étroit pour qu'il n'y ait pas de risque d'occulter une zone de forte discontinuité de la courbe. Il est donc envisageable de le fixer lorsqu'on a une certaine connaissance préalable de la forme générale de la courbe.

3.4.2 Méthodes incrémentales : usage du *Forward Differencing*

La technique de *Forward Differencing* permet de calculer les valeurs successives d'un polynôme $P(t)$ de degré p par incrément de sa valeur précédente, ce en n'effectuant que p additions par pas de calcul. Nous allons donc pouvoir déterminer plus efficacement que les méthodes directes classiques une série de points de la courbe.

Principe :

Soit $P(t)$ le polynôme de degré p que l'on souhaite évaluer pour les i valeurs $\{0, k, 2k, \ldots, (i-1)k\}$ espacées du pas k :

$$P(t) = \sum_{i=0}^{p} \alpha_i t^i \tag{3.2}$$

D'après [Wal], [Jan01], les p *forward differences* du polynôme sont alors définies de la manière suivante :

$$\begin{array}{ll} \Delta^1 P = P(t+k) - P(t) & \text{first forward difference} \\ \Delta^2 P = \Delta^1 P(t+k) - \Delta^1 P(t) & \text{second forward difference} \\ \Delta^3 P = \Delta^2 P(t+k) - \Delta^2 P(t) & \text{third forward difference} \\ \vdots & \\ \Delta^p P = c & \text{the constant } p\text{-th forward difference} \end{array} \tag{3.3}$$

Soit k fixé, on peut ensuite déterminer les valeurs de :

$$\{P(0); \Delta^1 P(0); \Delta^2 P(0); \Delta^3 P(0)\}.$$

Le calcul des valeurs successives de P se déroule alors par itération de l'algorithme suivant sur les valeurs de i :

$$\begin{array}{l} P(i+k) = P(i) + \Delta^1 P(i) \\ \Delta^1 P(i+k) = \Delta^1 P(i) + \Delta^2 P(i) \\ \Delta^2 P(i+k) = \Delta^2 P(i) + \Delta^3 P(i) \\ \Delta^3 P(i+k) = \Delta^3 P(t+k) - \Delta^4 P(t) \\ \vdots \\ \Delta^p P(i+k) = \Delta^{p-1} P(i) + c \end{array} \tag{3.4}$$

Il faut donc p additions dans le cas général[2] pour calculer les valeurs successives de P, ce qui est dérisoire en comparaison du nombre d'opérations requis par une évaluation classique. En effet, il faut p additions et multiplications pour évaluer une valeur du polynôme par la méthode de Horner, et $\frac{1}{2}q^2 + \frac{3}{2}q$ multiplications et p additions de manière directe (avec $q = p - 1$). Le coût des opérations requises par le calcul initial des *forward differences* n'a qu'une influence négligeable par rapport au coût des opérations économisées.

Un exemple d'application de cette technique sur un polynôme de degré 3 est traité à l'annexe 6.4.

3.4.2.1 *Forward Differencing* sur les courbes de Bézier

Remarque préalable : Dans la mesure où l'étude des courbes de Bézier ne nous concerne pas directement, nous nous référerons au chapitre 1.3 de [PT95] pour la justification des propriétés utilisées dans cette section.

[2] les p dernières itérations requièrent graduellement moins d'additions, cf figure 6.2 de l'annexe 6.4

Pour évaluer une série de points sur une courbe de Bézier en utilisant cette technique économe, il faut d'abord déterminer une représentation polynômiale de ces courbes, puis calculer les forward differences correspondant au point de la courbe pour la valeur 0 du paramètre t.

- La définition polynômiale des courbes de Bézier de degré 2 est :

$$C(t) = (1-t)^2\, P_0 + 2\,t(1-t)\, P_1 + t^2\, P_2$$

  Celle des courbes de degré 3 est :

$$C(t) = (1-t)^3\, P_0 + 3\,t(1-t)^2\, P_1 + 3\,t^2\,(1-t)\, P_2 + t^3\, P_3$$

- Lorsque $t = 0$, les *forward differences* des courbes de Bézier de degré 2 sont :

$$\begin{aligned}
\Delta^1 P(0) &= (k^2 - 2k) \cdot P_0 + (-2k^2 + 2k) \cdot P_1 + (k^2) \cdot P_2 \\
\Delta^2 P(0) &= 2k^2 \cdot P_0 - 4k^2 \cdot P_1 + 2k^2 \cdot P_2
\end{aligned}$$

  Leur calcul, développé dans [Jan01], tient en quelques lignes et reste donc assez facile à réaliser à la main. Ce n'est malheureusement plus le cas des *forward differences* des courbes de degré 3 : dans la mesure où nous ne les avons pas trouvées dans les articles dont nous disposions, nous avons dû les déterminer à l'aide d'un logiciel de calcul symbolique (*Maple V*). Nous obtenons :

$$\begin{aligned}
\Delta^1 P(0) &= ((1-k)^3 - 1) \cdot P_0 + 3k(1-k)^2 \cdot P_1 + 3k^2(1-k) \cdot P_2 + k^3 \cdot P_3 \\
\Delta^2 P(0) &= (6k^2 - 6k^3) \cdot P_0 + (-6k(1-k)^2 + 12k(1-2k)^2) \cdot P_1 \\
&\quad + (6k^2 - 18k^3) \cdot P_2 + 6k^3 \cdot P_3 \qquad (3.5) \\
\Delta^3 P(0) &= -6k^3 \cdot P_0 + (-36k(1-2k)^2 + 27k(1-3k)^2 + 9k(1-k)^2) \cdot P_1 \\
&\quad + -18k^3 \cdot P_2 + 6k^3 \cdot P_3
\end{aligned}$$

  Dans tous les cas, $C(0) = P_0$, conformément aux propriétés des courbes de Bézier.

Remarque : Contrairement aux autres algorithmes de tracé, le degré ne peut plus être qu'un simple paramètre. En effet, nous devons pour chacune des valeurs de p calculer les équations ci-dessus afin d'écrire le code de l'algorithme de tracé en conséquence. Cela signifie que nous ne pourrons traiter que les courbes de Bézier dont le degré aura été prévu à l'avance. En pratique, cela n'est pas très gênant pour l'instant dans la mesure où le degré $p = 3$ devrait probablement nous suffire.

Si cette méthode a le mérite de proposer une évaluation extrêmement peu coûteuse des courbes de Bézier, son principal inconvénient vient du fait que l'on perd la possibilité d'agir sur les valeurs de t. En effet, comme l'explique la section 3.4.1.1, l'espacement des points calculés ne reflète pas en général l'écart entre les valeurs de t correspondantes, ce qui nous oblige à un sur-échantillonnage de la courbe pour maintenir la distance maximale entre deux points sous un seuil acceptable.

Pour remédier à ce problème, une variante adaptative de la méthode de *Forward Differencing* a été mise au point (cf [LSP87]). Elle permet de modifier le pas de calcul par transformation linéaire des coefficients de la courbe afin de préserver un espacement régulier entre les points calculés (dans le cadre de cet article, l'espacement est fixé à la taille d'un pixel). En contrepartie, cette adaptation requiert un surplus d'opérations, dont le coût devrait tout de même être bien inférieur à celui du tracé de courbe par une méthode directe.

3.4.2.2 *Forward Differencing* sur les courbes NURBS

La représentation polynômiale d'une courbe de Bézier ne dépend que du degré de la courbe (cf section 3.4.2.1). C'est aussi le cas des courbes B-Splines uniformes, définies par une combinaison linéaire de deux polynômes différents (cf figure 3.2).
Le cas des courbes B-Splines extrémales "uniformes" est déjà plus complexe : les noeuds extrêmes de multiplicité $p + 1$ changent la forme des fonctions d'influence définies sur les segments extrémaux, ce qui nous oblige à définir des polynômes supplémentaires pour les représenter (cf figure 3.3). Toutes ces expressions polynômiales sont détaillées dans [Bla94] p29.

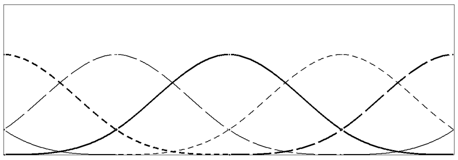

FIG. 3.2 – Fonctions d'influence d'une courbe B-Spline uniforme.

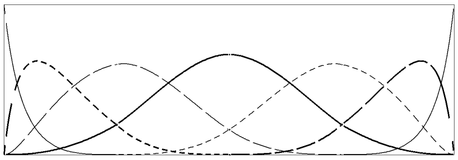

FIG. 3.3 – Fonctions d'influence d'une courbe B-Spline extrémale.

Mais lorsque l'on aborde le cas des NURBS, dont même les noeuds internes ne sont plus d'espacement uniforme, il n'existe plus de représentation polynômiale générale comme c'est le cas des modèles précédents. Nous devons alors calculer une représentation spécifique au vecteur nodal de la courbe. A cette fin, une technique est décrite dans [SC88] pp191-192. Cet article traite aussi d'une méthode de tracé adaptative identique à [LSP87].

En conclusion, la méthode de *Forward Differencing* semble prometteuse en termes de coût de calcul. Cependant, il sera certainement nécessaire d'employer les méthodes adaptatives afin de préserver l'homogénéité du tracé. De plus, le calcul de représentations polynômiales doit être effectué pour chaque intervalle de noeuds en ce qui concerne les courbes NURBS. Il faut alors vérifier si le surcoût de ces opérations n'altère pas les performances de cette méthode. Aussi, [SC88] (entre autres) rappelle que cette technique est a priori assez sensible aux imprécisions de calcul.

3.4.3 Méthodes de manipulation du réseau de contrôle

Ces méthodes cherchent à fournir une approximation polygonale de la courbe en raffinant son réseau de contrôle jusqu'à ce qu'il devienne une approximation acceptable de la courbe. Les points trouvés n'appartiennent pas forcément à la courbe.

3.4.3.1 Insertion de noeuds et de points de contrôle

(Cette section est inspirée de [Bla94] et [KM83]).

Soit une courbe B-Spline C de degré p définie par un vecteur nodal U_m et un réseau de contrôle $\{P_0, \dots, P_n\}$. Supposons que l'on veuille affiner le vecteur nodal en y ajoutant un nouveau noeud de valeur u_i, ce qui implique l'ajout du point de contrôle correspondant. L'algorithme d'insertion de noeud consiste à déterminer le vecteur nodal U_{m+1} et le réseau de contrôle $\{P_0, \dots, P_{n+1}\}$ qui définissent la même courbe C après cette insertion.

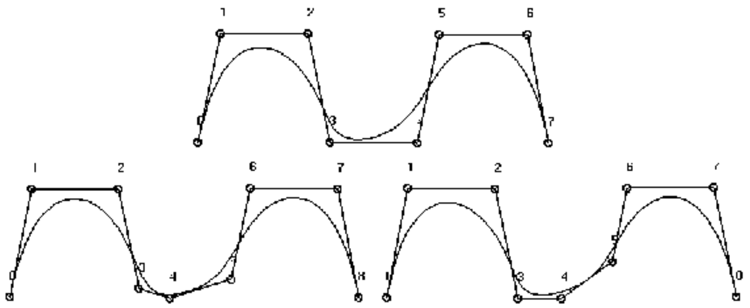

FIG. 3.4 – Insertion d'un noeud et insertion d'un point de contrôle sur une courbe B-Spline extrémale

Wolfgang Boehm a mis au point un algorithme d'insertion de noeuds adapté aux courbes B-Splines ([Boe80]) : la figure 3.4 nous en donne un exemple. L'intérêt premier de cet algorithme est de permettre des modifications locales de la forme de la courbe : en effet, un point de contrôle supplémentaire confère un degré de liberté supplémentaire pour modifier la courbe tout en réduisant la portion de courbe qui sera affectée (propriété de localité L^{p+1}).

On peut remarquer que dans le cadre de la modélisation géométrique, manipuler le vecteur nodal est moins naturel que manipuler le réseau de contrôle. C'est pourquoi Les Piegl a mis au point une variante de cet algorithme qui permet d'effectuer l'insertion à partir de la position d'un point de contrôle (cf [Pie89] p511). Les adaptations de ces algorithmes aux courbes NURBS sont décrites dans [Bla94] (pp58–59).

Ce qui nous intéresse dans le cadre du tracé de courbes est le constat suivant : la figure 3.4 nous montre que les réseaux de contrôle produits par ces algorithmes sont de meilleures approximations de la courbe au voisinage du lieu de l'insertion que le réseau de contrôle original.
Partant de ce constat, de nombreux articles traitent de méthodes de tracé de courbes par *subdivision* du vecteur nodal ou du réseau de contrôle. La plupart des méthodes envisagées insèrent récursivement soit un noeud au milieu de chaque intervalle[3] du vecteur nodal (dont [Cat74] et [LCWB80]), soit un point au milieu de chaque segment du réseau de contrôle (dont [LR80]). La subdivision s'arrête lorsque le réseau de contrôle obtenu est une approximation polygonale de la courbe jugée suffisamment fidèle :

- Dans [Cat74], on stoppe la subdivision lorsque la taille des segments atteint la taille d'un pixel du repère discret.

- Dans [LCWB80] et [LR80], on stoppe la subdivision lorsque les morceaux de courbe entre deux points du réseau de contrôle obtenu sont jugés suffisamment linéaires. Des procédures de test de linéarité, basées sur la mesure de la déviation entre une section de courbe et le segment du réseau de contrôle, sont décrites dans [FP79] et simplifiées dans [KM83].

Les auteurs de [KM83] remarquent que la méthode de subdivision du réseau de contrôle développée par [LR80] se base sur la propriété d'enveloppe convexe, assurant qu'à chaque étape de la subdivision, la courbe reste dans la nouvelle enveloppe convexe formée par les points de contrôle. Sachant que les courbes rationnelles vérifient cette propriété dans le repère homogène ($\mathbb{R}^4$) mais pas dans le repère Euclidien associé ($\mathbb{R}^3$), ils proposent alors une méthode de subdivision valide pour les courbes rationnelles

[3]non-extrémal

(dont les NURBS).

3.4.3.2 Elevation du degré de la courbe

Soit une courbe C de degré p définie par un vecteur nodal U_m et un réseau de contrôle $\{P_0, \ldots, P_n\}$. Supposons que l'on veuille trouver une courbe de degré $p+1$ dont le tracé serait identique à celui de C. L'algorithme d'élévation de degré consiste à déterminer le vecteur nodal $U_{m'}$ et le réseau de contrôle associé qui définissent cette courbe.

Pour les courbes de Bézier, l'algorithme d'élévation de degré se résume à l'application de la première itération de l'algorithme de Casteljau pour $t = \frac{p}{p+1}$: il s'agit donc d'une algorithme géométrique simple, et qui n'implique pas dans le calcul le vecteur nodal. Le nouveau réseau de $p+1$ points de contrôle, seule donnée que nous souhaitons extraire en vue du tracé de la courbe, est formé à partir des nouveaux points calculés (par exemple, $\{P_{0,0}; P_{1,0}; P_{1,1}; P_{1,2}; P_{0,3}\}$ sur la figure 3.1). Se référer à [BFJ84] pour plus de détails.

Pour les courbes B-Splines , la méthode générale consiste à incrémenter la multiplicité des noeuds du vecteur nodal comme illustré ci-dessous :

$$U_m = \{\underbrace{a, \ldots, a}_{p+1}, \underbrace{u_1, \ldots, u_1}_{\mu_{u_1}}, \ldots, \underbrace{u_i, \ldots, u_i}_{\mu_{u_i}}, \underbrace{b, \ldots, b}_{p+1}\} \Rightarrow U'_m = \{\underbrace{a, \ldots, a}_{p+2}, \underbrace{u_1, \ldots, u_1}_{\mu'_{u_1}=\mu_{u_1}+1}, \ldots, \underbrace{u_i, \ldots, u_i}_{\mu'_{u_i}=\mu_{u_i}+1}, \underbrace{b, \ldots, b}_{p+2}\}$$

Cela consiste donc à insérer des noeuds supplémentaires en utilisant par exemple l'algorithme de Boehm. L'algorithme fonctionne de la même manière pour les courbes NURBS (cf [PT95] pp188–194). Plus de détails peuvent notamment être trouvés dans les articles suivants : [Pra84], [CLS85], [PP91].

Remarques : Constatant la simplicité et la relative efficacité de l'algorithme d'élévation de degré des courbes de Bézier, Leslie Piegl et Wayne Tiller ont proposé dans [PT94] une méthode d'élévation de degré consistant à découper une courbe B-Spline en une série de courbes de Bézier (cf section 3.4.4), puis à élever le degré de chacune d'entre elles. D'après les tests réalisés, les performances de cette méthode la classent parmi les plus efficaces qui soient dans les cas étudiés.

En conclusion, Cohen et Schumacker ont montré dans [CS85] que l'élévation du degré fait converger le réseau de contrôle vers la courbe B-Spline de manière linéaire, alors que l'insertion de noeuds converge de manière quadratique. On pourrait alors supposer qu'il serait préférable d'adopter la seconde méthode pour trouver en moins d'itérations une approximation satisfaisante de la courbe, mais il faudrait d'abord mener une étude comparative de la consommation de ressources de ces méthodes et de leurs nombreuses variantes respectives.

3.4.4 Une méthode de transformation de NURBS en courbes de Bézier

Leslie Piegl et Wayne Tiller (entre autres) remarquent qu'une courbe B-Spline ou NURBS dont le vecteur nodal aurait la forme de U_{bzier} (cf équation 3.6) aurait pour fonctions d'influence celle des courbes de Bézier (cf [PT95] p52).

$$\{\underbrace{a, \ldots, a}_{p+1}, \ldots, \underbrace{b, \ldots, b}_{p+1}\} \tag{3.6}$$

La capacité du modèle NURBS à exprimer entre autre les courbes B-Splines et les courbes de Bézier est l'une des principaux motifs de son choix comme standard en modélisation géométrique. Cependant, de nombreuses plate-formes graphiques ne supportent effectivement que le tracé de courbes de Bézier. Ce modèle est en effet connu depuis plus longtemps que les NURBS, et ses algorithmes de tracé sont plus simples à implémenter et aussi moins coûteux, même si en contrepartie le modèle ne possède pas toutes les propriétés géométriques intéressantes (e.g localité) des NURBS.

C'est pourquoi certains algorithmes, basés sur notre constat initial, ont été mis au point pour transformer des courbes NURBS en suites de courbes de Bézier. Leur motivation est autant de pouvoir supporter le modèle des NURBS dans un système graphique sans ajouter de nouveaux algorithmes de tracé, que de considérer une méthode de tracé de NURBS probablement plus efficace qu'une itération de méthode de calcul d'un point de la courbe.

Pour parvenir à leur fin, ces algorithmes augmentent la multiplicité de chaque noeud non-extrémal jusqu'à la valeur p par insertion de noeuds dans le vecteur nodal, comme illustré ci-dessous. Nous pouvons remarquer qu'il y a $n+p+1$ intervalles séparant deux séries de p ou $p+1$ noeuds de valeurs distinctes. De par la propriété de localité L^{p+1} des fonctions d'influence, chaque série de $p+1$ fonctions d'influence valides sur un de ces intervalles n'est non-nulle que sur cet intervalle. De plus, la continuité de la jonction entre deux segments de courbe est alors $\mathcal{C}^0$, ce qui indique tout d'abord une connexité des segments sans continuité des vecteurs vitesse ou accélération, mais aussi que le réseau de contrôle obtenu par insertion interpole la courbe. Le constat initial nous rappelant que sur chaque intervalle, les fonctions d'influence sont identiques à celle d'une courbe de Bézier, nous pouvons constater, d'après les propriétés des courbes de Bézier, que nous avons transformé notre courbe B-Spline ou NURBS en une série de $n+p+1$ courbes de Bézier.

$$U_m = \{\underbrace{a, \dots, a}_{p+1}, u_{p+1}, \dots, u_n, \underbrace{b, \dots, b}_{p+1}\} \Rightarrow U'_m = \{\underbrace{a, \dots, a}_{p+1}, \underbrace{u_{p+1}, \dots, u_{p+1}}_{\mu_{u_{p+1}}=p}, \dots, \underbrace{u_n, \dots, u_n}_{\mu_{u_n}=p}, \underbrace{b, \dots, b}_{p+1}\}$$

L'algorithme le plus simple pour effectuer cette transformation est celui de raffinement de vecteur nodal : C'est une variante des algorithmes d'insertion d'un noeud qui permet plutôt d'insérer directement toute une série de noeuds, ce qui semble adapté à nos besoins dans notre cas. Nous pouvons remarquer que son utilité ne se limite pas à la conversion en courbes de Bézier (e.g, convergence du réseau de contrôle vers la courbe).

De nombreuses variantes plus ou moins performantes de ces deux familles d'algorithmes sont évoquées dans la littérature, mais nous nous contenterons plutôt de constater que Leslie Piegl propose un algorithme spécifiquement adapté à la transformation de NURBS en série de courbes de Bézier (cf [PT95] p173, [AP92] p154). En effet, il parcourt le vecteur nodal de gauche à droite et procède à l'insertion de noeuds en tenant compte de la multiplicité du noeud courant, tout en mémorisant temporairement certains coefficients qui interviennent à plusieurs reprises dans le calcul (cf section 4.3.4 pour code).

3.5 Conclusion

Le contexte d'une session X en mode client-serveur nous oblige à déterminer une approximation polygonale de la courbe afin de ne pas saturer le goulot d'étranglement que représente le réseau. En effet, la représentation discrète de notre courbe devra y circuler dans le cas général.

Nous avons évoqué différentes familles de méthodes déterminant des approximations polygonales de courbes. Certaines d'entre elles, dont les méthodes de tracé incrémentales, peuvent être pressenties comme plus efficaces que les autres. Cependant, la nécessité d'obtenir une approximation fidèle induit des méthodes d'adaptation du tracé au profil de la courbe, dont le coût encore indéterminé empêche toute conclusion hâtive.

C'est pourquoi nous allons entreprendre une étude détaillée des ressources consommées par une sélection d'algorithmes de tracé afin de nous permettre de les départager objectivement.

Chapitre 4

Optimisation des méthodes de tracé par spécialisation

Nous avons fait un tour d'horizon des principales méthodes de tracé de NURBS susceptibles de produire des approximations polygonales. Nous avons aussi montré que les algorithmes de tracé de NURBS étaient dans certains cas susceptibles de s'exécuter en temps réel : nous pouvons donc supposer que leur vitesse d'exécution va être le principal critère de sélection dans la suite de notre étude.
Nous allons donc pouvoir mener une étude comparative détaillée du coût des algorithmes de tracé. Mais du fait du temps imparti à un mémoire de DEA, nous ne pourrons traiter que certains d'entre eux, parfois choisis de manière arbitraire.

Cependant, puisque la vitesse d'exécution des algorithmes est définie comme facteur critique, il semble obligatoire d'optimiser le code de ces algorithmes de tracé. Ainsi, nous concentrerons notre attention sur le coût des algorithmes après leur optimisation. Mais pour pouvoir les comparer de manière objective, il faut d'abord établir des règles communes d'optimisation et d'évaluation des performances.

4.1 Optimisation par spécialisation de programmes

Cette section s'inspire du cours de Spécialisation de Programmes et Langages Adaptatifs de Charles Consel (DEA LaBRI, 2000/2001).

4.1.1 Modalités générales de l'optimisation

Lorsque un programme s'avère consommateur d'une trop grande quantité de *ressources machine* (temps machine ou mémoire), il convient alors de l'*optimiser*, c'est à dire de modifier son code source afin de réduire la consommation de ressources machine du programme final.

Conformément à la *règle du goulot d'étranglement* (bottleneck rule), les optimisations apportées à un programme ne devraient généralement avoir qu'un caractère local. En effet, cette règle empirique stipule que lorsque l'on examine l'exécution d'un programme, on constate assez souvent que seul 20% de son code est à l'origine de 80% de sa consommation en ressources machine. On dit aussi que ce code fait partie de la *section critique* du programme.

Cet examen se fait par l'intermédiaire d'outils de *profilage de code* (code profiling). Nous pouvons remarquer que les principaux systèmes Common Lisp, dont CMU Common Lisp, disposent en standard de tels outils, notamment capables de définir la consommation en ressources machine de chaque fonction exécutée.

Il convient donc d'optimiser le code d'un programme avec discernement. Pour chaque programme, on peut en effet prouver par profilage que si la totalité de son code était optimisée, la majorité des optimisations n'auraient aucune incidence notable sur la consommation en ressources du programme obtenu. Mais on peut surtout constater que les optimisations tendent à altérer la qualité du code, son efficacité primant alors sur sa lisibilité et donc sur sa maintenabilité.
Dans la mesure où la plupart des logiciels sont amenés à évoluer bien après leur date de "mise en service", cette dernière remarque est d'importance.

On peut aussi noter que la notion de *performance*, i.e. de faible consommation de ressources machine, est très subjective. En effet, il s'avère qu'un programme est souvent jugé suffisamment performant dans la mesure où sa vitesse d'exécution constatée[1] par ses utilisateurs convient à la majorité d'entre eux. Or, cette vitesse dépendra autant des attentes et besoins de ces utilisateurs que de la quantité de ressources machine disponibles, cette dernière tendant à évoluer de manière exponentielle au cours du temps.

En conséquence, il est nécessaire d'optimiser les codes des programmes en tenant compte aussi bien d'une consommation de ressources modérée que du temps de développement requis pour une version initiale et du temps de développement induit par la qualité du code sur les évolutions futures.

Traditionnellement, les fonctions de tracé figurent parmi les fonctions les plus consommatrices de ressources : il sera donc nécessaire de porter nos efforts sur leur optimisation. Pour ce faire, nous opterons dans un premier temps pour une méthode d'optimisation semi-automatique, consistant à adapter dès la compilation le code des programmes à leur environnement spécifique d'exécution : il s'agit de la méthode de *spécialisation de programmes*.

4.1.2 Spécialisation de programmes

4.1.2.1 Introduction

Pour permettre d'étendre autant que possible le domaine des cas que peut traiter une fonction, on tend à la doter d'un certain nombre de paramètres. Ce souci de généricité est légitime : la réutilisabilité du code en dépend. Cependant, cette généricité possède un certain coût, dans la mesure où chaque paramètre fourni va devoir être interprété pour influer sur le déroulement de la fonction.

Considérons par exemple le programme C `mini_printf` ci-dessous : c'est une implémentation minimaliste de la fonction `printf`, dont le but est d'afficher à l'écran une suite de caractères et de chiffres selon une chaîne de formatage passée en paramètre.

```
void mini_printf(char *fmt, int *values)
{
  for (i = 0; fmt != '/0'; fmt++)
    { /* normal character: display it */
      if (*fmt != '%')
```

[1]En notant qu'un dépassement de capacité mémoire nécessite un recours excessif à la mémoire virtuelle qui ralentira considérablement le système

```
        {putchar(*fmt);}
      else
        { /* format character: execute it */
          switch(*++fmt)
            {
              /* '%d' encountered: display the current integer from
                 'values' array */
            case 'd':
              putint(values[i++]); break;
            case '%':
              putchar('%'); break;
            default:
              error(); break;
            }
        }
    }
}
```

Il est indispensable de fournir au programmeur une fonction d'affichage à vocation générique, mais cependant, si l'on se place dans un cas particulier, le programme effectue alors un certain nombre d'instructions inutiles. Supposons en effet que l'on utilise à plusieurs reprises la fonction `mini_printf` avec pour chaîne de formatage $F1 =$ "`n = %d ;`" : la fonction va devoir évaluer la chaîne à chaque exécution.
Pourtant, un programmeur soucieux d'économie pourrait écrire le code suivant pour traiter ce cas particulier :

```
void mini_printf_F1(int *values)
{
  putchar('n');
  putchar(' ');
  putchar('=');
  putchar(' ');
  putint(values[0]);
}
```

Il est important de noter que les deux programmes (tirés de [Com98]) sont strictement équivalents dans le cadre du traitement de la chaîne $F1$. Cependant, `mini_printf_F1` est visiblement bien plus économe en instructions : on dit que c'est une instance *spécialisée* de `mini_printf` pour le contexte `*fmt = F1`.

4.1.2.2 Définition

Le processus de *spécialisation de programmes* travaille à partir d'un code source générique placé dans un contexte d'exécution spécifique pour produire une instance de code source spécialisée pour ce contexte. Pour définir un contexte d'exécution, il suffit de définir certains paramètres comme étant invariants tout au long de l'exécution d'une séquence de code. Le processus de spécialisation va notamment consister à retirer du code original toutes les instructions liées à l'interprétation de ces invariants.

- Ces paramètres invariants sont appelés *paramètres statiques* : leur valeur est fixée dès la phase de compilation du programme. Ils recouvrent des paramètres des fonctions à spécialiser mais aussi des éventuelles variables globales du programme que les fonctions peuvent utiliser.
- Les autres paramètres sont appelés *paramètres dynamiques* : comme leur valeur n'est communiquée au programme que pendant sa phase d'exécution, il est impossible d'orienter la compilation du

programme en conséquence.

Le processus de spécialisation est donc un processus statique : puisqu'il vise à produire un code source à compiler avec le compilateur adéquat (*gcc* pour un code C, *Python* de CMU Common Lisp pour un code Common Lisp), il fait partie intégrante du processus global de compilation de programme représenté sur la figure 4.1.

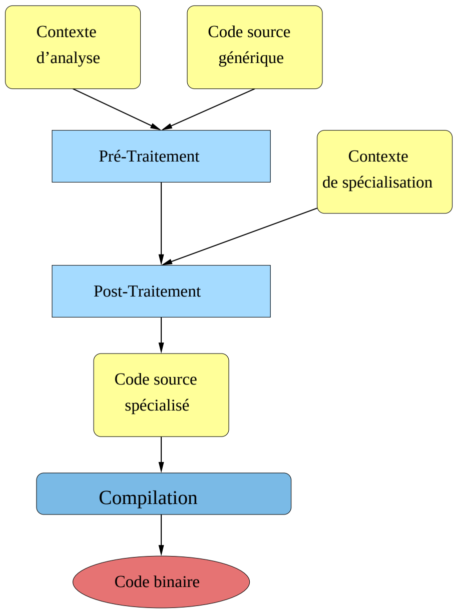

FIG. 4.1 – Etapes du processus de spécialisation de programmes

Prenons pour exemple la spécialisation de la fonction `mini_printf` décrite dans la section précédente. Le contexte d'analyse consiste en la déclaration de `fmt` comme paramètre statique et de `values` comme paramètre dynamique. Le contexte de spécialisation consiste en l'affectation de `fmt` à "`n = %d`".

Deux opérations ont été réalisées lors de la spécialisation de cette fonction :

- Un *déroulage de boucles* (loop unrolling) : la boucle `for`, itérant sur les valeurs de la chaîne de formatage statique, a donc pu être supprimée et remplacée par la succession des instructions exécutées à chaque itération pour la valeur courante du caractère de la chaîne (un `putchar` ou `putint`).

- Un *pliage de code* (code folding) : tous les calculs qui dépendent exclusivement de constantes ou de paramètres statiques sont réalisés dès la spécialisation, et seul leur résultat figure dans le code produit (arguments des `putchar`).

Le détail des différentes étapes de la figure 4.1 sera détaillé dans la section qui suit.

4.1.3 Deux méthodes de spécialisation

La bibliothèque FreeCLIM est une bibliothèque écrite en langage Common Lisp : on pourrait donc penser a priori que le code des fonctions de tracé soit obligatoirement du code Common Lisp. Or, dans le cadre du projet TER [JFB+00], nous avons remarqué que le système CMU Common Lisp propose une fonctionnalité *FFI* (Foreign Functions Interface) permettant d'intégrer dans le système du code binaire provenant d'un autre langage (cf [Sch97], chap. 8). Ces fonctionnalités sont courantes dans les systèmes Common Lisp, et leur usage l'est aussi pour importer des bibliothèques écrites en C (interfaces vers OpenGL) ou Fortran (MatLisp).

En reprenant les étapes de la figure 4.1, nous allons alors pouvoir comparer deux méthodes de spécialisation : l'usage du spécialiseur TEMPO pour du code C, et l'usage de macros pour du code Common Lisp.

4.1.3.1 Le spécialiseur TEMPO

- L'utilisateur fournit le contexte d'analyse sous forme de fichiers précisant la hiérarchie des fonctions à spécialiser, et pour chacune d'entre elles, les paramètres qui seront statiques ou dynamiques.
- Le pré-traitement va analyser les opportunités de spécialisation en propageant ces informations à travers le code afin de déterminer quelles sections vont pouvoir être pré-calculées. Il va ensuite produire automatiquement un programme générateur de code, qui prendra en paramètre les valeurs de spécialisation des paramètres statiques.
- Le post-traitement va exécuter ce programme pour des valeurs de spécialisation spécifiées par l'utilisateur dans un fichier généré lors du pré-traitement. Un code source spécialisé est alors produit.
- Enfin, le code final peut être compilé par gcc et exécuté.

Remarque : Dans la mesure où TEMPO évalue une partie du code source à partir de paramètres déclarés statiques, et passe par l'étape intermédiaire de pré-traitement, il fait partie de la famille des *évaluateurs partiels off-line*, par opposition aux “*on-line*” qui sautent cette étape en prenant directement des valeurs de spécialisation.

4.1.3.2 Les macros Common-Lisp

- Le contexte d'analyse et le code source original servent de spécification au programme générateur que l'utilisateur va devoir écrire.
- Le pré-traitement consiste en effet en l'écriture d'une macro Common Lisp qui prendra en paramètre des valeurs de spécialisation et sera capable de produire un code spécialisé en conséquence.
- Le post-traitement s'appelle la *macro-expansion* : elle a lieu pendant les premiers stades de la compilation du programme Common Lisp. En effet, les appels de macro figurant dans le code, avec pour paramètres des valeurs de spécialisation, sont remplacés par le code résultant de l'exécution de chaque macro pour les valeurs spécifiées.
  .
- Enfin, le code final peut être compilé par Python et exécuté.

4.1.3.3 Comparaison de ces méthodes

- **Le spécialiseur TEMPO :**

TEMPO présente l'avantage de spécialiser le code de manière non seulement automatique, mais aussi particulièrement efficace dans la mesure où il est capable d'anticiper sur les optimisations qu'effectuera gcc sur le code spécialisé.

Aussi, le langage C est très adapté à l'écriture de programmes particulièrement rapides, surtout dans le cadre de petites fonctions effectuant beaucoup de calculs mathématiques. En effet, le compilateur gcc est devenu extrêmement performant grâce aux efforts constants d'une très vaste communauté de développeurs motivés par le très grand nombre d'utilisateurs de ce compilateur.

Cependant, même si TEMPO a déjà fait ses preuves dans le cadre de projets industriels, il est encore au stade d'outil de recherche qui n'est pas encore complètement finalisé, et son utilisation reste encore peu commode pour un utilisateur lambda.

Aussi, cette option nous obligerait à intégrer du code C dans le code Common Lisp de la bibliothèque FreeCLIM : bien que nous n'ayons pas encore d'expérience en ce domaine, nous savons que cela posera des problèmes de portabilité.
En effet, l'intégration de fonction étrangères n'est pas normalisée ANSI Common Lisp, et dépend entièrement de l'implémentation du système Common Lisp utilisé (en notant que la plupart d'entre eux proposent cette option). En conséquence, il est impossible d'écrire notre bibliothèque avec du code ANSI Common Lisp par cette méthode. Mais surtout, le code C étranger devra être compilé sur chaque couple système d'exploitation / processeur, ce qui complique encore le problème.

- **Les macros Common Lisp :**

  Leur principal inconvénient est qu'il faut écrire une macro pour chaque fonction à spécialiser, ce qui allonge le temps de développement.

  Cependant :

  - Les fonctions de tracé de courbes ne sont pas très longues, en conséquence on peut raisonnablement coder des macros pour les spécialiser.

  - L'écriture de macros Common Lisp est aisée, dans la mesure où toutes les constructions du langage, très riche en la matière, sont disponibles. En effet, on peut s'efforcer d'écrire à ce stade un code très lisible et maintenable, dans la mesure où son exécution aura lieu pendant la compilation. Seule l'efficacité du code généré compte, et sa lisibilité n'a aucune importance puisqu'il ne fait pas partie du code source.
  - Nous sommes immédiatement prêts à utiliser cette méthode en se basant sur du code déjà écrit. En effet, lors du projet TER [JFB+00], nous avons écrit des macros spécialisant, entre autres, un module de calcul matriciel qui prend en compte le format[2] des matrices mises en jeu. Le résultat obtenu était satisfaisant, dans la mesure où une optimisation manuelle (très longue et fastidieuse) de notre part n'aurait pas donné de meilleur résultat.

  - La réputation selon laquelle les programmes Common Lisp sont lents à exécuter n'est pas fondée. Cela peut être le cas de certains programmes utilisant intensivement les fonctionnalités objet de CLOS ; cependant ces programmes sont en général à peine plus lents que leurs homologues C++ ou Java, et d'excellentes performances globales peuvent être obtenues pour peu que les program-

[2] Les matrices de transformation OpenGL comprennent une majorité de termes constants, formant des "motifs" guidant la spécialisation

meurs procèdent à des optimisations des sections critiques.

D'ailleurs, un certain nombre d'applications dites exigeantes en performances (cf [JFB+00] chap. 2), dont certains jeux vidéo 3D, sont programmées en Common Lisp pour tirer parti de la puissance et de la flexibilité du langage. Un bon niveau de performances est généralement maintenu en utilisant dans les sections critiques du code étranger (C ou assembleur), intégré par un protocole de type FFI.

Pourtant, tous les systèmes Common Lisp modernes disposent de compilateurs dont la plupart, surtout parmi les systèmes commerciaux, rivalisent de vitesse[3] avec gcc. Notons que le compilateur Python de CMU Common Lisp, système gratuit, figure parmi les plus performants disponibles. De plus, des tests récents publiés sur `comp.lang.lisp` montrent que c'est indiscutablement le plus performant des compilateurs gratuits.
Il faut cependant garder à l'esprit que sur la plupart des systèmes Common Lisp, de bonnes performances (temps machine et mémoire) ne sont atteintes que lorsque les programmes sont compilés, et ce avec des directives de compilation adéquates. Ces directives sont heureusement normalisées ANSI Common Lisp, ce qui permet à un même code possédant ces directives d'obtenir de bonnes performances sur tout système Common Lisp qui en tient effectivement compte (dont CMU Common Lisp).

De ce fait, nous pouvons obtenir un haut niveau de performances avec un code ANSI Common Lisp sans avoir recours à un code étranger.

Dans un premier temps, nous retiendrons l'option des macros Common Lisp. En effet, nous savons l'appliquer immédiatement, mais surtout le code produit et optimisé par la méthode de spécialisation sera intégralement ANSI Common Lisp.

Néanmoins, nous pouvons noter que les obstacles posés par la méthode TEMPO ne sont pas insurmontables. En effet, il est possible de créer des scripts automatisant la procédure d'installation de la bibliothèque pour les systèmes Common Lisp les plus utilisés, ces scripts se chargeant de compiler le code étranger pour le système d'exploitation courant et de l'inclure dans le système Common Lisp employé en utilisant ses directives spécifiques.

En conséquence, il serait profitable de tester et de comparer les deux méthodes à une étape ultérieure du projet afin de les départager plus nettement.

[3] d'exécution des programmes compilés

Nous avons donc décidé d'optimiser nos algorithmes de tracé en adoptant pour unique méthode d'optimisation la spécialisation de programmes. Nous mesurerons donc l'efficacité des programmes spécialisés sur la quantité d'opérations que nécessite leur exécution. Cela représente un moyen équitable de comparer leur efficacité, indépendamment du langage de programmation et du compilateur utilisé.

Cependant, pour évaluer le coût réel d'un algorithme en temps machine, il faut distinguer les opérations effectuées dans la mesure où chacune possède un coût propre.

Aussi, puisque nous travaillons dans le cadre de la spécialisation de programmes, il faut nécessairement définir au préalable un contexte d'analyse, déterminant les paramètres dynamiques et les paramètres statiques. Ensuite, puisque toute instance de code spécialisé est déterminée à partir de valeurs de spécialisation, nous allons devoir fixer arbitrairement de telles valeurs afin d'obtenir du code spécialisé qui servira de base à notre étude de performances.

Enfin, il faut noter que même si nous ne souhaitons pas exclure de nos projets les optimisations de compilation, qui jouent un rôle important en Lisp comme en C, nous devrons les tenir à l'écart de notre étude comparative dans la mesure où elles sont trop dépendantes de paramètres d'implémentation.

4.2 Hypothèses et notations

4.2.1 Code des programmes

4.2.1.1 Choix d'un langage de programmation

Lorsque le projet en sera au stade de l'implémentation du module NURBS ou au stade des tests effectifs d'algorithmes de tracé, nous emploierons le langage ANSI Common Lisp, conformément aux hypothèses que nous avons posé. Cependant, nous avons remarqué qu'une écrasante majorité des articles et ouvrages de référence fournissent directement leurs algorithmes sous forme de code source en langage C, peut-être par souci de diffuser des programmes directement utilisables dans un langage “efficace”.

Par simple souci de respecter cette convention de fait, nous emploierons le langage C pour décrire nos algorithmes sous forme originale et spécialisée.

4.2.1.2 Notations et coûts des opérations

Le coût de ces opérations dépend fortement du type d'objets auquel elles s'appliquent. C'est pourquoi les opérateurs qui interviennent dans le code de nos fonctions sont distingués par les notations suivantes :

- Les opérateurs $+, -, *, /$ sont définis pour les nombres entiers.
- Les opérateurs $\oplus$, $\ominus$, $\otimes$, $\oslash$ sont définis pour les nombres flottants.
- Les opérateurs $\bigoplus$, $\bigotimes$, $\bigodot$, $//$, $\equiv$ sont définis sur les vecteurs, les deux derniers figurant respectivement la division et l'affectation.

En se basant sur une estimation du nombre de cycles de calcul nécessaires à l'exécution des instructions sur un processeur de type Pentium, nous pouvons décider d'affecter à chacune d'entre elles des valeurs relatives de coût, exprimées en *unités arbitraires d'exécution* (UAE) (cf tableau 4.2.1.2) :

Remarques relatives aux coûts :

types ▶ opérations ▼	entiers	flottants
comparaisons	1	6
$+, -$	1	6
$*$	6	6
$/$	12	12
$=$	1	1

FIG. 4.2 – Coûts relatifs des différentes instructions en fonction du type numérique

- Son coût ne variant pas selon le type mis en jeu, l'opérateur $=$ est défini pour les entiers comme pour les flottants.

- Toute opération booléenne (`or, not, and`...) requiert une UAE.

- Les tests `if, while` requièrent 3 UAE : une pour le test booléen et 2 pour le saut. Il faut bien sûr y ajouter celles nécessaires à l'évaluation de l'expression booléenne testée.

- L'instruction `switch(int)` requiert 9 UAE (2 comparaisons et deux sauts préalables, plus une addition et un saut pour se brancher sur la bonne valeur) Cette estimation est valide dans le cas où les valeurs des clauses `case` sont relativement consécutives, ce qui est le cas dans nos algorithmes.

- Les tableaux de taille modérée sont généralement stockés dans le cache : ce devrait être le cas de tous les tableaux employés dans nos algorithmes. De ce fait, accéder à une valeur d'un tableau se fait très efficacement sans faire intervenir la mémoire centrale.

  Il faut une addition (1 UAE) pour accéder à une valeur du tableau à partir d'un indice variable, et aucune à partir d'un indice constant.

- Dans nos algorithmes, l'usage des vecteurs se limitera aux types `FPoint3` et `FPoint4`, i.e. à des points représentés par respectivement 3 et 4 coordonnées flottantes. Le coût d'une opération vectorielle équivaut bien entendu à n fois le coût des valeurs numériques contenues (ici, les flottants), n étant la dimension du vecteur.

Ces valeurs de coût sont purement indicatives, et dépendent complètement du type de processeur utilisé (d'où la nécessité d'unités relatives). Cependant, elles sont probablement assez représentatives de la réalité pour la plupart des processeurs. Dans cette mesure, elles devraient permettre une évaluation comparative suffisamment objective des différents algorithmes de tracé.

4.2.2 Spécialisation des programmes

Notation de spécialisation : L'opérateur $\widehat{expr}$ remplace dans le code l'expression *expr* par le calcul de sa valeur . Ce calcul ayant lieu pendant la compilation, l'expression ne devra contenir que des variables statiques ou des constantes.

4.2.2.1 Paramètres statiques et dynamiques

- Le degré p d'une courbe, son réseau de contrôle déterminé par P et n, et son vecteur nodal U_m sont fixés définitivement lors de la phase de modélisation de la courbe. Quand vient le moment de tracer la courbe, ces valeurs sont invariantes : ce sont donc des paramètres statiques.
  En conséquence, nous pourrons spécialiser les fonctions de tracé selon des valeurs de ces paramètres.

- Cependant, les valeurs de t ne sont a priori pas connues à l'avance. De ce fait, nous devrons laisser t comme paramètre dynamique.

4.2.2.2 Valeurs arbitraires de spécialisation

Si l'étude que nous menons nous permet d'élire certains algorithmes de tracé, nous allons écrire des programmes générateurs (sous forme de macros Common Lisp) capables de calculer le code d'instances de ces algorithmes et prenant en argument des valeurs de paramètres de spécialisation.

En conséquence, du code spécialisé pourra être généré pour n'importe quelles valeurs de ces paramètres. Paradoxalement, pour réaliser notre étude de performances, nous avons besoin d'examiner des instances de code spécialisé. C'est pourquoi il nous faut fixer des valeurs de spécialisation, sans lesquelles nous ne pouvons pas obtenir de code à étudier.

Les valeurs que nous allons choisir sont avant tout arbitraires et indicatives, même si nous nous sommes efforcés de choisir des valeurs proches des exemples étudiés jusqu'à présent. Il s'agit surtout d'établir un protocole commun permettant d'entreprendre l'étude comparative. Cependant, il faudra que l'analyse de performances qui en découle soit générale, c'est-à-dire indépendante des valeurs arbitraires de spécialisation.

En conséquence, voici celles que nous utiliserons pour toutes les fonctions spécialisées :

- $p = 3$: cette valeur est très réaliste (cf 2.2.2, section continuité).
- $n = 6$: cette valeur correspond simplement à l'exemple "type" de la figure 2.1.
- $P[7] = \{P_0, \ldots, P_6\}$: découle de n.
- $U[10] = \{u_0, \ldots, u_{10}\}$: découle de n et p.

Comme il serait trop fastidieux de trouver des valeurs pour ces deux derniers paramètres, et que substituer leurs valeurs dans le code spécialisé ne serait pas toujours très lisible, nous préférons les laisser sous forme de variables symboliques. Cela explique en partie la nécessité d'un opérateur $\widehat{expr}$.

4.2.3 Contexte d'exécution des algorithmes

- **Contexte dynamique :**

  Dans un contexte d'exécution dynamique (ou temps réel), on devra choisir des algorithmes d'approximation performants d'abord en temps de calcul puis en espace mémoire, quitte à faire des concessions sur la qualité visuelle du tracé.

- **Contexte statique :**

Dans ce contexte, nous devons afficher des objets dont les paramètres sont globalement invariants au cours d'une session X. Nous pouvons donc reporter leur calcul à la compilation, et en conséquence nous pouvons nous permettre de déterminer des approximations de meilleure qualité, c'est-à-dire faisant a priori appel à plus de segments.

Mais dans la mesure où nous serons finalement tenus d'afficher et de transmettre ces segments à travers le réseau[4] pendant la phase d'exécution, il serait souhaitable de chercher des approximations toujours aussi fidèles, mais pouvant éventuellement requérir moins de segments. Cette économie peut s'avérer utile pour des objets, tels les caractères de polices, qui seront affichés un grand nombre de fois.

Dans ce contexte, on peut raisonnablement envisager de stocker les approximations calculées dans une table de hachage. Sa taille en mémoire devant rester raisonnable, on considérera si nécessaire des stratégies de type FIFO ou LRU pour limiter le nombre d'entrées.

[4]dans le cas général, s'entend.

4.3 Spécialisation des méthodes de tracé

4.3.1 Spécialisation du calcul des fonctions de Schoenberg

4.3.1.1 Code Standard

Considérons le pseudo-code C issu des algorithmes développés dans [PT95] (pp68–70) visant à calculer la position d'un point d'une courbe NURBS pour une valeur donnée de t.

- La fonction `FindSpan` cherche à quel intervalle du vecteur nodal U appartient la valeur de t. Pour mémoire, p est le degré de la courbe et n le nombre de ses points de contrôle (définis dans l'espace homogène $\mathbb{R}^4$).

  D'après les contraintes d'extrémalité et nos conventions, le vecteur nodal a pour forme :

  $$U_{n+p+1} = \{\underbrace{0,0,0,0}_{p+1}, u_{p+1}, \dots, u_n, \underbrace{1,1,1,1}_{p+1}\}$$

  Donc, la recherche ne se déroule que sur le vecteur réduit à $\{0, u_{p+1}, \dots, u_n, 1\}$, de longueur $n + p + 2 - 2p = n - p + 2$. D'où les valeurs d'initialisation des indices de la recherche binaire, et le test de départ vérifiant si t est égal à la valeur maximale (1) du vecteur nodal. Remarquons que la même précaution aurait pu être prise pour $t = 0$: peut-être l'auteur a-t-il voulu minimiser le nombre de tests préalables.

```
int FindSpan(int n, int p, float t, float U[n+p+2])
{
  int low = p, high = n + 1, mid = (low + high) / 2;

  if (t == U[n + 1]) return n;

  while (t < U[mid] || t >= U[mid + 1])
    {
      if (t < U[mid]) high = mid;
      else low = mid;
      mid = (low + high) / 2;
    }
  return mid;
}
```

  Nous pouvons alors dénombrer les opérations nécessaires dans tous les cas et pour chaque itération (cf tableau 4.3.1.1 :

- La fonction `Schoenberg` calcule les $p+1$ valeurs des fonctions d'influence valides sur cet intervalle :

```
float N[p + 1] Schoenberg(int p, int i, float t, float U[n+p+2])
{
  int j, r;
  float N[p + 1], Left[p + 1], Right[p + 1], saved, temp;
  N[0] = 1.0;
```

opération	opérations d'initialisation	opérations par itération
=	3	3
comparaisons *float*	1	3
tests *if*/*while*	1	2
ops booléennes	1	3
+	3	2
/	1	1
[*var*]	1	3

FIG. 4.3 – opérations effectuées par l'algorithme Findspan

```
  for (j = 1; j <= p; j++)
    {
      Left[j] = t ⊖ U[i + 1 - j];
      Right[j] = U[i + j] ⊖ t;
      saved = 0.0;

      for (r = 0; r < j; r++)
        {
          temp = N[r] ⊘ (Right[r + 1] ⊕ Left[j - r]);
          N[r] = saved ⊕ Right[r + 1] ⊗ temp;
          saved = Left[j - r] ⊗ temp;
        }
      N[j] = saved;
    }
  return N;
}
```

- Enfin, la fonction `NURBSPoint` emploie les fonctions précédentes pour calculer la position du point de la courbe, obtenu par projection de $\mathbb{R}^4$ dans $\mathbb{R}^3$ (cf équations 2.8– 2.10) :

```
PointF3 NURBSPoint(int n, int p, float t, float U[n+p+2], PointF4 P[n+1])
{
  int span = FindSpan(n, p, t, U);
  float N[p + 1] = Schoenberg(p, span, t, U);
  PointF4 CH ≡ 0.0;

  for (j = 0; j <= p; j++)
    {
      CH ≡ CH ⨁ (N[j] ⨂ P[span - p + j]);
    }

  Project(CH);
}
```

La fonction auxiliaire `Project` calcule bien entendu la projection dans $\mathbb{R}^3$ du point exprimé dans $\mathbb{R}^4$.

Le seul point remarquable est que l'on a pré-calculé la division dans la mesure où la division flottante est a priori toujours beaucoup plus coûteuse que la multiplication (cf hypothèses du tableau 4.2.1.2).

```
/* project a R4 point into R3 */
PointF3 Project(PointF4 P)
{
  float 1_by_w = 1 ⊘ P[4];
  return PointF3(P[1] ⊗ 1_by_w, P[2] ⊗ 1_by_w, P[3] ⊗ 1_by_w);
}
```

4.3.1.2 Influence de la spécialisation

Rappels :
- t est dynamique.
- p, n, P, U_m sont statiques. Ils peuvent alors être considérés comme des variables globales, et en conséquence retirés des signatures des fonctions qui les employaient.

- **Fonction FindSpan :**

La fonction procède essentiellement à des tests, et tous dépendent de t dynamique. En conséquence, elle n'est (presque) pas spécialisable.

On peut tout de même, afin d'économiser 4 opérations entières et un accès mémoire, remplacer les deux premières lignes de code par :
`int low = 3, high = 7, mid = 5;`
`if (t ==` $\widehat{U[7]}$`) return 6;`

Cependant, un algorithme de recherche binaire sur un vecteur de N éléments effectue au plus $log(N)$ itérations, ce qui fait ici $log(n-p+2)$, soit 2 du fait des hypothèses de spécialisation. On peut alors estimer le nombre moyen d'opérations effectuées (cf tableau

opération	opérations effectuées	opérations économisées	reste
=	9	0	9
comparaisons $float$	7	0	7
tests $if/while$	5	0	5
ops booléennes	7	0	7
+	7	4	3
/	3	1	2
$[var]$	7	1	6

FIG. 4.4 – opérations effectuées et économisées par l'algorithme Findspan

Après pondération, nous avons besoin de 123 UAE en moyenne pour une version standard, et de 107 UAE pour la version spécialisée, soit une économie modique de 13%.

- **Fonction Schoenberg :**

Les valeurs des itérateurs j, r des deux boucles imbriquées décrivent p. Celui-ci étant connu, les boucles peuvent être déroulées (ce qui supprime une addition, comparaison, et affectation par itération). De même, Les calculs d'indices de tableau qui dépendent de j, r peuvent aussi être effectués.

Cependant, i, résultat de la fonction `FindSpan`, dépend de t : c'est donc un paramètre dynamique.

– Déterminons alors la version spécialisée de la fonction :

```
float N[4] Schoenberg(int i, float t)
{
  /* int j, r; */
  float N[4], Left[4], Right[4], saved, temp;
  N[0] = 1.0;

  /* j = 1 */
  Left[1] = t ⊖ U[i];
  Right[1] = U[i + 1] ⊖ t;
  saved = 0.0;

      /* r = 0 ; 0 < 1 */
      temp = N[0] ⊘ (Right[1] ⊕ Left[1]);
      N[0] = saved ⊕ Right[1] ⊗ temp;
      saved = Left[1] ⊗ temp;

  N[1] = saved;

  /* j = 2 */
  Left[2] = t ⊖ U[i - 1];
  Right[2] = U[i + 2] ⊖ t;
  saved = 0.0;

      /* r = 0 ; 0 < 2 */
      temp = N[0] ⊘ (Right[1] ⊕ Left[2]);
      N[0] = saved ⊕ Right[1] ⊗ temp;
      saved = Left[2] ⊗ temp;

      /* r = 1 ; 0 < 2 */
      temp = N[1] ⊘ (Right[2] ⊕ Left[1]);
      N[1] = saved ⊕ Right[2] ⊗ temp;
      saved = Left[1] ⊗ temp;

  N[2] = saved;

  /* continuing up to j = p */
  ⋮

  return N;
}
```

– Calculons alors dans chaque boucle le nombre normal d'opérations effectuées et le nombre d'opérations économisées par spécialisation (cf figure 4.3.1.2) :

opération	opérations effectuées	opérations économisées
$=$	5	1
$\leq$	1	1
$+$	3	1
$-$	1	1
$\ominus$	2	0
$[var]$	5	3

opération	opérations effectuées	opérations économisées
$=$	4	1
$<$	1	1
$+$	3	3
$-$	2	2
$\oplus$	2	0
$\oslash$	1	0
$\otimes$	2	0
$[var]$	6	6

FIG. 4.5 – opérations effectuées et économisées dans chaque boucle

– Comptons ensuite le nombre total d'opérations de l'algorithme (cf figure 4.3.1.2), en remarquant que :
  – La première boucle "`j de 1 à p`" effectue p itérations.
  – La boucle imbriquée "`r de 0 à j - 1`" effectue de 1 itération pour $j = 1$ à p itérations pour $j = p$. D'après l'équation 4.1, cette boucle effectuera donc $\frac{1}{2}(p^2 - p)$ itérations au final.

opération	opérations effectuées	$p = 3$	opérations économisées	$p = 3$	% de gain ($p = 3$)	reste ($p = 3$)
$=$	$2p^2 + 7p + 1$	40	$\frac{1}{2}p^2 + \frac{3}{2}p$	9	22	31
$\leq$	p	3	p	3	100	0
$<$	$\frac{1}{2}p^2 + \frac{1}{2}p$	6	$\frac{1}{2}p^2 + \frac{1}{2}p$	6	100	0
$+$	$\frac{3}{2}p^2 + \frac{9}{2}p + 3$	30	$\frac{3}{2}p^2 + \frac{5}{2}p + 3$	21	70	9
$-$	$p^2 + 2p$	15	$p^2 + 2p$	15	100	0
$\oplus$	$p^2 + p$	12	0	0	0	12
$\ominus$	$2p$	6	0	0	0	6
$\otimes$	$p^2 + p$	12	0	0	0	12
$\oslash$	$\frac{1}{2}p^2 + \frac{1}{2}p$	6	0	0	0	6
$[var]$	$3p^2 + 8p$	51	$3p^2 + 5p$	42	82	9

FIG. 4.6 – opérations effectuées et économisées dans la fonction Schoenberg

D'après la figure 4.3.1.2, les opérations qui ont pu être supprimées regroupent toutes les comparaisons entières, la quasi-totalité des additions et soustractions entières, le quart des affectations et presque tous les accès à des tableaux par indice variable. Cependant, aucune opération sur les nombres flottants n'a pu être économisée, ce qui tempère le gain de la spécialisation dans la mesure où elles sont bien plus coûteuses que les opérations entières.

Après application des facteurs relatifs de pondération sur chaque opération, on arrive à 397 UAE nécessaires pour la version standard, et 301 pour la version spécialisée, ce qui représente une économie non négligeable de 24%.

$$0 + 1 + 2 + \ldots + n = \sum_{i=0}^{n} i = \frac{n^2 + n}{2} \tag{4.1}$$

- **Fonction NURBSPoint :**

Les variables *span*, N sont dynamiques puisqu'elles dépendent de la valeur de t. Cependant, nous remarquons qu'il y a encore une boucle sur p : nous allons pouvoir encore procéder à un dépliage de boucle :

```
PointF3 NURBSPoint(float t)
{
  int span = FindSpan(t);
  float N[4] = Schoenberg(span, t);
  PointF4 CH ≡ 0.0;

  CH ≡ CH ⊕ (N[0] ⊗ P[span - 3]);
  CH ≡ CH ⊕ (N[1] ⊗ P[span - 2]);
  CH ≡ CH ⊕ (N[2] ⊗ P[span - 1]);
  CH ≡ CH ⊕ (N[3] ⊗ P[span - 0]);

  Project(CH);
}
```

Comptons alors les opérations effectuées par `NURBSPoint`, fonctions auxiliaires exclues :

opération	opérations effectuées	$p = 3$	opérations économisées	$p = 3$	% de gain ($p = 3$)	reste ($p = 3$)
$=$	$5p + 9$	24	$p + 1$	4	16	20
$\leq$	p+1	4	p+1	4	100	0
$+$	$2p + 3$	9	$2p + 3$	9	100	0
$-$	$p + 1$	4	0	0	0	4
$\oplus$	$4p + 4$	16	0	0	0	16
$\otimes$	$4p + 4$	16	0	0	0	16
$[var]$	$8p$	24	$4p$	12	50	12

FIG. 4.7 – opérations effectuées et économisées dans la fonction NURBSPoint

D'après la figure 4.3.1.2 et près application des facteurs relatifs de pondération sur chaque opération, on arrive à 257 UAE nécessaires pour la version standard, et 228 pour la version spécialisée, ce qui représente une économie assez modique de 11%.

La fonction `Project` n'est à l'évidence pas spécialisable. Son coût est de 34 UAE.

4.3.1.3 Bilan

Nous pouvons alors comptabiliser l'ensemble des opérations effectuées et économisées pour calculer la position d'un point de la courbe pour une valeur de t donnée (cf tableau 4.3.1.3). Il ne faut pas oublier que nous avons dû estimer le nombre d'opérations de `Findspan`, ce qui ne devrait pas trop fausser les résultats du fait de son coût modéré vis-à-vis de l'ensemble.

algorithme	UAE algo standard	UAE algo spécialisé
Findspan	123	107
Schoenberg	397	301
NurbsPoint	257	228
Project	34	34
Total	811	670

Fig. 4.8 – opérations effectuées sur l'ensemble du processus avant et après spécialisation

L'économie d'UAE réalisée par spécialisation atteint alors les 17%, ce qui n'est pas énorme mais reste tout de même fort appréciable pour un ensemble d'algorithmes susceptibles d'être appelés un grand nombre de fois pour calculer une courbe.

4.3.2 Spécialisation de l'algorithme de *de Casteljau*

4.3.2.1 Code standard

```
PointF3 Casteljau(int n, float t, PointF3 P[n+1])
{
  PointF3 Q[n+1] ;

  /* copied array sparing original control points*/
  for (i = 0; i <= n; i++) {Q[i] ≡ P[i];}

  for (k = 1; k <= n; k++){
    for (i = 0; i <= (n - k); i++)
      {Q[i] ≡ (1.0 ⊖ t) ⊗ Q[i] ⊕ t ⊗ Q[i + 1];}}

  return Q[0];
}
```

4.3.2.2 Code spécialisé

Le nombre n de points de contrôle d'une courbe de Bézier coïncide avec son degré. Donc, ici $n = p = 3$ est statique, ce qui nous permet de dérouler toutes les boucles, et de plier certains calculs d'indice :

```
PointF3 Casteljau(float t)
{
  PointF3 Q[n+1];

  /* copied array sparing original control points*/
  Q[0] ≡ P0;
  Q[1] ≡ P1;
  Q[2] ≡ P2;
  Q[3] ≡ P3;

  /* k = 1 ; 1 <= 3 */
  Q[0] ≡ (1.0 ⊖ t) ⊗ Q[0] ⊕ t ⊗ Q[1];
  Q[1] ≡ (1.0 ⊖ t) ⊗ Q[1] ⊕ t ⊗ Q[2];
  Q[2] ≡ (1.0 ⊖ t) ⊗ Q[2] ⊕ t ⊗ Q[3];

  /* k = 2 ; 2 <= 3 */
  Q[0] ≡ (1.0 ⊖ t) ⊗ Q[0] ⊕ t ⊗ Q[1];
  Q[1] ≡ (1.0 ⊖ t) ⊗ Q[1] ⊕ t ⊗ Q[2];

  /* k = 3 ; 3 <= 3 */
  Q[0] ≡ (1.0 ⊖ t) ⊗ Q[0] ⊕ t ⊗ Q[1];

  return Q[0];
}
```

En rappelant qu'une opération vectorielle consiste ici en 3 opérations flottantes, nous pouvons en déduire le tableau 4.3.2.2 :

opération	opérations effectuées	$p = 3$	opérations économisées	$p = 3$	% de gain ($p = 3$)	reste ($p = 3$)
$=$	$2p^2 + 7p + 4$	43	$\frac{1}{2}p^2 + \frac{5}{2}p + 1$	13	30	30
$\leq$	$\frac{1}{2}p^2 + \frac{5}{2}p + 1$	13	$\frac{1}{2}p^2 + \frac{5}{2}p + 1$	13	100	0
$+$	$p^2 + 3p + 2$	20	$p^2 + 3p + 2$	20	100	0
$-$	$\frac{1}{2}p^2 - \frac{1}{2}p$	6	$\frac{1}{2}p^2 - \frac{1}{2}p$	6	100	0
$\oplus$	$\frac{3}{2}p^2 + \frac{3}{2}p$	18	0	0	0	18
$\ominus$	$\frac{1}{2}p^2 + \frac{1}{2}p$	6	0	0	0	6
$\otimes$	$3p^2 + 3p$	36	0	0	0	36
$[var]$	$5p + 5$	18	$5p + 5$	18	100	0

FIG. 4.9 – opérations effectuées et économisées dans l'algorithme de *de Casteljau*

On déduit du tableau 4.3.2.2 par spécialisation, on a pu supprimer près du tiers des affectations, la quasi-totalité des opérations entières, toutes les comparaisons et accès variables aux tableaux, mais aucune opération flottante.

Cela nous fait un total de 460 UAE pour une version standard, contre 390 pour une version spécialisée, soit une économie de 15%.

4.3.3 Spécialisation de l'algorithme de *de Boor*

4.3.3.1 Code standard

```
PointF3 deBoor(int n, int p, float t, float U[n+p+2], PointF4 P[n+1])
{
  int span = FindSpan(n, p, t, U);
  int i, j, k;
  PointF4 Q[p+1];
  float alpha;

  /* copied array sparing original control points */
  for (j = 0; j <= p; j++) {Q[j] ≡ P[span - j];}

  for (k = 1; k <= p; k++){
    for (j = span; j >= (span - p + k); j--)
      {
        alpha = (t ⊖ U[j]) ⊘ (U[j + p + 1 - k] ⊖ U[j]);
        omalpha = 1.0 ⊖ alpha;
        i = span - j;
        Q[i] ≡ omalpha ⊗ Q[i + 1] ⊕ alpha ⊗ Q[i];
      }
  }

  return Project(Q[0]);
}
```

4.3.3.2 Code spécialisé

Si la première boucle peut être facilement déroulée (elle ne dépend que de p), ce n'est pas le cas de la seconde qui dépend de la valeur dynamique de *span*. Si l'on s'en tient à ces hypothèses, le gain de la spécialisation va être minime, puisque la boucle extérieure n'effectue que p itérations, contre $\frac{p^2-p}{2}$ pour la boucle intérieure.

Pour remédier à ce problème, on peut remarquer que les valeurs de *span*, indice entier du vecteur nodal, se restreignent à l'intervalle $[p, n]$. Cet intervalle de valeurs entières est fini et dépend de valeurs statiques : il peut donc être lui aussi déclaré comme paramètre statique. En conséquence, on peut dérouler la boucle intérieure pour chaque valeur possible de *span*, la sélection du code à exécuter se faisant une seule fois par un *switch*(*int*), instruction très peu coûteuse, d'autant plus que les différentes valeurs possibles sont consécutives.

Aussi, dans la mesure où les indices des accès au vecteur nodal sont entièrement déterminés par des paramètres statiques, on peut remplacer les accès à ce vecteur par les valeurs correspondantes, puisque le vecteur est lui aussi un paramètre statique.

```
PointF3 deBoor(float t)
{
  int span = FindSpan(t);
  int i, j, k;
  PointF4 Q[4];
```

```
float alpha;

switch(span)
  {
  case 3:
    /* copied array sparing original control points */
    Q[0] ≡ P[3]; Q[1] ≡ P[2]; Q[2] ≡ P[1]; Q[3] ≡ P[0];

    /* k = 1 ; k <= 3 */
          /* j = 3 ; j >= 1 */
          alpha = (t ⊖ U[3]) ⊘ (U[6] ⊖ U[3]);
          omalpha = 1.0 ⊖ alpha;
          /* i = 0 */
          Q[0] ≡ omalpha ⊗ Q[1] ⊕ alpha ⊗ Q[0];

          /* j = 2 ; j >= 1 */
          alpha = (t ⊖ U[2]) ⊘ (U[5] ⊖ U[2]);
          omalpha = 1.0 ⊖ alpha;
          /* i = 1 */
          Q[1] ≡ omalpha ⊗ Q[2] ⊕ alpha ⊗ Q[1];

          /* j = 1 ; j >= 1 */
          alpha = (t ⊖ U[1]) ⊘ (U[4] ⊖ U[1]);
          omalpha = 1.0 ⊖ alpha;
          /* i = 2 */
          Q[2] ≡ omalpha ⊗ Q[3] ⊕ alpha ⊗ Q[2];

    /* k = 2 ; k <= 3 */
          /* j = 3 ; j >= 2 */
          alpha = (t ⊖ U[3]) ⊘ (U[5] ⊖ U[3]);
          omalpha = 1.0 ⊖ alpha;
          /* i = 0 */
          Q[0] ≡ omalpha ⊗ Q[1] ⊕ alpha ⊗ Q[0];

          /* j = 2 ; j >= 2 */
          alpha = (t ⊖ U[2]) ⊘ (U[4] ⊖ U[2]);
          omalpha = 1.0 ⊖ alpha;
          /* i = 1 */
          Q[1] ≡ omalpha ⊗ Q[2] ⊕ alpha ⊗ Q[1];

    /* k = 3 ; k <= 3 */
          /* j = 3 ; j >= 3 */
          alpha = (t ⊖ U[3]) ⊘ (U[4] ⊖ U[3]);
          omalpha = 1.0 ⊖ alpha;
          /* i = 0 */
          Q[0] ≡ omalpha ⊗ Q[1] ⊕ alpha ⊗ Q[0];
    break;
  case 4:
```

```
  ⋮
    case 5:
  ⋮
    case 6:
  ⋮
    default: error();
    }

  return Project(Q[0]);
}
```

Le code obtenu peut paraître illisible et long à écrire, mais cela n'a aucune incidence dans la mesure où ce code sera calculé automatiquement à partir du code standard. De plus, le code spécialisé ne risque pas de consommer trop de mémoire, ce dans la mesure ou le nombre de points de contrôle d'une courbe NURBS n'est jamais très grand, et que la quantité de code à ajouter par point de contrôle se compte seulement en dizaines d'octets. Il est alors probable qu'après la compilation, ce code supplémentaire n'implique pas une augmentation sensible de la taille du programme obtenu.

opération	opérations effectuées	$p = 3$	opérations économisées	$p = 3$	% de gain ($p = 3$)	reste ($p = 3$)
$=$	$4p^2 + 10p + 1$	67	$p^2 + p$	12	17	50
$\leq, \geq$	$\frac{1}{2}p^2 + \frac{5}{2}p$	12	$\frac{1}{2}p^2 + \frac{5}{2}p$	12	100	0
$+$	$2p^2 + 4p + 1$	31	$2p^2 + 4p + 1$	31	100	0
$-$	$2p^2 + 3p$	27	$2p^2 + 3p$	27	100	0
$\oplus$	$2p^2 + 2p$	24	0	0	0	24
$\ominus$	$\frac{3}{2}p^2 + \frac{3}{2}p$	18	$\frac{1}{2}p^2 + \frac{1}{2}p$	6	33	12
$\otimes$	$4p^2 + 4p$	48	0	0	0	48
$\oslash$	$\frac{1}{2}p^2 + \frac{1}{2}p + 3$	9	0	0	0	6
$[var]$	$\frac{3}{2}p^3 + \frac{9}{2}p^2 + 3p$	63	$\frac{3}{2}p^3 + \frac{9}{2}p^2 + 3p$	63	100	0

FIG. 4.10 – opérations effectuées et économisées dans l'algorithme de *de Boor*

Par spécialisation (cf tableau 4.3.3.2), on a pu supprimer tous les accès mémoire, toutes les comparaisons et additions/soustractions entières, quelques affectations et pour une fois le tiers des soustractions flottantes.

Il faut cependant tenir compte des fonctions `FindSpan` et `Project` déjà étudiées pour évaluer le coût final du calcul d'un point sur une courbe NURBS pour une valeur arbitraire de t. On constate alors une économie finale de 23% sur l'ensemble du processus, dont 26% pour la fonction `deBoor`.

algorithme	UAE algo standard	UAE algo spécialisé
Findspan	123	107
deBoor	848	626
Project	34	34
Total	1005	770

FIG. 4.11 – opérations effectuées sur l'ensemble du processus avant et après spécialisation

4.3.4 Un scénario de spécialisation opportun

Nous avons commencé à spécialiser des algorithmes de tracé parmi les plus directs, ce en obtenant des économies d'instructions non négligeables (vers les 20%) sans être pour autant très conséquentes. Dans la mesure où le temps nous est compté, nous allons directement présenter la méthode qui nous semble, d'après celles que nous avons pu envisager, la plus susceptible d'économiser un grand nombre d'instructions par spécialisation.

Parmi les méthodes de tracé nécessitant le moins d'instructions pour s'exécuter, nous avons remarqué la méthode de *Forward Differencing* sur les courbes de Bézier, qui ne requiert que quelques additions par point de courbe après un bref calcul de mise en route (cf section 3.4.2.1). Nous avons aussi considéré une méthode pour transformer une courbe NURBS en une série de $n - p + 1$ courbes de Bézier (cf section 3.4.4). Il serait intéressant de déterminer dans quelle mesure on peut spécialiser l'utilisation conjointe de ces deux méthodes.

4.3.4.1 Code standard

- **Fonction NURBS_to_Bezier :**

  La fonction `NURBS_to_Bezier` convertit une courbe NURBS en une série de courbes de Bézier, chacune étant définie par $p+1$ points de contrôle. Ces derniers sont donc calculés et rangés dans un vecteur de $p+1$-uplets de points de contrôle de courbes de Bézier, exprimés dans l'espace homogène $\mathbb{R}^4$. La fonction renvoie donc ce vecteur accompagné de sa taille.

```
(PointF4[nb-sections][p+1], int)
NURBS_to_Bezier(int n, int p, float U[n+p+2], PointF4 P[n+1])
{
  int m = n + p + 1;
  int a = p, b = p + 1, nb = 0, mult, r;
  float numer, alphas[p];

  for (i = 0; i <= p; i++) {Q[nb][i] ≡ P[i];}

  while(b < m)
    {
      i = b;

      /* knot of multiplicity > 1 encountered: go to its last occurence */
      while ((b < m) && (U[b + 1] == U[b])) {b++;}
```

```
      mult = b - i + 1;
      /* knot insertion only up to multiplicity p */
      if (mult < p)
        {
          numer = U[b] ⊖ U[a];
          for (j = p; j < mult; j--)
            {alphas[j - mult - 1] = numer ⊘ (U[a + j] ⊖ U[a]);}

          /* insert knots up to multiplicity p */
          r = p - mult;
          for (j = 1; j <= r; j++)
            {
              alpha = alphas[k - s];
              Q[nb][k] ≡ alpha ⊗ Q[nb][k] ⊕ (1.0 ⊖ alpha) ⊗ Q[nb][k - 1];
            }
          /* control point of next segment */
          if (b < m) {Q[nb + 1][save] ≡ Q[nb][p];}
        }

      /* Bezier segment computed and stored */
      /* now initializing next segment computation */

      if (b < m)
        {
          for (i = p - mult; i <= p; i++) {Q[nb][i] ≡ P[b - p + i];}
          a = b;
          b = b + 1;
        }
    }
    return (Q, nb);
}
```

- **Fonction Bezier3_Forward_Render :**

  Cette fonction va calculer par la méthode de *Forward Differencing* les valeurs successives des points sur une courbe de Bézier, telle celles déterminées par la fonction `NURBS_to_Bezier` (cf section 3.4.2.1). Nous avons constaté dans cette section que la forme polynômiale et surtout les *forward differences* dépendaient complètement de la valeur du degré de la courbe.

  En conséquence, le code de l'algorithme sera uniquement valide pour le degré $p = 3$. Certes, cela correspond aux hypothèses de spécialisation établies, mais nous déplorons le fait que c'est le seul code envisagé jusqu'ici qui ne puisse être automatiquement calculé par une macro à partir d'une valeur du degré. Ce code est basé sur l'algorithme 3.4 et les équations 3.5 :

```
Bezier3_Forward_Render(int nb_steps, PointF4 P0, P1, P2, P3)
{
  /* determine k, the step value */
  float k = 1.0 ⊘ nb_steps;
```

```
  /* avoid unnecessary computations involving k */
  float k2 = k ⊗ k, k3 = k2 ⊗ k;
  float mk = 1.0 ⊖ k, mk2 = mk ⊗ mk, mk3 = mk2 ⊗ mk;
  float mkk = mk ⊖ k, mkkk = mkk ⊖ k;
  PointF4 FD0, FD1, FD2, FD3;

  /* determine the forward differences for k */
  FD0 ≡ P0;
  FD1 ≡ (mk3 ⊖ 1.0) ⨂ P0 ⨁ (3 ⊗ k ⊗ mk2) ⨂ P1
          ⨁ (3 ⊗ k2 ⊗ mk) ⨂ P2 ⨁ k3 ⨂ P3;
  FD2 ≡ (6 ⊗ (k2 ⊖ k3)) ⨂ P0 ⨁ (6 ⊗ k ⊗ (-mk2 ⊕ 2 ⊗ mkk ⊗ mkk)) ⨂ P1
          ⨁ (6 ⊗ (k2 ⊖ 3 ⊗ k3)) ⨂ P2 ⨁ (6 ⊗ k3) ⨂ P3;
  FD3 ≡ (-6 ⊗ k3) ⨂ P0 ⨁ (9 ⊗ k ⊗ (-4 ⊗ mkk ⊗ mkk ⊕ 3 ⊗ mkkk ⊗ mkkk ⊕ mk2 ⊗ mk2))
          ⨂ P1 ⨁ (-18 ⊗ k3) ⨂ P2 ⨁ (6 ⊗ k3) ⨂ P3;

  /* finally, incremental computation and rendering
     of Bezier curve points */
  for (i = 0; i < nb_steps; i++)
    {
      PlotOnRasterDevice(Project(FD0));
      FD0 ≡ FD0 ⨁ FD1;
      FD1 ≡ FD1 ⨁ FD2;
      FD2 ≡ FD2 ⨁ FD3;
    }
}
```

Evaluons le coût en opérations de cet algorithme (cf tableau 4.3.4.1) en distinguant les calculs initiaux incompressibles des calculs nécessaires pour le calcul d'un point. Nous excluons de notre test la fonction `PlotOnRasterDevice`, qui "affiche" effectivement le point à l'écran, dans la mesure où toutes les fonctions précédemment envisagées doivent implicitement l'utiliser elles aussi.

opération	opérations d'initialisation	opérations par point
$=$	24	13
$<$	0	1
$+$	0	1
$\oplus$	52	12
$\ominus$	6	0
$\otimes$	74	0
$\oslash$	1	0

FIG. 4.12 – opérations effectuées par l'algorithme de *Forward Differencing*.

4.3.4.2 Code spécialisé

- **Fonction NURBS_to_Bezier :**

Les paramètres de cette fonction sont tous des paramètres statiques. En conséquence, cette fonction va pouvoir s'exécuter intégralement pendant la phase de compilation. Les $n+p+1$ courbes de Bézier produites sont alors soumises à l'algorithme de tracé par *Forward Differencing*.

- **Fonction Bezier3_Forward_Render :**

  Nous savons que le défaut de la méthode de forward differencing est de ne pas permettre de modifier le pas de calcul afin de s'assurer que la distance entre deux points ne dépasse pas un seuil défini. Cependant, il est tout à fait possible dès la compilation de faire une estimation arbitraire d'après un critère géométrique raisonnable (e.g la diagonale du carré de côté le plus grand segment du réseau de contrôle), puis d'exécuter l'algorithme pendant la compilation afin de vérifier si les limites sont respectées. On pourra doubler le nombre de pas à chaque fois que les limites sont dépassées pour assurer une convergence quadratique de cette procédure de contrôle.

  Cette méthode peut paraître assez "brutale", mais dans le contexte de spécialisation de programmes, nous avons le droit de prendre tout le temps nécessaire pour la compilation de notre programme. De ce fait, seules les opérations dans la deuxième colonne du tableau 4.3.4.1 seront exécutées.

4.3.4.3 Bilan

Nous allons maintenant évaluer le nombre d'UAE nécessaires au calcul d'un seul point de la courbe : même si cela n'a pas trop de sens pour la méthode que nous étudions, cela nous permet de comparer nos résultats avec ceux des méthodes précédentes.

Pour éviter de calculer le nombre exact d'UAE nécessaires à la version non-spécialisée de `NURBS_to_Bezier`, on peut effectuer une approximation raisonnable en remarquant qu'il procède à peu près à $n-p$ itérations de l'algorithme de deBoor. La précision de ce résultat n'a pas d'importance car nous sommes surtout intéressés par le nombre d'instructions après spécialisation.

algorithme	UAE algo standard	UAE algo spécialisé
NurbsToBézier	2000	0
Bezier3Forward	879	51
Project	34	34
Total	2913	85

FIG. 4.13 – opérations effectuées sur l'ensemble du processus avant et après spécialisation

En conclusion, notre méthode de spécialisation permet d'économiser 97% des instructions qui auraient dû se dérouler pendant la phase d'exécution : c'est un résultat considérable.

Même en exécutant cette méthode sans aucune spécialisation, nous pouvons remarquer que si le coût des opérations initiales est assez élevé (près de 3000 UAE), il est très vite compensé dès qu'on calcule plusieurs points, chacun ne nécessitant que 85 UAE supplémentaires. Pour mémoire, les autres méthodes requièrent 700 UAE après spécialisation pour chaque point.

4.4 Conclusions de l'étude de spécialisation

Voici, en résumé de l'étude de spécialisation des méthodes de tracé, un récapitulatif des opérations requises pour le calcul d'un point sur une courbe NURBS (cf 4.4).

méthode	UAE algo standard	UAE algo spécialisé	% de gain
Schoenberg	811	670	17
Casteljau	460	390	15
deBoor	1005	770	23
NurbsToBezier	2913	85	97

FIG. 4.14 – opérations effectuées par chaque méthode de tracé de point avant et après spécialisation

4.4.0.4 Validité liée aux valeurs de spécialisation

Nous pouvons remarquer que ces résultats se basent sur des valeurs fixées aux paramètres de spécialisation que sont le vecteur nodal, le réseau de contrôle et le degré de la courbe. Cependant, nous avons pu constater que les performances des algorithmes employés n'étaient pas vraiment influencés par les valeurs de la plupart de ces paramètres :

- L'influence de n porte sur la partie statique du calcul de la méthode deBoor et de NurbsToBezier, mais n'a aucune incidence sur la performance de tous les programmes spécialisés. Par contre, n influence directement le nombre d'itérations de l'algorithme `Findspan`, employé dynamiquement par les méthodes Schoenberg et deBoor. Cependant, n ne devrait pas atteindre des valeurs excessives en pratique. De plus, seules 47 UAE sont rajoutées par itération de l'algorithme qui, de plus, est de complexité logarithmique sur n.

  En résumé, l'influence de la valeur de n est assez négligeable sur les résultats de notre étude.

- L'influence de p s'étend de manière très sensible sur les parties dynamiques de tous les codes des programmes. C'est pourquoi les résultats intermédiaires de nos études sont exprimés en fonction de p, si l'on excepte le cas particulier très délicat du *Forward Differencing* impliqué dans la méthode NurbsToBezier.

  En pratique, il est presque certain que la valeur $p = 3$ adoptée pour cette étude nous suffise complètement. Cependant, si nous devions envisager d'autres valeurs du degré, les résultats de notre étude seront assez faciles à mettre à jour.

- Toutes les autres valeurs des paramètres statiques n'ont aucune d'incidence notable (les valeurs du vecteur nodal sont utilisées pour certains calculs statiques).

4.4.0.5 Remarques liées à la spécialisation de programmes

Il semble que dans la plupart des cas, la spécialisation de programmes permet parfaitement d'économiser entre 15 et 25% du temps de calcul des fonctions de tracé. Ce résultat peut sembler modeste, mais il n'est pas négligeable compte tenu du grand nombre de fois que l'on va employer ces fonctions de tracé, dont la rapidité est souvent le goulot d'étranglement d'un système graphique[5]. Nous pouvons donc affirmer que dans le cas général des méthodes de tracé de NURBS, sous réserve d'hypothèses de spécialisation judicieuses, la méthode de spécialisation de programmes donne des résultats suffisamment significatifs pour ne pas être négligée.

La dernière méthode de tracé basée sur le *Forward Differencing* donne des résultats de spécialisation parfaitement spectaculaires. En réalité, elle a été choisie pour montrer un cas extrême où la spécialisation permet de complètement remettre en cause les performances relatives de plusieurs méthodes de tracé. Cette méthode présente en effet quelques lacunes que nous aborderons bientôt, et dont la résolution devrait probablement réduire ce taux de spécialisation. Mais au moins elle nous aura montré qu'une méthode de tracé a priori anodine peut être particulièrement bien optimisée selon le contexte de spécialisation fixé et les opportunités que présente son code. Le but est de nous encourager à poursuivre ces études de spécialisation sur d'autres méthodes de tracé, afin de trouver d'autres méthodes bien plus efficaces que leurs homologues non-spécialisées.

Cependant, il faut constater que la réalisation d'une telle étude est extrêmement fastidieuse, et oblige à exécuter de nombreux calculs mécaniques et répétitifs. Il serait intéressant d'envisager des méthodes pour automatiser au moins en partie ces tâches, surtout si l'étude doit se poursuivre sur d'autres algorithmes plus longs. Cela deviendrait même d'autant plus secourable si l'on devait être amené à varier la liste des paramètres déclarés comme statiques ou dynamiques, ce qui nous forcerait à reprendre la plupart des algorithmes. Cependant, aucune idée n'est encore entrevue dans ce sens à ce jour.

4.4.0.6 Remarques liées au choix d'un algorithme

Nous devons admettre que faute de temps, nous n'avons pu traiter qu'une partie assez restreinte des algorithmes de tracé envisagés dans la section 3.2. Notre étude comparative n'est donc pas très représentative de l'ensemble des méthodes existantes, et mériterait d'être étendue.

Cependant, la comparaison des méthodes Schoenberg (évaluation directe des valeurs des fonctions d'influence selon un algorithme pertinent) et deBoor (méthode géométrique procédant par "moyennes" successives des points de contrôle sans calcul des fonctions d'influence) donne indiscutablement la première gagnante, avec ou sans spécialisation. Ce résultat est pourtant surprenant a priori, dans la mesure où la méthode géométrique semblait plus simple et concise que son homologue. En réalité, la méthode géométrique doit travailler avec des points de dimension 4, ce qui quadruple le volume des calculs nécessaires par rapport à une méthode qui calcule les coefficients des points de contrôle et reporte les calculs vectoriels au dernier moment.

La méthode NurbsToBezier présente l'avantage d'une rapidité de calcul assez impressionnante qui laisse sur place les autres méthodes considérées. Cependant, Il est impossible de contrôler précisément le pas de calcul de t, contrairement aux autres méthodes envisagées qui nous laissent un total degré de liberté de ce côté. Si les suggestions émises pour limiter l'espacement maximal entre deux points sont assez pertinentes, elles vont nous obliger à sur-échantillonner la courbe dans le cas général, ce qui posera des problèmes si on souhaite avoir un nombre limité de segments composant notre approximation polygonale. C'est pourquoi dans le cas général la littérature suggère plutôt des méthodes toujours incrémentales pour limiter le volume des calculs, mais capables de s'adaper au profil de la courbe. Il est fort probable que ces méthodes soient elles aussi très performantes et qu'elles présentent des opportunités appréciables de spécialisation, mais il faudrait nous en assurer.

[5]d'après nos hypothèses, en tous cas.

Chapitre 5

Conclusions et perspectives

Nous avons montré que le modèle géométrique des NURBS était probablement le mieux adapté à la représentation de régions CLIM. En effet, la modélisation est considérablement facilitée (modélisation par détermination de trajectoires arbitraires, modifications locales possibles), la classe de formes représentables recouvre totalement celle de l'ancien modèle, et les incohérences constatées, liées au principe de composition de régions, pourront certainement être évitées.

En examinant le contexte d'exécution d'une session X, nous avons constaté que dans le cas général, le tracé des objets représentés par des NURBS (contour de régions ou caractères de polices) devait se faire sous forme polygonale pour éviter de saturer le goulot d'étranglement du réseau.

Nous avons alors entrepris une étude générale des principales méthodes de tracé pouvant produire de telles approximations. Nous avons essayé de prendre en compte le fait que ces méthodes devaient ultimement produire des approximations fidèles des courbes NURBS, problème peu trivial dans la mesure où l'écart de valeurs de paramètres d'une courbe reflète rarement l'espacement effectif des points correspondants sur la courbe. Cependant, nous n'avons pas encore fixé de critères généraux d'évaluation de la qualité des approximations calculées, même si des méthodes de contrôle ont été étudiées.

Afin de commencer à sélectionner certaines méthodes, nous avons entrepris une étude comparative basée sur la vitesse d'exécution. Pour ce faire, un protocole commun d'optimisation de programmes a été établi. En effet, nous avons décidé de procéder à la spécialisation des algorithmes de tracé envisagés, après avoir fixé des hypothèses de spécialisation adaptées au contexte d'exécution des algorithmes tel que nous pouvons le concevoir pour l'instant. Le but est autant de trouver une méthode de tracé satisfaisante que d'essayer de déterminer l'utilité potentielle de la spécialisation de programmes dans notre cadre.

La spécialisation s'est avérée capable dans le cas général d'améliorer de près de 20% en moyenne la vitesse d'exécution des algorithmes, ce qui est un rapport suffisamment élevé pour justifier l'opportunité d'une telle technique. Un cas assez particulier a été testé où nous pouvons économiser près de 97% du temps de calcul. Cela nous porte à croire qu'il serait intéressant d'examiner les algorithmes de tracé existants sous le jour nouveau de la spécialisation de programmes, technique jamais abordée dans la collection d'articles pourtant très exhaustive que nous avons rassemblé sur les courbes NURBS, Splines et de Bézier. Des opportunités très intéressantes pourraient en effet se dégager pour des cas plus "réalistes".

Cependant, la qualité des approximations n'a pas encore été prise en compte dans l'étude comparative, dont le domaine est d'ailleurs trop restreint. Il faudrait donc envisager d'y inclure la plupart des méthodes précédemment examinées, tout en essayant d'harmoniser autant que possible le niveau de qualité des approximations, afin que la vitesse d'exécution devienne alors un critère de sélection totalement objectif.

Il pourrait de même s'avérer intéressant d'élargir le champ des méthodes de tracé à toutes les méthodes trouvées dans la littérature, dans le but d'essayer de constituer une synthèse du domaine. Un tel document aurait été en effet très utile dans le cadre de notre recherche initiale.

Cependant, il serait probablement prioritaire d'implémenter les algorithmes spécialisés afin d'éprouver par l'expérience la validité de notre protocole de test, basé essentiellement sur la réduction du nombre d'instructions d'un code source, ces instructions étant pondérées par des valeurs relatives de coût basées sur les caractéristiques générales des processeurs. En effet, peut-être que les compilateurs peuvent interférer dans les résultats finaux, ou que ces coûts supposés sont trop approximatifs. Enfin, cela nous permettrait d'avoir une idée encore plus précise du contexte dans lequel nous avons réellement besoin d'exécuter nos algorithmes. C'est d'autant plus nécessaire que le degré d'optimisation par spécialisation de programmes dépend essentiellement de l'opportunité des hypothèses de contexte avancées.

Chapitre 6

Annexes

6.1 Définition des courbes coniques

Les *courbes coniques* (ou *sections coniques*) sont définies en 2D par l'intersection d'un cône et d'un plan. Suivant les valeurs successives de angle d'incidence i variant dans l'intervalle $[0°, 90°]$, on peut obtenir des cercles ($i = 0°$), des ellipses, des paraboles ou des hyperboles.

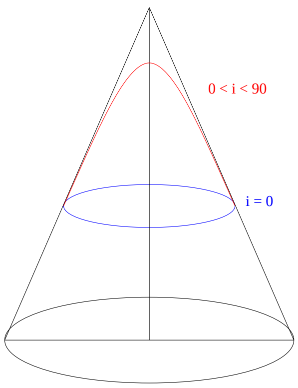

Fig. 6.1 – Exemples de sections coniques.

Remarquons que pour $i = 90°$, nous obtenons un triangle, et que plus i augmente, plus la section conique tend vers cette forme.

En 3D, la famille des coniques recouvre tous les modèles que l'on peut obtenir par extrusion de ces courbes de base (e.g. cylindres, tores, cônes).

6.2 Invariance affine des courbes B-Splines

En reprenant une preuve développée dans [Pie91] (p59), nous allons montrer que les courbes B-Splines vérifient la propriété d'invariance par transformation affine :

pour appliquer une transformation affine à une courbe B-Spline , il suffit de l'appliquer à son réseau de contrôle puis de calculer la nouvelle B-Spline qui en découle.

Une transformation affine A est définie comme la combinaison d'une transformation linéaire L (e.g. rotation, zoom) et d'une translation T. Associant un point P' à tout point P du plan, une transformation affine se caractérise alors par :

$$A[P] = L[P] + T \tag{6.1}$$

La transformation affine d'une courbe B-Spline se caractérise par l'application de l'équation 6.1 sur tout point de la courbe, d'où :

$$A[C(t)] = L[C(t)] + T \tag{6.2}$$

Par la définition d'une courbe B-Spline , nous pouvons établir l'égalité suivante :

$$L[C(t)] + T = \sum_{k=0}^{n} L[P_i]N_{k,p}(t) + T \tag{6.3}$$

Développons alors l'équation 6.2 en appliquant la définition des B-Splines :

$$\begin{aligned} A[C(t)] = L[C(t)] + T \Leftrightarrow \sum_{k=0}^{n} A[P_i]N_{k,p}(t) &= \sum_{k=0}^{n} (L[P_i] + T)N_{k,p}(t) \\ &= \sum_{k=0}^{n} L[P_i]N_{k,p}(t) + T \underbrace{\sum_{k=0}^{n} N_{k,p}(t)}_{=1 \text{ par normalité}} \\ &= \sum_{k=0}^{n} L[P_i]N_{k,p}(t) + T \end{aligned} \tag{6.4}$$

En combinant ce résultat avec 6.3, on obtient :

$$\sum_{k=0}^{n} A[P_i]N_{k,p}(t) = L[C(t)] + T$$

Enfin, en appliquant 6.2, on peut établir :

$$A[C(t)] = \sum_{k=0}^{n} A[P_i]N_{k,p}(t) \tag{6.5}$$

Cette dernière égalité signifie que pour calculer la transformation affine A d'une courbe B-Spline , il suffit d'appliquer cette transformation à chacun de ses points de contrôle et de calculer ensuite la courbe résultante. Cela correspond exactement à la définition de l'invariance affine.

6.3 Equation paramétrique d'un cercle

Le but est de montrer que l'équation paramétrique d'un cercle ne peut jamais être écrite sous forme de polynômes.

Par application du théorème de Pythagore, Un cercle vérifie l'équation :

$$x^2 + y^2 = 1 \Leftrightarrow x^2 + y^2 - 1 = 0 \tag{6.6}$$

Hypothèse 1 *Supposons que l'équation paramétrique d'un cercle puisse être représentée à partir de polynômes de degré n.*

Alors, nous pouvons poser :

$$\begin{cases} x(t) = a_0 + a_1 t^1 + \cdots + a_n t^n \\ y(t) = b_0 + b_1 t^1 + \cdots + b_n t^n \end{cases} \tag{6.7}$$

Par mesure de commodité, nous nous restreindrons pour cette exemple à des polynômes de degré 2 :

$$\begin{cases} x(t) = a + bt + ct^2 \\ y(t) = d + et + ft^2 \end{cases} \tag{6.8}$$

Dès lors, par 6.6 on obtient :

$$\begin{aligned} x^2 + y^2 - 1 = 0 \Leftrightarrow & (a + bt + ct^2)^2 + (d + et + ft^2)^2 - 1 = 0 \\ \Leftrightarrow & a^2 + b^2t^2 + c^2t^4 + 2abt + 2act^2 + 2bct^3 + \\ & d^2 + e^2t^2 + f^2t^4 + 2det + 2dft^2 + 2eft^3 - 1 = 0 \text{ (par 6.10)} \\ \Leftrightarrow & (c^2 + f^2)t^4 + 2(bc + ef)t^3 + (b^2 + 2ac + e^2 + 2df)t^2 + \\ & 2(ab + de)t + (a^2 + d^2 - 1) = 0 \end{aligned}$$

D'après 6.11, on a :

- $c^2 + f^2 = 0 \Rightarrow c = f = 0$
- $b^2 + 2ac + e^2 + 2df = 0 \Leftrightarrow b^2 + e^2 = 0 \Rightarrow b = e = 0$
- $a^2 + d^2 - 1 = 0$

On peut facilement en déduire que pour des polynômes de degré n, seuls les coefficients a_0 et b_0 sont non nuls. De ce fait, l'équation 6.7 devient :

$$\begin{cases} x(t) = a_0 \\ y(t) = b_0 \end{cases} \tag{6.9}$$

De toute évidence, cette paramétrisation ne peut représenter un cercle[1]. De ce fait, l'hypothèse posée est fausse. On peut donc affirmer qu'il est impossible de décrire l'équation d'un cercle à partir de polynômes

[1] c'est celle d'un point !

de degré n.

Rappels :

$$(a+b+c)^2 = a^2 + b^2 + c^2 + 2(ab + ac + bc) \tag{6.10}$$

$$\forall t \in \mathbb{R}, f(t) = a_0 + a_1 t^1 + \cdots + a_n t^n = 0 \Rightarrow \{a_0, a_1, \cdots, a_n\} = 0 \tag{6.11}$$

6.4 Exemple de *Forward Differencing*

Nous allons présenter ici un exemple d'application du *Forward Differencing*, technique permettant d'évaluer de manière incrémentale les valeurs successives d'un polynome.
Illustrons cette technique en évaluant le polynome cubique suivant pour un pas $k = 1$:

$$P(t) = 2t^3 + 3t^2 + 4t + 5 \tag{6.12}$$

$$\begin{aligned}
&\Delta^1 P = 2\,(t+k)^3 + 3\,(t+k)^2 + 4\,k - 2\,t^3 - 3\,t^2 \\
&\Delta^2 P = 2\,(t+2\,k)^3 + 3\,(t+2\,k)^2 - 4\,(t+k)^3 - 6\,(t+k)^2 + 2\,t^3 + 3\,t^2 \\
&\Delta^3 P = 12k^3 = \text{ constante}
\end{aligned}$$

Pour $t = 0$, on obtient :

$$\begin{aligned}
&P(0) = 5 \\
&\Delta^1 P(0) = 2k^3 + 3k^2 + 4k \\
&\Delta^2 P(0) = 12k^3 + 6k^2 \\
&\Delta^3 P(0) = 12k^3
\end{aligned}$$

Enfin, pour $k = 1$, on obtient :

$$\begin{aligned}
&P(0) = 5 \\
&\Delta^1 P(0) = 9 \\
&\Delta^2 P(0) = 18 \\
&\Delta^3 P(0) = 12
\end{aligned}$$

On peut alors calculer les valeurs successives de P(cf figure 6.2) :

t	$P(t)$	$\Delta^1 P$	$\Delta^2 P$	$\Delta^3 P$
0	5	9	18	12
1	14	27	30	12
2	41	57	42	12
3	98	99	54	12
4	197	153	66	
5	350	219		
6	569			

FIG. 6.2 – Valeurs du polynome 6.12 obtenues par *Forward Differencing*

Bibliographie

[Alt93] Laurent Alt. Rational linear reparametrization of nurbs and the blossoming principle. *Computer Aided Geometric Design*, 10 :465–467, 1993.

[AP92] Narayanan Anantakrishnan and Leslie Piegl. Integer de casteljau algorithm for rasterizing nurbs curves. *Computer Graphics Forum*, 11(2) :151–162, 1992.

[BFJ84] W. Boehm, G. Farin, and J.Kahmann. A survey of curve and surface methods in cagd. *Computer Aided Geometric Design*, 1(1) :1–60, 1984.

[Bla94] Carole Blanc. *Techniques de modélisation et de déformation de surfaces pour la synthèse d'images*. PhD thesis, Ecole Doctorale de Mathématiques et Informatique, Université Bordeaux I, September 1994.

[Blo99] Mark D. Bloomenthal. Error bounded approximate reparametrization of non-uniform rational b-spline curves, August 1999.

[Boe80] Wolfgang Boehm. Inserting new knots into b-spline curves. *Computer-Aided Design*, 12(4) :199–201, 1980.

[Boo72] Carl De Boor. On calculating with b-splines. *Journal of Approximation Theory*, 6 :50–62, 1972.

[Cat74] Edwin Earl Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, Department of Computer Science, University of Utah, December 1974.

[CLS85] Elaine Cohen, Tom Lyche, and Larry L. Schumacker. A survey of curve and surface methods in cagd. *ACM Transactions on Graphics*, 4(3) :171–181, July 1985.

[CM96] Giulio Casciola and Serena Morigi. Reparametrization of nurbs curves. *International Journal of Shape Modeling*, 2(2) :103–116, 1996.

[Com98] Compose Project, IRISA/INRIA - LaBRI, :http ://compose.labri.u-bordeaux.fr/. *Tempo Specializer - A Partial Evaluator for C*, 1998.

[Cox72] M. Cox. The numerical evaluation of b-splines. *Journal of Mathematic Applications*, 10(2) :134–149, 1972.

[CS85] Elaine Cohen and Larry L. Schumacker. Rates of convergence of control polygons. *Computer Aided Geometric Design*, 2 :229–235, 1985.

[dB78] Carl de Boor, editor. *A Practical Guide to Splines*. Springer Verlag, 1978.

[dC86] Pierre de Casteljau, editor. *Shape Mathematics and CAD*. London : Kogan Page, 1986.

[dC93] Pierre de Casteljau. *Polar Forms for Curve and Surface Modeling as used at Citroën*. London : Academic Press, 1993.

[Far90] G. Farin, editor. *Curves and Surfaces for Computer Aidd Geometric Design*. Academic Press, 1990.

[FP79] I. Faux and M. J. Pratt, editors. *Computational Geometry for Design and Manufacture*. Ellis-Horwood, 1979.

[FvDF+95] Foley, van Dam, Feiner, Hugues, and Philips, editors. *Introduction à l'Infographie*. Addison-Wesley, 1995.

[Har98] Harlequin group ltd, :http ://www.harlequin.com/. *LispWorks Common Lisp Interface Manager 2.0 User's Guide*, 1998.

[Jan01] Hin Jang. *Rendering Bézier Forms via Forward Differencing*. :http ://, 2001.

[JFB+00] Bailleul Jonathan, Bazzaro Fabrice, Degand Benoît, Hubert Renaud-Roland, and Pujos Michaël. *GLOS : Implémentation d'une Bibliothèque Graphique en Common Lisp*. Maîtrise Informatique, Université Bordeaux I, :http ://glos.sourceforge.net/, May 2000.

[KM83] P. A. Koparkar and S. P. Mudur. A new class of algorithms for the processing of parametric curves. *Computer-Aided Design*, 15(1) :41–45, 1983.

[LCWB80] Lane, Carpenter, Whitted, and Blinn. Scanline methods for displaying parametrically defined surfaces. *Communications ACM*, 23 :23–24, 1980.

[LR80] J M Lane and Riesenfeld. A theoretical development for the computer display of piecewise polynomial surfaces. *IEEE Trans. RAMI*, 2 :35, 1980.

[LR83] J.M. Lane and R.F. Riesenfeld. A geometric proof of the variation diminishing property of b-spline approximation. *Journal of Approximation Theory*, 37 :1–4, 1983.

[LSP87] Sheue-Ling Lien, Michael Shantz, and Vaughan Pratt. Adaptative forward differencing for rendering curves and surfaces. *Computer Graphics*, 21(4) :111–118, 1987.

[MGLV] Mounib Mekhilev, Pierre Germain-Lacour, and Henri Veyserre. Une nouvelle approche pour la restitution de courbes avec des b-splines rationnelles non uniformes.

[MY92] Scott McKay and William York. *Common Lisp Interface Manager Release 2.0 Specification*, 1992.

[Pie83] Leslie Piegl. A fast algorithm for parametric curve plotting. *Computer & Graphics*, 7(3) :361–367, 1983.

[Pie89] Leslie Piegl. Modifying the shape of rational b-splines curves. part 1 : curves. *Computer-Aided Design*, 21(8) :509–518, October 1989.

[Pie91] Leslie Piegl. On nurbs : A survey. *IEEE Computer Graphics & Applications*, 11(1) :55–71, January 1991.

[PP91] Hartmut Prautzsch and Bruce Piper. A fast algorithm to raise the degree of spline curves. *Computer Aided Geometric Design*, 8(3) :253–265, 1991.

[Pra84] Hartmut Prautzsch. Degree elevation of b-spline curves. *Computer Aided Geometric Design*, 1 :193–198, 1984.

[PT89] Leslie Piegl and Wayne Tiller. A menagerie of rational b-spline circles. *IEEE Computer Graphics & Applications*, 9(5) :48–56, 1989.

[PT94] Leslie Piegl and Wayne Tiller. Software-engineering approach to degree-elevation of b-spline curves. *Computer-Aided Design*, 26(1) :17–28, January 1994.

[PT95] Leslie Piegl and Wayne Tiller, editors. *The NURBS Book*. Springer, 1995.

[SC88] Michael Shantz and Sheue-Ling Chang. Rendering trimmed nurbs with adaptative forward differencing. *Computer Graphics*, 22(4) :189–198, 1988.

[Sch46] I. Schoenberg. Contributions to the problem of approximation of equidistant data by analytic functions. *Quarterly Journal of Applied Mathematics*, 4 :45–99, 1946.

[Sch97] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. *CMU Common Lisp User's Manual*, October 1997.

[Sil90] M.J. Silbermann. High-speed implementation of nonuniform rational b-splines (nurbs). *Curves and Surfaces in Computer Vision and Graphics (Proceedings of SPIE)*, 1251 :338–345, 1990.

[Wal] Bob Wallis. *Tutorial on Forward Differencing*, pages 594–602. Academic Press.