AUF CD
Die Quelltexte sowie die fertig übersetzten Routinen finden Sie im Verzeichnis *Praxis/Programmierung/PC Underground*.

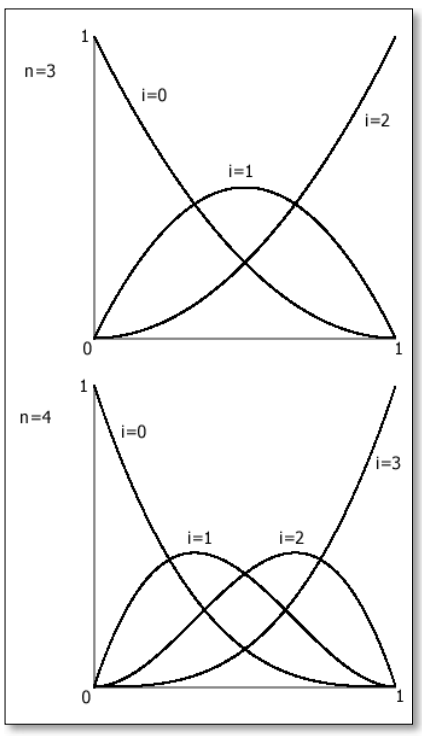
Bézier- und Spline-Kurven

Mathematische Reize

Wer als Anwender mit Bézier-Kurven harmonische Rundungen von Autoblechen am Rechner gestaltet, braucht keine höhere Mathematik. Programmierern bleibt sie nicht erspart.

CARSTEN DACHSBACHER

Die steigende Rechenleistung moderner CPUs und die Entwicklung hochleistungsfähiger 3D-Grafikkarten haben dazu geführt, dass professionelles Modelling Einzug in Computerspiele gehalten hat. Die Grundlage für Modelling sind parametrische (glatte, gekrümmte) Flächen. Eine parametrische Fläche legen Sie durch Basisfunktionen und Stütz-/Kontrollpunkte fest. Die Grundlagen für die Basisfunktionen und deren Auswertung lesen Sie in diesem Beitrag. Zunächst zeichnen Sie Kurven. Deren Form verändern Sie durch die Position der Stützpunkte. Mit diesem Handwerkszeug meistern Sie auch die Flächen im dreidimensionalen Raum. Für eine parametrische Kurve geben Sie – wie bei Flächen – eine Reihe von Basisfunktionen und Stützpunkten an. Die Bézier-Kurven sind die bekanntesten parametrischen Kurven. Sie wurden um 1960 entwickelt und in der französischen Automobilindustrie zum Karosseriedesign verwendet (Computer Aided Geometric Design, CAGD). Die Basisfunktionen, die Sie bei Bézier-Kurven verwenden, heißen Bernstein-Polynome.



BERNSTEIN-POLYNOME sind die Basis der Bézier-Kurven.

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

$$\text{mit } \binom{n}{i} = \frac{n!}{(n-k)!\,k!}$$

Diese Funktionen besitzen drei Variablen:

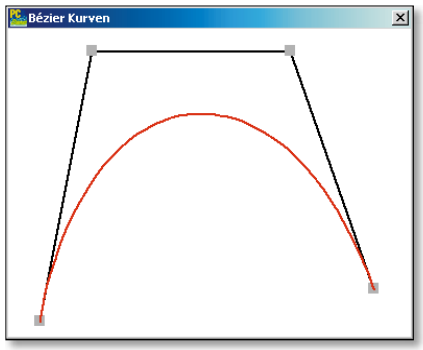
- *u* ist der Laufindex und nimmt Werte zwischen *0* und *1* an.
- *n* ist eine Ganzzahl und gibt den Grad der Kurve an. Das ist zum einen die höchste Potenz, in der die Laufvariable vorkommt, zum anderen bestimmen Sie dadurch die Zahl der Stützpunkte.
- Die Bézier-Kurve hat *(n+1)* Stützpunkte. Für verschiedene Indizes *i* erhalten Sie verschiedene Funktionen (abhängig von der Variablen *u*). Die Funktionswerte liegen im Intervall von [0,1]. Sie stellen die Gewichtung der einzelnen Stützpunkte dar, was auch in der Formel für Bézier-Kurven zu sehen ist.

$$F(u) = \sum_{i=0}^{n} B_i^n(u) \cdot b_i$$

Der Stützpunkt b_i wird mit dem Bernstein-Polynom *i* vom Grad *n* multipliziert. Alle Punkte, die Sie für *u* zwischen *0* und *1* erhalten, liegen auf der Bézier-Kurve. Nehmen Sie eine direkte Auswertung mit den Bernstein-Polynomen vor. Diese sieht wie folgt aus:

```
// Koordinate d des Punkts
// abhängig von u: d = F(u)
d.x = d.y = 0;
for ( i = 0; i< grad; i ++ )
{
    d = d + ( b[ i ] *
 bernstein( u, i ) );
}

...

// wertet Bernstein-Polynom aus
double bernstein
  (double u,long i)
{
return bin( grad, i ) *
pow( u, i ) *
pow( 1.0-u, grad-i );
}

// berechnet Fakultät von n
double  fac( long n )
{
    double  r = 1.0;
    for (i = 2; i <= n; i++ )
        r *= (double)i;
    return r;
}

// Binomialkoeffizient
double  bin(long n, long k )
{
return fac(n) /
 ( Fac(n-k) * Fac(k) );
}
```

Der Sourcecode *2dvector.c* zeigt eine definierte Vektorstruktur und überladene



EINE BÉZIER-KURVE vom Grad *n=3* und darüber das Kontrollpolygon als Linienzug zwischen den Kontrollpunkten

Operatoren-Anwendung. Bevor Sie die Bézier-Kurven genauer betrachten, verallgemeinern Sie die Formel zu einem beliebigen Intervall *[s,t]* für die Variable *u*:

$$B_i^{\Delta,n}(u) = \binom{n}{i}\left(\frac{u-s}{t-s}\right)^i\left(1-\frac{u-s}{t-s}\right)^{n-i}, \Delta = [s,t]$$

$$F(u) = \sum_{i=0}^{n} B_i^{\Delta,n}(u)\cdot b_i$$

Eigenschaften von Bézier-Kurven

Bézier-Kurven für *u* aus *[s,t]* liegen in der abgeschlossenen konvexen Hülle. Die konvexe Hülle einer Punktmenge können Sie so veranschaulichen, dass Sie mit einer gespannten Schnur versuchen, alle Punkte einzuschnüren. Weiterhin können Sie sehen, dass die Bézier-Kurve im ersten Stützpunkt *b0* beginnt und im letzten *b3* endet (Endpunkt-Interpolation).

Die Kurve endet nicht nur in den Endpunkten des Kontrollpolygons, sie verläuft dort auch tangentiell an den Kanten der Kontrollpolygone. Weiterhin sind Bézier-Kurven *affin invariant*: Bei einer affinen Transformation (eine Drehung und/oder eine Verschiebung) der Kontrollpunkte wird die Kurve mittransformiert, behält aber ihre Form.

Die Kurve schwankt nicht stärker als ihr Kontrollpolygon (*Variation-Diminishing-Property*, variationsreduzierend). Sie zeichnen Bézier-Kurven nicht punktweise, doch Sie werten die Bernstein-Polynome für jeden Punkt aus. Stattdessen approximieren Sie am Bildschirm die Kurve mit vielen Linien. Die Zahl der Linien hängt von der Größe der Kurve auf dem Bildschirm und der Auflösung ab. Die Linien können Sie schneller zeichnen als die einzelnen Pixel, deren Position Sie rechenintensiv auswerten müssten.

Der de-Casteljau-Algorithmus

Ein schnellerer Auswerte-Algorithmus – nicht für die Bernstein-Polynomen – für die Punkte auf Bézier-Kurven ist der de-Casteljau-Algorithmus. Er bestimmt die Koordinate eines Kurvenpunktes durch schrittweise Unterteilung des Kontrollpolygons.

Formal benötigen Sie folgende Definitionen, wobei Sie die Variablen wie folgt deuten können:

$$b_i^0 := b_i$$

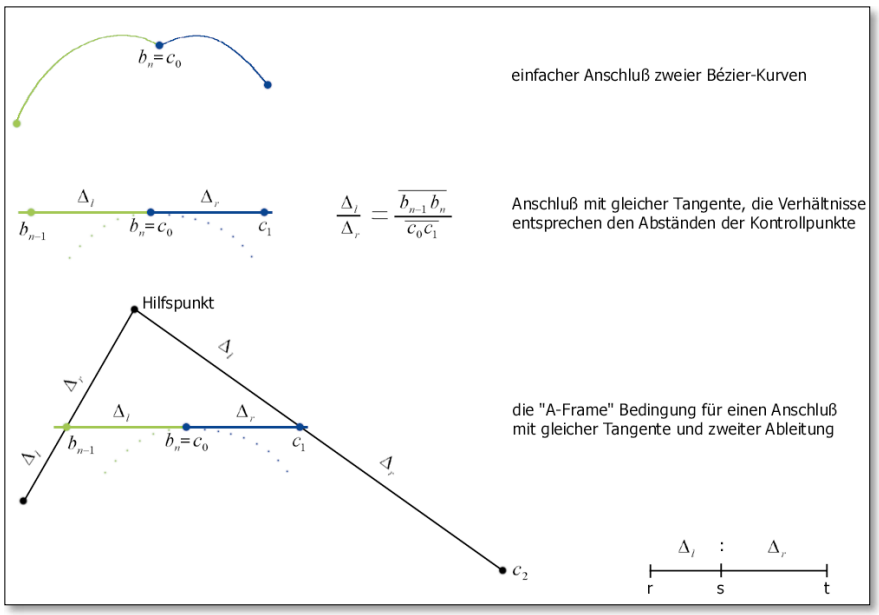
$$b_i^l = \alpha(u)\cdot b_{i+1}^{l-1} + (1-\alpha(u))\cdot b_i^{l-1}$$

$$\alpha(u) = \frac{u-s}{t-s}; l = 1,\ldots,n; i = 0,\ldots,n-l$$

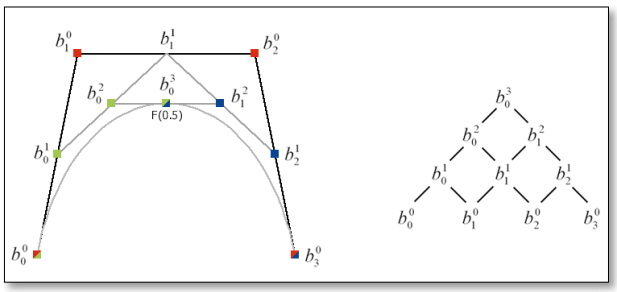
$$F(u) = b_0^n$$

ten, sind die Kontrollpunkte zweier neuer Bézier-Kurven, die zusammen die bisherige Kurve ergeben. Mit einem Unterschied: Die neuen Kontrollpolygone liegen näher an der tatsächlichen Bézier-Kurve. Wenn Sie also den de-Casteljau-Algorithmus rekursiv auf die neuen Bézier-Kurven anwenden, erhalten Sie Kontrollpolygone (Linienzüge), mit denen Sie die Bézier-Kurve zeichnen. Der de-Casteljau-Algorithmus lässt sich effizient implementieren, wie Sie dem Codeausschnitt im Quellcode entnehmen. Dieser zeigt eine Mittelpunktsunterteilung (alpha=0.5).

Wenn Sie Flächen mit vielen Details modellieren wollen, müssen Sie Bézier-Kurven mit einem hohen Grad *n* verwenden. Ändern Sie den Ort eines Kontrollpunkts, ändern Sie damit die ganze Kurve. Das umgehen Sie, indem Sie mehrere Bézier-Kurven von niedrigerem Grad (zum Beispiel kubisch, *n=3*) aneinanderhängen. Die Flächen lassen sich leicht lückenlos aneinander fügen,



VERSCHIEDENE ÜBERGÄNGE zweier Bézier-Kurven und geometrische Übergangsbedingungen perfektionieren die Kurven.



DER DE-CASTELJAU-ALGORITHMUS wertet Bézier-Kurven aus und teilt sie in diesem Beispiel.

Den eigentlichen Clou beim de-Casteljau-Algorithmus mit dem Ziel, die Kurve schnell mit Linien zu approximieren, sehen Sie im rechten Teil des Bildes: Die Punkte, die Sie als Zwischenergebnis am Rand der de-Casteljau-Pyramide erhalda die Kurven am Endpunkt interpolierend sind. Entscheidend für die Darstellung ist auch die Steigung und Krümmung der Kurven an den Anschlussstellen.

An einer Anschlussstelle entscheidet sich, ob Sie einen unerwünschten Knick erhalten. Im Automobilbau gibt es eine weitere Anforderung: Die Kurven müssen am Anschlusspunkt auch in der zweiten Ableitung gleich sein. Sonst ist der Übergang bei Reflexionen, zum

Beispiel auf Autolacken, sichtbar. Im unteren Teil des rechten Bildes auf der vorigen Seite sehen Sie die geometrischen Bedingungen, die zwei Bézier-Kurven erfüllen müssen, um den entsprechenden Anforderungen zu genügen. Trotz der etwas umständlichen Beschreibung detailreicher Flächen haben sie aber trotzdem eine Existenzberechtigung: Rechner werten Bézier-Kurven effizient und in Echtzeit aus. Damit haben Bézier-Flächen die Eigenschaften, die für Echtzeit-Rendering von Vorteil sind.

B-Spline-Kurven

B-Spline-Kurven sind eine neue Gattung mathematischer Kurvenbeschreibungen. Wir beschäftigen uns mit B-Spline-Kurven, die die Eigenschaft der affinen Invarianz (Begriff: siehe oben) mitbringen. Die Definition eine B-Spline-Kurve lautet:

$$F(u) = \sum_{i=0}^{n} N_i^n(u) \cdot d_i$$

Die Stützpunkte bezeichnen Sie mit d_i (de-Boor-Punkte, nach Carl de Boor). Zusätzlich gibt es einen Knotenvektor t, dessen Werte sich in den rekursiv definierten B-Spline-Basisfunktionen niederschlagen:

$$t_i \le u < t_{i+1} \Rightarrow N_i^0(u) = 1$$

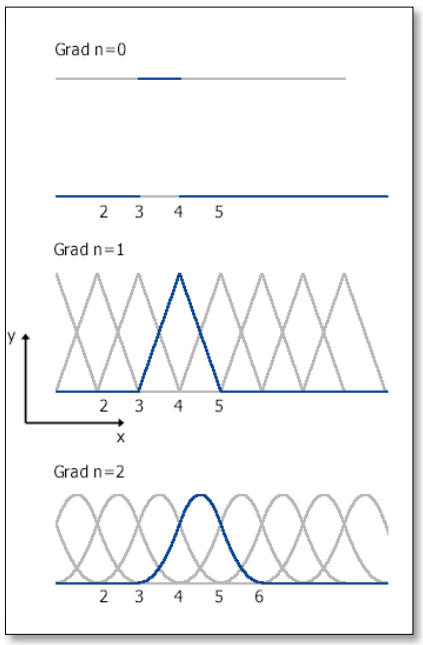
$$sonst \quad N_i^0(u) = 0$$

$$N_i^l(u) = \frac{u - t_i}{t_{i+l} - t_i} N_i^{l-1}(u) + \frac{t_{i+l+1} - u}{t_{i+l+1} - t_{i+1}} N_{i+1}^{l-1}(u),$$

$$l = 1,\ldots,n$$

Im Bild oben rechts sehen Sie Basisfunktionen vom Grad *0* bis *2*. Daran können Sie einen Vorteil gegenüber den Bernstein-Polynomen als Basisfunktionen ablesen: Die B-Spline-Funktionen sind nur in einem begrenzten Bereich ungleich Null. Bernstein-Polynome sind im gesamten Bereich, in dem sich die Laufvariable *u* befindet, ungleich Null. Dies ist gleichbedeutend damit, dass ein Kontrollpunkt nur auf einem sehr begrenzten Bereich der Kurve Einfluss ausübt. Damit können Sie an bestimmten Teilen eine B-Spline-Kurve detailreich modellieren, ohne die Kurve zu ändern.

B-Spline-Basisfunktionen vom Grad *n* sind stückweise polynomiell (durch Polynome beschreibbar) und bieten deshalb optimale Glattheit. Dadurch werden die geometrischen Übergangsbedingungen überflüssig. Um ein Gefühl für



DIE B-SPLINE-BASISFUNKTIONEN sind nur in kleinen Bereichen von Null verschieden.

die Auswirkungen des Knotenvektors auf die Kurve zu bekommen, experimentieren Sie am besten mit unserem Beispielprogramm. Der Knotenvektor hat so viele Werte wie Grad *n* plus Anzahl der Stützpunkte plus 2. Der Knotenvektor beeinflusst den Verlauf der Kurve innerhalb der konvexen Hülle des Kontrollpolygons. B-Spline-Kurven sind zum Beispiel nur Endpunktinterpolierend, wenn jeweils die ersten *(n+1)* und die letzten *(n+1)* Werte des Knotenvektors gleich sind.

Die direkte Auswertung der B-Splines können Sie mit folgendem Codeausschnitt berechnen. Beachten Sie die Spezialfälle für den Knotenvektor bei der Rekursion im Listing *bspline.c*.

Betrachten Sie eine B-Spline-Kurve vom Grad *n* mit *m* de-Boor-Punkten und einem Knotenvektor *t*. Nutzen Sie die folgenden Eigenschaften, um Kurven gezielt zu modellieren:

- Fallen *n* de-Boor-Punkte zusammen (sind also identisch), so verläuft die Kurve durch diesen Punkt und liegt dort tangentiell an dem Kontrollpolygon an. Damit können Sie Ecken in der Kurve modellieren.
- Wenn Sie *n* de-Boor-Punkte auf einer Geraden platzieren, berührt die Kurve diese Gerade. Wenn sich *(n+1)* Punkte auf einer Gerade befinden, liegt ein Abschnitt der Kurve auf dieser Geraden.
- Fallen *n* Knoten (Werte im Knotenvektor) zusammen, also $t=t_i+1=\ldots=t_{i+n}$, so gilt $F(t)=d_i$. Das heißt, dass die Kurve durch einen Kontrollpunkt verläuft und dort tangentiell am Kontrollpolygon anliegt.
- Als letzte Eigenschaft können Sie die „lokale konvexe Hülle“ ausnutzen. Für ein *u* im Intervall $[t_i, t_{i+1}]$ liegt die Kurve in der abgeschlossenen konvexen Hülle der *(n+1)* vielen Kontrollpunkte $d_{i-n}, \ldots, d_i$.

Der de-Boor-Algorithmus

Auch für B-Spline-Kurven gibt es elegante Algorithmen zur Auswertung, die aber trotzdem rechenintensiver als die für Bézier-Kurven sind. Als Pendant zum de-Casteljau-Algorithmus gibt es für B-Spline-Kurven den – rekursiv definierten – de-Boor-Algorithmus. Seine Definition:

$$d_i^0 := d_i$$

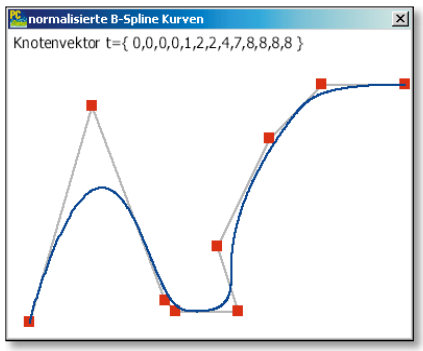
$$d_i^l = \left(1 - \frac{u - t_i}{t_{i+n+1-l} - t_i}\right) \cdot d_{i-1}^{l-1} + \left(\frac{u - t_i}{t_{i+n+1} - t_i}\right) \cdot d_i^{l-1}$$

$$l = 1,\ldots,n; i = j - n + j,\ldots,j$$

$$F(u) = d_j^n$$

Um ihn anschaulich darzustellen, bedarf es einer anderen Darstellung der B-Spline-Kurve, der so genannten Polarform.

Laut Definition gilt der de-Boor-Algorithmus nur für Parameter aus dem Intervall $[t_j, t_{j+1}]$. Folgender Programm-



EINE NORMALISIERTE B-Spline-Kurve und ihr Knotenvektor

code berechnet den de-Boor-Algorithmus für die Kurve an der Stelle *u* vom Grad *l* im Intervall *[t(i), t(i+1)]*:

```
//VECTOR deBoor
➥ (double u,long l,long i)
    {
        if ( l == 0 )
// letzte ausgewertete Stelle
// im letzten Intervall !
 if ( i == nKontrollPunkte )
  return  d[ nKontrollPunkte-1];
  else
  return  d[ i ];

  double  t2=( u - t [i])/
```

```
(t [i + grad + 1 - l] - t[i]);
      double  t1 =1.0 - t2;

 return  deBoor( u, l-1, i-1)
  *t1 + deBoor(u,l-1,i) * t2;
  }
```

Wenn Sie eine Spline-Kurve an der Anzahl *steps* Stellen pro Intervall des Knotenvektors auswerten wollen, verwenden Sie folgenden Code:

```
//Speicher ausreichend Punkte
   VECTOR result[ GENUGPUNKTE ];
    int    nPunkte = 0;

// Kurvengrad n, „steps“-Stellen
// „nIntervals“ im Knotenvektor
    for (i = n;
    i< nIntervals +n;i++)
    {
// Vorraussetzung im de Boor Al-
gorithmus
        if( t[i + 1 ] > t[ i ] )
        {
    for( j = 0; j <= steps; j++)
      {
       double u=t[i]+(double)j*
 ( t[i+1]- t[i])/(double)steps;

    result[ nPunkte++ ] =
         deBoor(u,grad,i);
            }
         }
     }
```

ET

Literatur:

Gerald Farin: Curvers and Surfaces for Computer Aided Geometric Design, Academic Press, ISBN 0-12-249054-1

1 2dvector.c

```
 1: //2dvector.c: 2D Vektor Struktur
 2: typedef struct {
 3:     double  x, y;
 4: }VECTOR;
 5:
 6: // Vektor + Vektor
 7: VECTOR  operator + ( const VECTOR &a, const VECTOR &b ) {
 8:     VECTOR result;
 9:     result.x = a.x + b.x;
10:     result.y = a.y + b.y;
11:     return result;
12: }
13:
14: // Vektor * Skalar
15: VECTOR  operator * ( const VECTOR &a, const double b ) {
16:     VECTOR result;
17:     result.x = a.x * b;
18:     result.y = a.y * b;
19:     return result;
20: }
21:
22: // Berechnet eine de Casteljau Pyramide mit alpha(u)=0.5
23: void  calcPyramide( long n, VECTOR *Pyramide )
24: {
25:   long  i, j;
26:
27:   // Ebene
28:   for ( i = 1; i < n; i++ )
29:     {
30:       // Spalte
31:       for ( j = 0; j < n - i; j++ )
32:     {
33:       Pyramide[ j + i * n ] =
34:           (Pyramide[j+(i-1)*n] + Pyramide[(j+1)+(i-1)*n])*0.5;
35:     }
36:     }
37: }
38:
39: // Berechnet rekursiv de Casteljau
40: void  DeCasteljauRekursiv
41: ( long rek, long n, long k, VECTOR *data )
42: {
43:   long  i;
44:   VECTOR  *Pyramide = new VECTOR[ n * n ];
45:
46:   // unterste Pyramidenebene mit
47:         // Kontrollpunkten der Bezierkurve fullen
48:   for ( i = 0; i < n; i++ )
49:       Pyramide[ i + 0 * n ] = data[ i ];
50:
51:   // Pyramide berechnen
52:   CalcPyramide( n, Pyramide );
53:
54:   if ( rek == k )
55:   {
56:     // Rekursionstiefe erreicht => Punkte speichern
57:     for ( i = 0; i < n; i++ )
58:       Points[ nPoints ++ ] = Pyramide[ 0 + i * n ];
59:     for ( i = 0; i < n; i++ )
60:      Points[ nPoints ++ ] = Pyramide[( n - i - 1 ) + i * n];
61:   } else
62:   {
63:     // weiter unterteilen
64:     VECTOR  *data = new VECTOR[ n ];
65:
66:     for ( i = 0; i < n; i++ )
67:       data[ i ] = Pyramide[ 0 + i * n ];
68:
69:     DeCasteljauRekursiv( rek + 1, n, k, data );
70:
71:     for ( i = 0; i < n; i++ )
72:       data[ i ] = Pyramide[ ( n - i - 1 ) + i * n ];
73:
74:     DeCasteljauRekursiv( rek + 1, n, k, data );
75:
76:     delete data;
77:   }
78:
79:   delete Pyramide;
80: }
81:
82: void  DeCasteljau( long n, long k, VECTOR *data,
83: VECTOR **result, long *nP, long *nBeziers )
84: {
85:   // soviele Bezierkuven werden wir haben
86:   *nBeziers = 1 << k;
87:   // und soviele Punkte
88:   *nP = *nBeziers * (n+1);
89:
90:   // Speicher dafur allokieren
91:   nPoints = 0;
92:   Points = new VECTOR[ *nP ];
93:
94:   DeCasteljauRekursiv( 1, n + 1, k, data );
95:
96:   *result = Points;
97: }
```

***2dvector.c* zeigt eine definierte Vektorstruktur und eine überladene Operatoren-Anwendung.**

2 bspline.c

```
 1: //bspline.c
 2: double  bspline
 3: ( long i, long l, double u )
 4:     {
 5:         if ( l == 0 )
 6:         {
 7:        // Rekursionstiefe 0
 8:        if ( T( i ) <=
 9:   u && u < T( i + 1 ) )
10:    return 1.0; else return 0.0;
11:    } else
12:    {
13:    // hoehere Rekursionstiefen
14:     double a, b;
15:
16:     a = Nb( i, l - 1, u );
17:     b = Nb( i + 1, l - 1, u );
18:
19:     double d1=( T(i+l)-T(i) );
20:     double d2 =(T(i+l+1)-T(i+1) );
21:
22:     double q1=(u-T(i));
23:     double q2=(T(i+l+1)-u);
24:
25:     // Division durch 0 abfangen
26:     if (d1 == 0.0) d1=q1=1.0;
27:     if (d2 == 0.0) d2=q2=1.0;
28:
29:     if (q1==0.0 && d1==0.0) q1 =1.0;
30:             else q1 /= d1;
31:     if (q2==0.0 && d2==0.0) q2 =1.0;
32:             else q2 /=d2;
33:
34:     return  q1 * a + q2 * b;
35:         }
36:     }
37:
38:     ...
39:
40:     // Auswertung der Kurve mit
41:     // m Stützpunkten d[]: f = F(u)
42:     f.x = f.y = 0;
43:
44:     for ( i= 0; i <= m;i++)
45:         f=f+d[i]*bspline(i,grad,u);
```

Der Ausschnitt *bspline.c* wertet B-Splines aus.