University of Houston Downtown

Department of Computer and Mathematical Sciences

Recursive Evaluation Algorithms for 2-D h-Bézier Surfaces

A Senior Project
by
Wilfredo Molina

Faculty Advisor:
Dr. Plamen Simeonov, CMS

December 1, 2012

h-Bernstein Polynomials

An h-Bernstein basis polynomial of degree n has the following form[1][5]:

$$B_k^n(t;h) = \binom{n}{k} \frac{\prod_{i=0}^{k-1}(t+ih)\prod_{i=0}^{n-k-1}(1-t+ih)}{\prod_{i=0}^{n-1}(1+ih)}, \tag{1}$$

where $k = 0, 1, \ldots, n$, and $h \in [0, 1]$ is a fixed parameter, which is called a shape parameter. In equation (1),

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

are the familiar binomial coefficients. The h-Bernstein polynomials defined by equation (1) can be generated by the following recursive relation[1][5]:

$$B_k^n(t;h) = \frac{t+(k-1)h}{1+(n-1)h} B_{k-1}^{n-1}(t;h) + \frac{1+t+(n-k-1)h}{1+(n-1)h} B_k^{n-1}(t;h), \tag{2}$$

where $k = 0, 1, \ldots, n$, and $B_0^0(t;h) = 1$. $B_k^n(t;h)$ is defined to be equal to zero whenever $k < 0$ or $k > n$.

A linear combination of h-Bernstein basis polynomials

$$B_n(t) = \sum_{k=0}^{n} \beta_k B_k^n(t;h) \tag{3}$$

is called an h-Bernstein polynomial of degree n, and the coefficients β_n are called h-Bernstein coefficients or h-Bézier coefficients.

If we choose in equation (3), $\beta_k = f(k/n)$, $k = 0, \ldots, n$, where $f(t)$ is a continuous function of t on $[0, 1]$, then $B_n(t) \to f(t)$ as $n \to \infty$ uniformly on $[0, 1]$. This is a well-known result due to Stancu[7].

The set of $n+1$ h-Bernstein basis polynomials $\{B_k^n(t;h)\}_{k=0}^n$ defines a discrete probability distribution on the interval $[0, 1]$ because $B_k^n(t,h) \geqslant 0$ for $t \in [0, 1]$, and $k = 0, 1, \ldots, n$. Hence, the partition of unity property holds[5], that is,

$$\sum_{k=0}^{n} B_k^n(t;h) = 1. \tag{4}$$

h-Bézier Curves

For every polynomial $P(t)$ of degree n, there is a unique set of $n+1$ coefficients $\{P_k\}_{k=0}^n$ called control points, such that

$$P(t) = \sum_{k=0}^{n} P_k B_k^n(t;h). \tag{5}$$

A polynomial $P(t)$ written in the form of equation (5) is called an h-Bézier curve of degree n. Furthermore, observe that equation (5) is equivalent to equation (3) with $\beta_k = P_k$. Thus, an h-Bézier curve is a special case of an h-Bernstein polynomial.

The h-de Casteljau Recursive Evaluation Algorithm

The h-de Casteljau recursive evaluation algorithm for h-Bézier curves is described as follows:

We let

$$P(t) = \sum_{k=0}^{n} P_k B_k^n(t;h)$$

be an h-Bézier curve. Set $P_k^0 = P_k$, for $k = 0, 1, \ldots, n$. Then, define recursively:

$$P_k^r(t) = \frac{1 - t + (n - r - k)h}{1 + (n - r)h} P_k^{r-1}(t) + \frac{t + kh}{1 + (n - r)h} P_{k+1}^{r-1}(t), \tag{6}$$

where $k = 0, 1, \ldots, n - r$ and $r = 1, 2, \ldots, n$. At level n, we get a single function $P_0^n(t) = P(t)$.

This result follows from equation (2) and induction in r[1][5].

2-D h-Bernstein Polynomials

We define a two-dimensional (or two-variable) version of the h-Bernstein basis polynomials by setting

$$B_{k,l}^n(t, s; h) = B_k^n(t; h) B_l^n(s; h), \tag{7}$$

where $k, l = 0, \ldots, n$. Hence, by equations (2) and (7), a 2-D version of the recursive relation (2) can be derived as follows:

$$\begin{aligned} B_{k,l}^n(t, s; h) = &\left(\frac{t + (k-1)h}{1 + (n-1)h} B_{k-1}^{n-1}(t; h) + \frac{1 - t + (n - k - 1)h}{1 + (n-1)h} B_k^{n-1}(t; h) \right) \\ &\times \left(\frac{s + (l-1)h}{1 + (n-1)h} B_{l-1}^{n-1}(s; h) + \frac{1 - s + (n - l - 1)h}{1 + (n-1)h} B_l^{n-1}(s; h) \right), \end{aligned} \tag{8}$$

where $k, l = 0, 1, \ldots, n$. To simplify this expression, we set

$$\alpha_{n,k}(t; h) = \frac{t + (k-1)h}{1 + (n-1)h}, \qquad k = 1, \ldots, n. \tag{9}$$

Then equation (8) becomes:

$$\begin{aligned} B^n_{k,l}(t,s;h) =& \alpha_{n,k}(t;h)\alpha_{n,l}(s;h)B^{n-1}_{k-1,l-1}(t,s;h) \\ &+\alpha_{n,k}(t;h)\alpha_{n,n-l}(1-s;h)B^{n-1}_{k-1,l}(t,s;h) \\ &+\alpha_{n,n-k}(1-t;h)\alpha_{n,l}(s;h)B^{n-1}_{k,l-1}(t,s;h) \\ &+\alpha_{n,n-k}(1-t;h)\alpha_{n,n-l}(1-s;h)B^{n-1}_{k,l}(t,s;h). \end{aligned} \tag{10}$$

2-D h-Bézier Surfaces

Similar to their 1-D counterparts, for every polynomial $P(t,s)$ of degree at most n in each variable, there is a unique set of $(n+1)^2$ coefficients $\{P_{k,l}\}^n_{k,l=0}$, also called control points, such that:

$$P(t,s) = \sum_{k,l=0}^{n} P_{k,l}B^n_{k,l}(t,s;h). \tag{11}$$

A polynomial $P(t,s)$ written in the form of equation (11) is called a 2-D h-Bézier surface.

It is easy to see that the partition of unity property described in equation (4) also holds for the 2-D h-Bernstein basis polynomials:

$$\sum_{k,l=0}^{n} B^n_{k,l}(t,s;h) = \sum_{k=0}^{n} B^n_k(t;h) \sum_{l=0}^{n} B^n_l(s;h) = 1.$$

Therefore, for each n, the $(n+1)^2$ 2-D h-Bernstein polynomials $\left\{B^n_{k,l}(t,s;h)\right\}^n_{k,l=0}$ form a discrete probability distribution on $[0,1]\times[0,1]$. Moreover, property (11) implies the affine invariance property of 2-D h-Bézier surfaces. That is,

$$\sum_{k,l=0}^{n} \left(P_{k,l}+v\right) B^n_{k,l}(t,s;h) = P(t,s)+v. \tag{12}$$

This means that if all control point $P_{k,l}$ are shifted by a constant vector v, then the entire 2-D h-Bézier surface is also shifted by v.

The 2-D h-de Casteljau Recursive Evaluation Algorithm

The h-de Casteljau recursive evaluation algorithm for 2-D h-Bézier surfaces is defined as follows:

We let

$$P(t,s) = \sum_{k,l=0}^{n} P_{k,l} B_{k,l}^{n}(t,s;h)$$

be a 2-D h-Bézier surface. Set $P_{k,l}^{0} = P_{k,l}$, for $k,l = 0,1,\ldots,n$. Assume that for some $0 \leqslant r < n$, all the points at the r-th level $P_{k,l}^{r}$, for $k,l = 0,1,\ldots,n-r$, have been defined so that

$$P(t,s) = \sum_{k,l=0}^{n-r} P_{k,l}^{r} B_{k,l}^{n-r}(t,s;h). \tag{13}$$

To compute the points at the next level $r+1$, substitute for $B_{k,l}^{n-r}(t,s;h)$ in (13) by the expression from equation (10). After this substitution, equation (13) becomes

$$\begin{aligned}
P(t,s) = \sum_{k,l=0}^{n-r} P_{k,l}^{r} \big[& \alpha_{n-r,k}(t;h)\alpha_{n-r,l}(s;h) B_{k-1,l-1}^{n-r-1}(t,s;h) \\
& + \alpha_{n-r,k}(t;h)\alpha_{n-r,n-r-l}(1-s;h) B_{k-1,l}^{n-r-1}(t,s;h) \\
& + \alpha_{n-r,n-r-k}(1-t;h)\alpha_{n-r,l}(s;h) B_{k,l-1}^{n-r-1}(t,s;h) \\
& + \alpha_{n-r,n-r-k}(1-t;h)\alpha_{n-r,n-r-l}(1-s;h) B_{k,l}^{n-r-1}(t,s;h) \big] \\
= \sum_{k,l=0}^{n-r-1} & P_{k,l}^{r+1} B_{k,l}^{n-r-1}(t,s;h).
\end{aligned} \tag{14}$$

Therefore, by collecting the coefficients of $B_{k,l}^{n-r-1}(t,s;h)$ in equation (13), we see that the control points at level $r+1$ are given by

$$\begin{aligned}
P_{k,l}^{r+1} &= P_{k+1,l+1}^{r} \alpha_{n-r,k+1}(t;h)\alpha_{n-r,l+1}(s;h) \\
&+ P_{k+1,l}^{r} \alpha_{n-r,k+1}(t;h)\alpha_{n-r,n-r+1-l}(1-s;h) \\
&+ P_{k,l+1}^{r} \alpha_{n-r,n-r-k}(1-t;h)\alpha_{n-r,l+1}(s;h) \\
&+ P_{k,l}^{r} \alpha_{n-r,n-r-k}(1-t;h)\alpha_{n-r,n-r-l}(1-s;h),
\end{aligned} \tag{15}$$

for $k,l = 0,1,\ldots,n-r-1$ and $r = 0,1,\ldots,n-1$.

Clearly, each point at the r-th level $P_{k,l}^{r} = P_{k,l}^{r}(t,s)$ is a polynomial of t and s of degree r in each variable. At the top level, that is, $r = n$, we get a single point which is $P_{0,0}^{n}(t,s) = P(t,s)$. So this last point gives the value of the point $z = P(t,s)$ on the 2-D h-Bézier surface. To run the n levels of the recursive evaluation algorithm defined by equation (14), we have to perform a total of $(1/6)n(n+1)(2n+1)$ linear operations of this type. This is so because

$$\sum_{r=1}^{n} (n-r+1)^2 = \frac{1}{6} n(n+1)(2n+1),$$

where $(n-r+1)^2$ is the number of operations of type (15) used to compute the $(n-r+1)^2$ control points at level r.

C++ Implementation

```
// Client File: main.cpp

// This program creates a 2-D h-Bezier surface. It
// prompts the user for a perfect-square number of points
// and creates a Maple 15 file ready for compilation.

//    Wilfredo Molina, 11/30/12.

#include <iostream>
#include <fstream>
#include <cmath>
#include "Point.h"
#include "HelperFunctions.h"
#include "MainFunctions.h"

using namespace std;

int main()
{
   int n, N, cnt = 0;
   char c;
   double h;
   Point **set, p;
   ifstream init;
   ofstream file;

   cout << "********************************" << endl;
   cout << "* 2-D h-Bezier Surface Plotter *" << endl;
   cout << "********************************" << endl << endl;
   cout << "Enter h: ";
   cin >> h;
   cout << "Enter the size of the input: ";
   cin >> n;

   // Create and define a 2-D dynamic array of points.
   N = (int)sqrt((double)n);

   set = new Point*[N];

   for (int i = 0; i < N; i++)
```

```
      set[i] = new Point[N];

   for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++) {
         cout << "Enter point " << ++cnt << " of " << n << ": ";
         cin >> set[i][j].x >> set[i][j].y >> set[i][j].z;
      }

   file.open("output.mw");

   // Initialize the Maple 15 file.
   init.open("init.txt");
   init.get(c);
   while (init) {
      file << c;
      init.get(c);
   }
   init.close();

   // Plot the surface.
   file << "with(plots):</Text-field><Text-field prompt=\"&gt; \" ";
   file << "style=\"Maple Input\" layout=\"Normal\">pointplot3d({";
   for (double t = 0; LessThanOrEqualTo(t, 1); t += 0.025)
      for (double s = 0; LessThanOrEqualTo(s, 1); s += 0.025) {
         p = f(set, 0, 0, N - 1, N - 1, t, s, h);
         file << '[' << p.x << ',' << p.y << ',' << p.z << ']';
         if (LessThanOrEqualTo(t + 0.025, 1) ||
             LessThanOrEqualTo(s + 0.025, 1))
            file << ',';
      }
   file << "},axes=boxed);</Text-field></Group></Worksheet>";
   file.close();
   cout << endl << "The process has finished.";

   // Delete the dynamic array.
   for (int i = 0; i < N; i++)
      delete [] set[i];
   delete [] set;

   cin.get();
   return 0;
}
```

```
// Specification File: Point.h

// This structure defines the points used by the algorithms.

//    Wilfredo Molina, 11/30/12.

#ifndef POINT_H
#define POINT_H

struct Point {
   double x, y, z;                  // Coordinates

   Point();                         // Default Constructor
   Point(double, double, double);   // Constructor
   Point operator*(double);         // Operator this * other Overload
   Point operator+(Point);          // Operator this + other Overload
};

#endif // POINT_H
```

```
// Implementation File: Point.cpp

//    Wilfredo Molina, 11/30/12.

#include "Point.h"

Point::Point()
{
   x = y = z = 0;
}

Point::Point(double x, double y, double z)
{
   this->x = x;
   this->y = y;
   this->z = z;
}

Point Point::operator*(double rhs)
{
   return Point(this->x * rhs, this->y * rhs, this->z * rhs);
}

Point Point::operator+(Point rhs)
{
   return Point(this->x + rhs.x, this->y + rhs.y, this->z + rhs.z);
}
```

```
// Specification File: HelperFunctions.h

// This file contains the very low-level helper
// functions used by the main functions.

//    Wilfredo Molina, 11/30/12.

#ifndef HELPERFUNCTIONS_H
#define HELPERFUNCTIONS_H

#define EPSILON 0.0001

bool LessThanOrEqualTo(double, double);   // Compares Doubles
double alpha(int, int, double, double);   // Theoretical Function
double prod(int, int, double, double);    // Particular Product Notation
int fact(int);                            // Factorial
int choo(int, int);                       // n Choose k

#endif // HELPERFUNCTIONS_H
```

```
// Implementation File: HelperFunctions.cpp

//    Wilfredo Molina, 11/30/12.

#include "HelperFunctions.h"

bool LessThanOrEqualTo(double a, double b)
{
   double diff = a - b;
   return diff <= EPSILON;
}

double alpha(int k, int n, double t, double h)
{
   return (t + (k - 1) * h) / (1 + (n - 1) * h);
}

double prod(int i, int n, double t, double h)
{
   if (i > n)
      return 1;
   double q = 1;
   for (int j = i; j <= n; j++)
      q *= t + j * h;
   return q;
}

int fact(int n)
{
   if (n == 0)
      return 1;
   if (n <= 2)
      return n;
   return n * fact(n - 1);
}

int choo(int n, int k)
{
   return fact(n) / (fact(k) * fact(n - k));
}
```

```
// Specification File: MainFunctions.h

// This file contains the main functions
// used by the program to compute surfaces.

//    Wilfredo Molina, 11/30/12.

#ifndef MAINFUNCTIONS_H
#define MAINFUNCTIONS_H

#include "Point.h"

// Non-Recursive h-Bernstein Polynomial
double B(int, int, double, double);

// Recursive h-Bernstein Polynomial
double BR(int, int, double, double);

// 1-D h-de Casteljau Recursive Evaluation Algorithm
double PR(int, int, int, double, double, double*);

// 2-D h-de Casteljau Recursive Evaluation Algorithm
Point f(Point**, int, int, int, int, double, double, double);

#endif // MAINFUNCTIONS_H
```

```
// Implementation File: MainFunctions.cpp

//    Wilfredo Molina, 11/30/12.

#include "MainFunctions.h"
#include "HelperFunctions.h"

double B(int k, int n, double t, double h)
{
   return choo(n, k) * (prod(0, k - 1, t, h)
        * prod(0, n - k - 1, 1 - t, h))
        / prod(0, n - 1, 1, h);
}

double BR(int k, int n, double t, double h)
{
   if (k < 0 || k > n)
      return 0;
   if (k == 0 && n == 0)
      return 1;
   return (t + (k - 1) * h) / (1 + (n - 1) * h)
        * BR(k - 1, n - 1, t, h) + (1 - t + (n - k - 1) * h)
        / (1 + (n - 1) * h) * BR(k, n - 1, t, h);
}

double PR(int k, int r, int n, double t, double h, double *I)
{
   if (r == 0)
      return I[k];
   else
      return (1 - t + (n - r - k) * h) / (1 + (n - r) * h)
           * PR(k, r - 1, n, t, h, I) + (t + k * h)
           / (1 + (n - r) * h) * PR(k + 1, r - 1, n, t, h, I);
}

Point f(Point **set, int k, int l, int r,
        int n, double t, double s, double h) {
   if (r == 0)
      return set[k][l];
   else
      return f(set, k + 1, l + 1, r - 1, n, t, s, h)
           * alpha(k + 1, n - r + 1, t, h)
```

```
            * alpha(l + 1, n - r + 1, s, h)
            + f(set, k + 1, l, r - 1, n, t, s, h)
            * alpha(k + 1, n - r + 1, t, h)
            * alpha(n - r + 1 - l, n - r + 1, 1 - s, h)
            + f(set, k, l + 1, r - 1, n, t, s, h)
            * alpha(n - r + 1 - k, n - r + 1, 1 - t, h)
            * alpha(l + 1, n - r + 1, s, h)
            + f(set, k, l, r - 1, n, t, s, h)
            * alpha(n - r + 1 - k, n - r + 1, 1 - t, h)
            * alpha(n - r + 1 - l, n - r + 1, 1 - s, h);
}
```

Maple 15 Implementation

```
restart;              # Clears the system.

with(VectorCalculus): # Used to create 3-D points.
with(plots):          # Used to animate surface.

P := [[<0, 2, -1>, <1, 2, 1>, <2, 2, -1>], # User-Defined Points
      [<0, 1, 1>, <1, 1, -10>, <2, 1, 1>],
      [<0, 0, -1>, <1, 0, 1>, <2, 0, -1>]]:

alpha :=                                  # Alpha Function
proc (k, n, t, h)
   options operator, arrow; (t + (k - 1) * h) / (1 + (n - 1) * h)
end proc:

# h-de Casteljau Recursive Evaluation Algorithm
f :=
proc (k, l, r, n, t, s, h)
   if r = 1 then
      return P[k, l]
   else
      return f(k + 1, l + 1, r - 1, n, t, s, h)
           * alpha(k, n - r + 1, t, h)
           * alpha(l, n - r + 1, s, h)
           + f(k + 1, l, r - 1, n, t, s, h)
           * alpha(k, n - r + 1, t, h)
           * alpha(n - r + 2 - l, n - r + 1, 1 - s, h)
           + f(k, l + 1, r - 1, n, t, s, h)
           * alpha(n - r - k + 2, n - r + 1, 1 - t, h)
           * alpha(l, n - r + 1, s, h)
           + f(k, l, r - 1, n, t, s, h)
           * alpha(n - r - k + 2, n - r + 1, 1 - t, h)
           * alpha(n - r + 2 - l, n - r + 1, 1 - s, h)
   end if
end proc:

g := simplify(f(1, 1, 3, 3, t, s, h)); # Evaluate the surface.

# Plot and animate the surface.
animate(plot3d, [g, t = 0 .. 1, s = 0 .. 1], h = 0 .. 1, axes = boxed);
```

List of References

[1] Goldman, R., 1985. Pólya's urn model and computer aided geometric design. SIAM J. Alg. Disc. Meth. 6, 1-28

[2] Goldman, R., 2003. Pyramid Algorithms: A Dynamic Programming Approach to Curves and Surfaces for Geometric Modeling, The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Elsevier Science.

[3] Goldman, R., Barry, P., 1991. Shape parameter deletion for Pólya curves. Numer. Algorithms 1, 121-137.

[4] Phillips, G.M., 1997a. A de Casteljau algorithm for generalized Bernstein polynomials. BIT 37, 232-236.

[5] Simeonov, P., Zafiris, V., Goldman, R., 2011. h-Blossoming: A New Approach to Algorithms and Identities for h-Bernstein Bases and h-Bézier Curves.

[6] Simeonov, P., Zafiris, V., Goldman, R., 2010. q-Blossoming: A new approach to algorithms and identities for q-Bernstein bases and q-Bézier curves. J. Approx. Theory, in press, doi:10.1016/j.jat.2011.09.006.

[7] Stancu, D., 1968. Approximation of functions by a new class of linear polynomial operators. Rev. Roumaine Math. Pures Appl. 13, 1173-1194.