Module RAD
Bézier & B-Spline (2/2)

Nom : laxenaire
Prénom : flavian
2bts snir

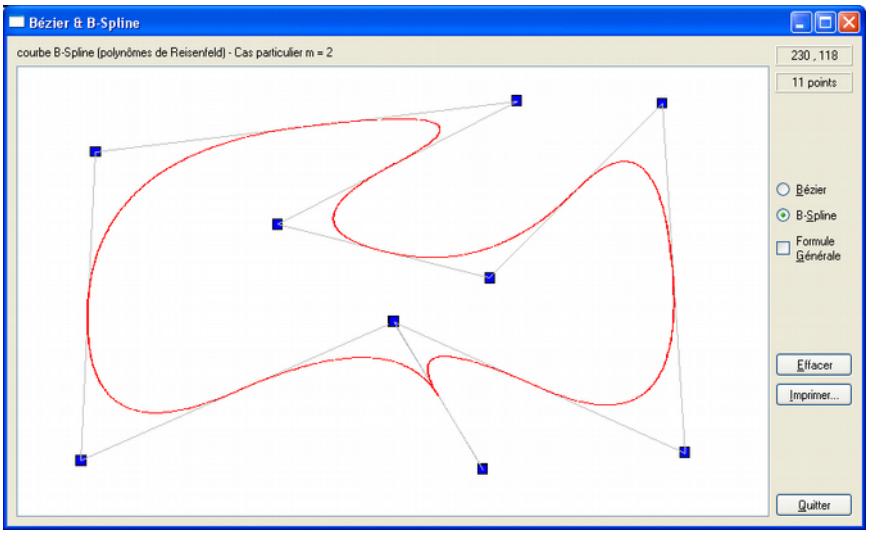
Pour ces travaux, vous devez utiliser une machine équipée du SDK Qt version 1.6.x ou supérieure ou à minima du Qt Framework version 5.3.x ou supérieure ; l'usage de l'EDI Qt Creator est cependant recommandé.

1. Rappel du Cahier des Charges

La finalité de ces travaux pratiques est la réalisation d'une application graphique permettant à l'utilisateur de construire des courbes gauches à partir d'une enveloppe segmentée.

L'application doit assurer la mise en oeuvre des courbes de Bézier ainsi que celle des B-Splines uniformes ; les annexes au présent document fournissent la théorie mathématique nécessaire au tracé des courbes.

Exemple de produit fini →



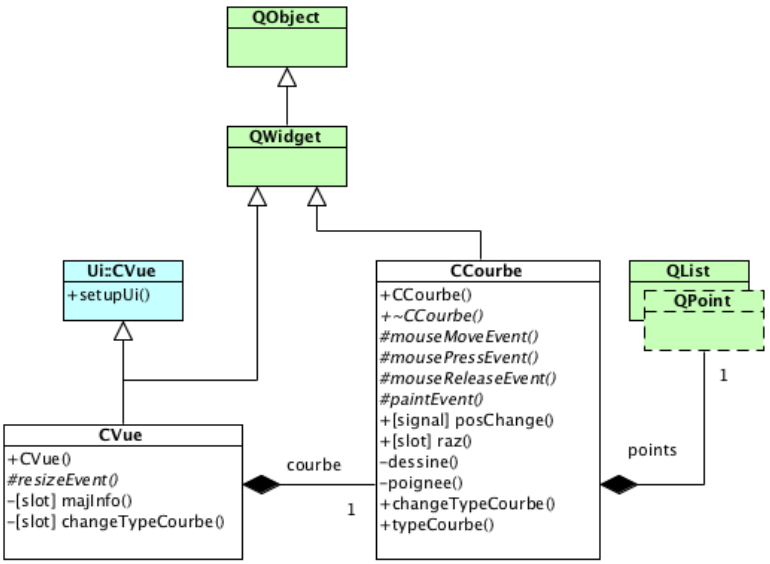
Le développement est exclusivement réalisé en programmation C++ avec la librairie Qt. Cette deuxième partie se charge des calculs mathématiques et du dessin des courbes.

2. IHM de l'application

La première partie de ces travaux vous a permis de construire pas à pas une IHM répondant au diagramme de classes ci-contre et conforme au *screen shot* ci-dessus.

Pour mémoire :

- la classe `CVue` est la classe d'interface de l'UI
- la classe `CCourbe` est la classe de gestion de l'enveloppe segmentée de la courbe



3. Relation entre l'enveloppe segmentée et la courbe

Les courbes qui nous intéressent sont toutes reconnues sous le vocable de *spline*. Mathématiquement, une B-spline est une combinaison linéaire de *splines* ; et les B-splines sont la généralisation des courbes de Bézier,

L'application doit être en mesure de montrer des courbes de Bézier de deux types : la première méthode traite les cas particuliers exposés en annexe, sur un maximum de quatre points (méthode des barycentres), tandis que la deuxième méthode met en oeuvre la formule mathématique générale.

Définissez une nouvelle classe `CBezier` dérivée de `QObject`. Le constructeur de `CBezier` est calqué sur celui de sa classe de base vers lequel il doit propager ses arguments.
Ceci est normalement automatiquement réalisé par l'assistant d'ajout de classe de Qt Creator.

Q1. Comment les paramètres du constructeur sont-ils propagés vers la classe de base ?

.....
.....

Un objet de classe `CBezier` va pouvoir se dessiner à partir du moment où il sera en possession de 3 éléments :

- le périphérique de sortie (*device*) sur lequel effectuer le dessin ;
- la liste des points constituant l'enveloppe ;
- la méthode de tracé à utiliser (formule générale ou non).

L'interface publique de `CBezier` doit donc proposer une méthode du style :

```
void traceCourbe(QPainter&, QList<QPoint>&, bool ) ;
```

Q2. Que devez-vous ajouter dans *cbezier.h* après ajout de cette méthode et tentative de compilation ?

Il faut include la librairie : "QPainter" .....

Le tracé proprement dit de la courbe sera assuré par l'une ou l'autre des méthodes protégées virtuelles :

- `virtual void casParticulier(QPainter& paint, QList<QPoint>& pts ) ;`
- `virtual void formuleGenerale(QPainter& paint, QList<QPoint>& pts ) ;`

Ces méthodes sont protégées et virtuelles parce que nous prévoyons de les surdéfinir en cas d'héritage pour tracer d'autres types de courbes (par exemple des B-Spline…).

Mettez en place ces 2 nouvelles méthodes avec pour le moment un corps d'instructions vide.

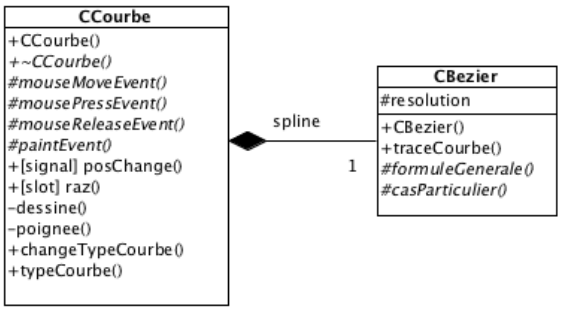
Q3. Comment allez-vous logiquement compléter `traceCourbe()` ?

```
void CBezier::traceCourbe(QPainter& paint, QList<QPoint>& points, bool formGen )
{
if(formGen) formuleGenerale(paint, points);
else casParticulier(paint, points);
}
```

Comme vous pouvez le voir en annexe, le tracé des *splines* repose sur l'évolution d'une variable *t* de 0 jusqu'à 1. La finesse du tracé dépend donc de la résolution utilisée ; une résolution égale à 500 signifie par exemple que *t* évoluera par pas de $1/500^{e}$ soit 0,002.

Ajoutez à la classe `CBezier` un membre protégé de type entier nommé `resolution`.
Initialisez ce membre à la valeur 100 dans le constructeur.

Le diagramme ci-contre montre la relation à établir entre les classes `CCourbe` et `CBezier`.



Q4. De quel type de relation s'agit-il ?

C'est une relation de composition

Q5. Où devez-vous placer chacun des éléments suivants ?

annonce de classe	`class CBezier ;`	dans ccourbe.h avant la création de class CCourbe
inclusion d'interface	`#include "cbezier.h"`	dans ccourbe.h
déclaration	`CBezier* spline ;`	dans ccourbe.h en tant que membre privé
création d'objet	`spline = new CBezier(this) ;`	dans ccourbe.cpp dans le constructeur
destruction d'objet	`delete spline ;`	dans ccourbe.cpp dans le destructeur

Q6. Il ne manque que l'appel : `spline->traceCourbe(paint, points, isFormGen ) ;`
Attention : à provoquer ssi `points` est non vide. Où allez-vous judicieusement insérer cet appel ?

Je l'ai placé dans la fonction dessine.
`if(!points.isEmpty()) spline->traceCourbe(paint, points, isFormGen);`

Bon, cette fois tout est en place, plus possible de reculer, il faut regarder de plus près l'annexe mathématique pour être en mesure de remplir nos méthodes de tracé…

4. Courbes de Bézier

Après une lecture attentive de l'annexe, vous comprenez que la méthode `formuleGenerale()` est liée à l'équation *(4)* tandis que la méthode `casParticulier()` va traiter les équations *(1)*, *(2)* et *(3)*.

Vous avez aussi noté que le codage de ces équations nécessite de disposer d'un certain nombre d'outils mathématiques : élévation d'une expression à une puissance, calcul de factorielle et calcul des combinaisons de *n* objets pris *k* à *k*. Même si la bibliothèque standard propose une fonction `pow()` pour l'élévation à la puissance, nous allons coder notre propre version simplifiée (la puissance est ici toujours entière).

Préparez 3 nouvelles méthodes protégées de CBezier : `double puiss(double x, int n ) ;` `double fact(int n ) ;` et `double cnk(int n, int k ) ;`

Une implémentation itérative de `fact()` vous est gracieusement fournie ci-contre →

```
1  double CBezier::fact(int n )
2  {
3      double  r = 1 ;
4      while ( n ) r *= n-- ;
5      return r ;
6  }
```

Q7. Comment pouvez-vous développer `puiss()` ?

```
double CBezier::puiss(double x, int n )
{
    double r = 1 ;
    while ( n) r *= x, n-- ;
    return r ;
}
```

Q8. Et la méthode `cnk()` ?

```
double CBezier::cnk(int n, int k )
{
    return (fact(n) / (fact(k) * fact(n - k)));
}
```

Commençons par les cas particuliers…

```
1  void CBezier::casParticulier(QPainter& paint, QList<QPoint>& pts )
2  {
3      paint.save() ;
4      paint.setPen( QPen(Qt::darkGreen, 2 ) ) ;          // sélection d'un crayon
5
6      QPoint  ptAct, ptPrec = pts[0] ;
7      int     n = pts.size() - 1 ;                       // ordre de la courbe
8
9      for ( int i = 0 ; i <= resolution ; i++ ) {
10
11         double  t = i / (double)resolution ;           // variation de 0 à 1
12         double  x = 0.0 , y = 0.0 ;
13         double  a, b, c, d ;
14
15         switch( n ) {
16         case 1 :                                       // équation (1)
17             a = 1 - t ;
18             b = t ;
19             x = pts[0].x() * a + pts[1].x() * b ;
20             y = pts[0].y() * a + pts[1].y() * b ;
21             break ;
22         case 2 :                                       // équation (2)
23             a = puiss(1 - t, 2 ) ;
24             b = 2 * t * ( 1 - t ) ;
25             c = puiss(t, 2 ) ;
26             x = pts[0].x() * a + pts[1].x() * b + pts[2].x() * c ;
27             y = pts[0].y() * a + pts[1].y() * b + pts[2].y() * c ;
28             break ;
29         default :                                      // équation (3)
30             a = puiss(1 - t, 3 ) ;
31             b = 3 * t * puiss(1 - t, 2 ) ;
32             c = 3 * puiss(t, 2 ) * ( 1 - t ) ;
33             d = puiss(t, 3 ) ;
34             x = pts[0].x() * a + pts[1].x() * b + pts[2].x() * c + pts[3].x() * d ;
35             y = pts[0].y() * a + pts[1].y() * b + pts[2].y() * c + pts[3].y() * d ;
36             break ;
37         }
38         ptAct.setX( (int)x ) ;
39         ptAct.setY( (int)y ) ;
40         paint.drawPoint(ptAct ) ;
41         paint.drawLine(ptPrec, ptAct ) ;
42         ptPrec = ptAct ;
43     }
44     paint.restore() ;
45 }
```

Q9. Quel est le rôle des lignes 38 à 42 ?

Les lignes 38 et 39 permette de sauvegarder le point actuel, celui qu'on vient de créer. La ligne 40 dessine ce point. La ligne 41 permet de faire un trait entre l'avant dernier point et le dernier (l'actuel). La ligne 42 sauvergarde l'avant dernier point en tant qu'actuel pour préparer le prochain trait...

Testez le programme ; vous devez être en mesure de tracer de jolis courbes de Bézier d'ordre 3 !

Q10. Pour la formule générale, le code ci-dessus peut être reproduit à l'exception des lignes 13 à 37. Par quelles instructions devez-vous remplacer ces lignes pour satisfaire l'équation *(4)* ?

```
for ( int k = 0 ; .....
.....
.....
.....
.....
```

Vérifiez que vous êtes maintenant capables de tracer des courbes de Bézier d'ordre quelconque.

5. Courbes B-Spline

Une grande partie du travail est déjà faite ! Nous allons réutiliser le code de `CBezier` en adaptant simplement les méthodes de tracé.

- Ajoutez au projet une nouvelle classe `CBSpline` dérivée de `CBezier` avec des surcharges des méthodes virtuelles `casParticulier()` et `formuleGenerale()`.
  N'oubliez pas le constructeur de `CBSpline` qui, même s'il est vide d'instructions, doit invoquer le constructeur de sa classe de base.

- Complétez `CCourbe::changeTypeCourbe()` pour prendre en compte cette nouvelle classe →
  Notez qu'un pointeur `CBSpline*` est compatible avec un `CBezier*`.

```
1  delete spline ;
2  if ( isBezier ) spline = new CBezier ;
3  else            spline = new CBSpline ;
4  update() ;
```

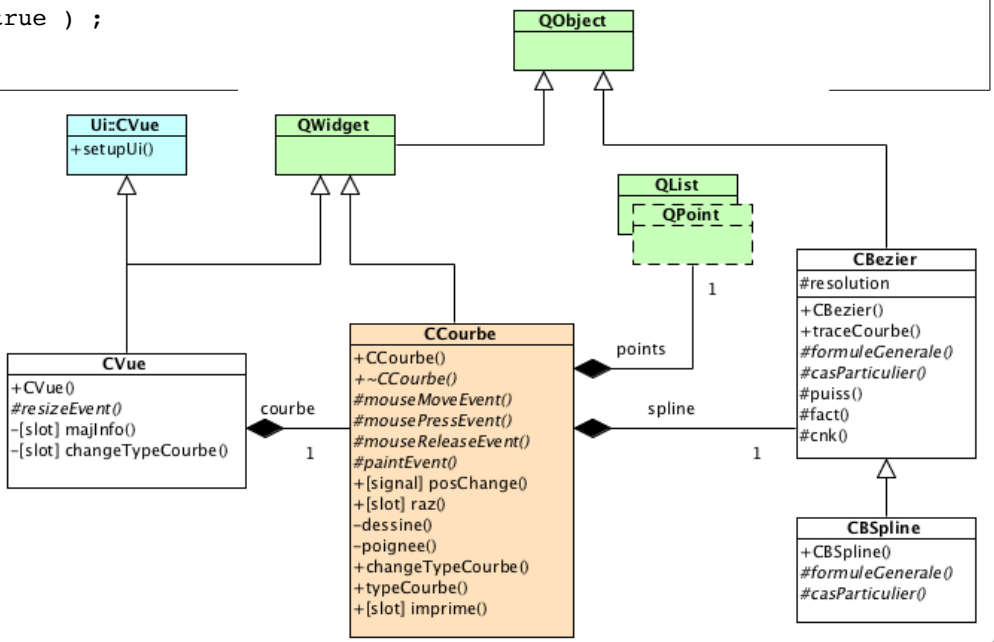
- À vous de coder les équations *(5)* et *(6)* (avec *m* = 3) pour voir (enfin) ces fameuses courbes B-Spline…

6. Finalisation

En cadeau pour finir ces travaux, le slot qui va vous permettre d'imprimer vos superbes créations *splinienne*s… (nécessite `QT += printsupport` et quelques inclusions).

```
1  void CCourbe::imprime()
2  {
3      QPrinter printer;
4      QPrintDialog*   dlg = new QPrintDialog(&printer, this ) ;
5      if ( dlg->exec() != QDialog::Accepted )    return ;
6
7      QPainter paint ;
8      if ( ! paint.begin( &printer ) )    return ;
9
10     int marge = (int)( 0.1 * printer.width() ) ;            // marge 10%
11     int w = (int)( 0.8 * printer.width() ) ;                // zone imp. 80%
12     int h = (int)( height() * (double)w / width() ) ;       // hauteur prop.
13
14     paint.drawText(marge, 50, typeCourbe() ) ;
15     paint.drawText(marge, 75, QString("(%1 points)").arg(points.size()) ) ;
16     paint.drawRect(marge, 100, w, h ) ;
17     paint.setViewport(marge, 100, w, h ) ;
18
19     dessine(paint, true ) ;
20     paint.end() ;
21 }
```

Et le diagramme de classes du produit fini !



7. Annexe 1 : Courbes de Bézier

Le modèle de Bézier (proposé par Pierre Bézier en 1962) a été développé dans les bureaux d'étude de Renault. Les courbes de Bézier sont une définition mathématique de courbes gauches construites par une enveloppe segmentée : ce sont des courbes paramétriques qui permettent très simplement, par construction itérée de barycentres (algorithme proposé conjointement par De Casteljau – Citroën), de réaliser des arcs de courbes continus d'extrémités imposées, avec des points de contrôle qui définissent les tangentes.

L'exploitation informatique du principe de Bézier présente un intérêt certain en DAO/CAO lors de la conception de formes complexes comme les éléments de carrosserie automobile…

Les courbes de Bézier et dérivés sont aussi à la base des polices vectorielles de caractères et images vectorielles utilisées en informatique.

Principe de base

Soit un segment M_0M_1 de barycentre G, avec M_i de coordonnées (x_i,y_i) dans un plan. Lorsqu'on associe respectivement à M_0 et à M_1 les poids (1-*t*) et (*t*) pour *t* variant de 0 à 1, l'équation paramétrique de la courbe C entre M_0, G et M_1 s'écrit :

(1) $\vec{OM}(t)=\vec{OM}_0.(1-t)+\vec{OM}_1.t$ soit $x(t)=x_0.(1-t)+x_1.t$ et $y(t)=y_0.(1-t)+y_1.t$

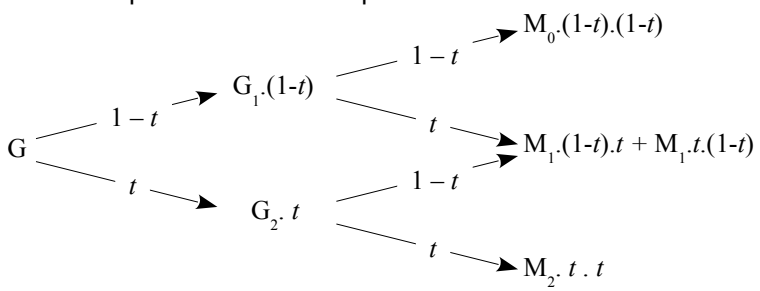
(1 – *t*) et (*t*) sont les polynômes de Bernstein de degré 1. La courbe respecte les tangentes imposées par le segment. Le calcul donne bien les coordonnées du barycentre G pour *t* = 0,5.

Suivant le même principe, il est possible de construire une courbe dont l'enveloppe est constituée de *n* segments. On parle alors de courbe de Bézier d'ordre *n* (l'ordre *n* est égal au nombre de points de construction moins un…).

Cas de l'ordre 2

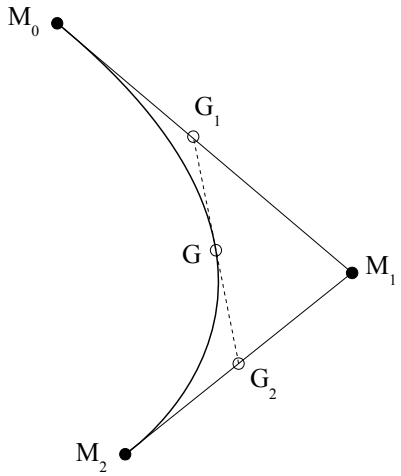
G barycentre de G_1G_2, G_1 barycentre de M_0M_1 et G_2 barycentre de M_1M_2.

Recherche des poids associés aux points M :



Les polynômes de Bernstein de degré 2 sont $(1 – t)^2$, $2.t.(1 – t)$ et t^2.
On obtient l'équation C (arc de parabole) :

(2) $\vec{OM}(t)=\vec{OM}_0.(1-t)^2+\vec{OM}_1.2.t.(1-t)+\vec{OM}_2.t^2$



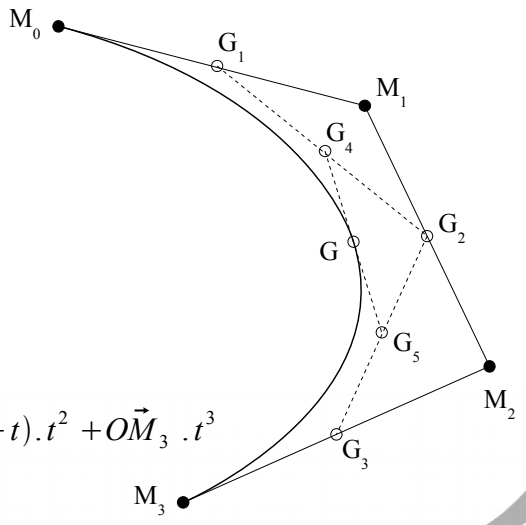
Cas de l'ordre 3

G barycentre de G_4G_5, G_4 barycentre de G_1G_2,
G_5 barycentre de G_2G_3, G_1 barycentre de M_0M_1,
G_2 barycentre de M_1M_2, G_3 barycentre de M_2M_3.

On montre par la même démarche que les polynômes de Bernstein de degré 3 sont $(1 – t)^3$, $3.t.(1 – t)^2$, $3.(1 – t).t^2$ et t^3

L'équation paramétrique de la courbe C s'écrit donc :

(3) $\vec{OM}(t)=\vec{OM}_0.(1-t)^3+\vec{OM}_1.3.t.(1-t)^2+\vec{OM}_2.3.(1-t).t^2+\vec{OM}_3.t^3$



Théorie mathématique générale

La construction itérative précédente peut être poursuivie au-delà du degré 3, jusqu'à un degré quelconque *n*. L'algorithme de De Casteljau permet d'obtenir une courbe de Bézier de degré *n*, possédant *n* + 1 points de contrôle.

Les polynômes de Bernstein de degré *n* sont alors les *n* + 1 termes du développement de $((1-t)+t)^n$

Soit M_0 , M_1 , ... M_n les *n* + 1 points de construction, l'équation de la courbe C s'écrit :

(4) $\vec{OM}(t)=\sum_{k=0}^{n} C_n^k . t^k . (1-t)^{n-k} . \vec{OM}_k$ pour mémoire : $C_n^k=\frac{n!}{k!(n-k)!}$ et $\begin{cases} n!=n.(n-1)! \\ 0!=1 \end{cases}$

8. Annexe 2 : Courbes B-Spline

Une courbe de Bézier est totalement modifiée dès qu'on déplace un point de contrôle : on dit que la méthode de Bézier est une méthode globale.

Les Courbes B-Splines Uniformes, issues des courbes de Bézier, ont été créés dans les années 80 pour remédier à cet inconvénient : le déplacement d'un point de contrôle de la courbe n'affecte ainsi plus qu'une partie limitée de la courbe, ce qui amène un plus grand confort dans la Conception Assistée par Ordinateur.

La méthode B-Spline est donc une méthode locale. Les B-Splines sont un cas très particulier de Courbes Splines, courbes paramétrées définissables de façon analytique, par morceaux, avec conditions imposées de régularité.

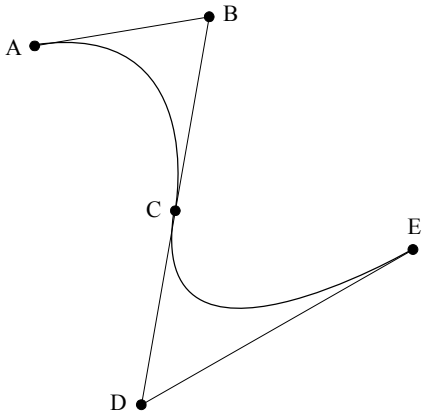
Principe de base

Une B-Spline Uniforme peut être sommairement définie comme une suite de courbes de Bézier accolées les unes aux autres de façon à ce que les raccordements entre elles soient suffisamment lisses aux points de raccord. On recherche la coïncidence de part et d'autre des vitesses (degré 2), voire des accélérations (degré 3).

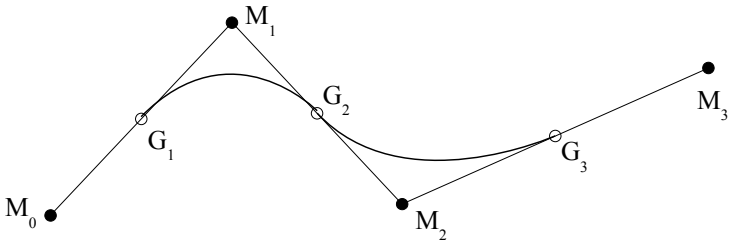
La figure ci-contre montre un exemple de B-Spline Uniforme de degré 2, construite à partir de deux courbes de Bézier elles aussi de degré 2.

La première respecte les points de construction A, B et C, tandis que la deuxième suit les points C, D et E.

Le raccordement est parfait (égalité des vitesses) lorsque le point C est le milieu de [BD].



Les points de contrôle des courbes de Bézier (suivant la définition de Pierre Bézier), deviennent dans ce contexte très difficile à manier : l'américain Farin a eu l'idée judicieuse de les redéfinir totalement afin que deux courbes de Bézier consécutives aient en commun tous les points de contrôle, sauf évidemment le premier de l'une et le dernier de l'autre…



Les nouveaux points de contrôle s'expriment comme des barycentres des points originels.

Cas particulier

La figure précédente illustre la construction d'une B-Spline de degré *m* = 2. En effet, deux segments sont nécessaires à la construction d'un premier arc de parabole.

Soit M_0 , M_1 , ... M_n les $n + 1$ points de construction, avec $2 \leqslant k \leqslant n$,

Toujours sur le même principe d'association des poids (1 – *t*) et (*t*) pour *t* variant de 0 à 1, l'équation paramétrique de la courbe C peut s'écrire :

(5)
$$\vec{OM}_k(t)=(\frac{1}{2}-t+\frac{t^2}{2}).\vec{OM}_{k-2}+(\frac{1}{2}+t-t^2).\vec{OM}_{k-1}+\frac{t^2}{2}.\vec{OM}_k$$

Cette équation respecte le modèle proposé par Richard Reisenfeld.

Généralisation

La formule précédente peut être généralisée pour une courbe B-Spline de degré *m* supérieur à 2.

Soit M_0 , M_1 , ... M_n les $n + 1$ points de construction, avec $m \leqslant n$ et $m \leqslant k \leqslant n$, on admet que l'équation de la courbe s'écrit :

(6)
$$\vec{OM}_k(t)=\sum_{i=0}^{m} R_{i,m}(t).\vec{OM}_{k+i}$$

où $R_{i,m}(t)$ représente la forme générale des polynômes de Reisenfeld :

(7)
$$R_{i,m}(t)=(m+1).\sum_{j=0}^{m-i}(-1)^j.\frac{(t+m-i-j)^m}{j!(m-j+1)!}$$

Remarque

Les points de contrôle ne comprennent pas les extrémités, pour imposer le passage par un point donné, il suffit de répéter ce point un nombre suffisant de fois (nombre lié au degré de la B-Spline).

La figure ci-dessous montre les étapes de construction d'une B-Spline de degré 2, la forme est parfaitement fermée par une répétition des deux premiers points de contrôle.