Computing Values and Derivatives of Bézier and B-spline Tensor Products

Computer Science Department
University of Waterloo
Research Report CS-93-31

Stephen Mann
University of Waterloo
Tony DeRose, Georges Winkenbach
University of Washington

Abstract

When evaluating tensor product surfaces it is often necessary to calculate both the position and the normal to the surface. We give an efficient algorithm for evaluating Bézier and B-spline tensor products for such information. The algorithm is an extension of a method for computing the position and tangent to a Bézier curve, and is asymptotically twice as fast as the standard bilinear algorithm. [1]

Note: This is a corrected version of the technical report.

1 Introduction

Many applications require evaluating both the value and derivatives of a function. For example, one way to generate an offset to a degree n Bézier curve $F(t) = \sum P_i B_i^n(t)$ is to evaluate the position and normal to the curve at a sufficiently large number of values of the parameter t. Similarly, a degree $n \times m$ tensor product Bézier surface

$$F(t^1; t^2) = \sum_{i,j} P_{i,j} B_i^n(t^1) B_j^m(t^2)$$

is often rendered by tessellating the surface (for reasons that will become clear in Section 2, we use superscripts to differentiate between the parametric directions of F). To use smooth shading, both the position and the normal must be evaluated for a sufficiently large number of values for t^1 and t^2. Finally, scalar tri-variate B-spline

[1]This work was supported in part by the Xerox Corporation, Hewlett-Packard, the Digital Equipment Corporation, and the National Science Foundation under grant CCR-8957323.

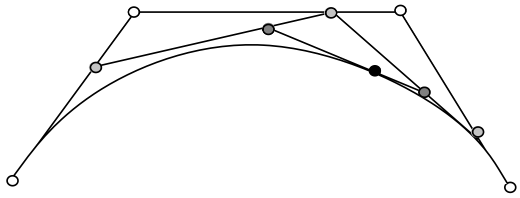

Figure 1: De Casteljau's Algorithm. The Bézier control points are shown in white. At each step, intermediate points are computed as convex combinations of the points in the previous step.

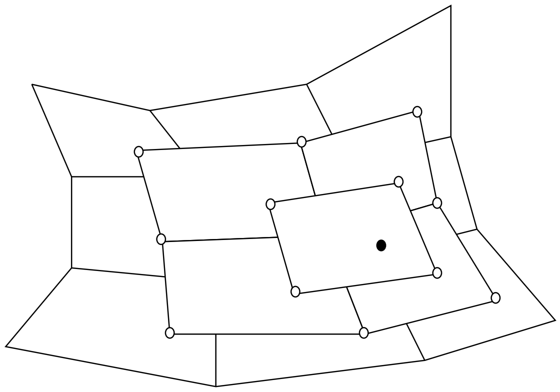

Figure 2: Bilinear interpolation.

functions have been used for solid three dimensional textures and bump mapping. The normal to the bump map is given by the gradient of the tri-variate B-spline function. For a discussion of Bézier curves and surfaces, see [Far93].

A univariate Bézier curve F is typically evaluated using de Casteljau's algorithm (Figure 1). For a degree n curve, the algorithm proceeds in n steps. The j-th step of the algorithm takes $n - j + 2$ control points and produces a new set of $n - j + 1$ control points. The single point produced by the n-th step is the point of evaluation. It is well known that the derivative of F at the point of evaluation is proportional to the difference of the two points produced at the $n - 1$st step.

We can evaluate a bi-variate $n \times n$ Bézier surface by performing repeated bilinear interpolation (see Figure 2 and Farin [Far93]). The final point is the point of evaluation, and again, the points in the next to last step can be used to calculate the derivatives of the surface. However, this algorithm does not suffice for an $n \times m$ tensor product surface with $m \neq n$.

In this paper we develop an algorithm to handle the general case of arbitrary m and n. Our approach is to run the univariate version of de Casteljau's algorithm successively in each parametric dimension. Each time, we stop the evaluation "one short" of completion, as described above for the evaluation of a Bézier curve for its position and derivative. The resulting multi-linear function is then evaluated to compute both the value and the derivatives.

In Section 2, we develop the theory behind our algorithm. In Section 3, we present the algorithm, and finally, we analyze the efficiency of our algorithm in section 3.1, and show that it is asymptotically twice as fast as the bilinear interpolation algorithm.

2 Theory

In this section, we describe the mathematical quantities computed by the algorithm, quantities that are best characterized using the notion of blossoming. The details of the computation are presented in the next section.

The blossoming principle states:

> For every polynomial $F(t)$ of degree n, there is a unique, symmetric, n-affine map, $f(t_1, \ldots, t_n)$, that agrees with F on the diagonal, i.e., $F(t) = f(t, \ldots, t)$.

The blossom of a polynomial can be evaluated to obtain the Bézier and B-spline control points for a segment of the curve. For a degree n Bézier curve $F(t)$ defined on the interval $[a, b]$, the Bézier control points are given by $f(a, \ldots, a)$, $f(a, \ldots, a, b)$, $\cdots$, $f(b, \ldots, b)$. For a degree n B-spline curve $F(t)$ with knots $x_0 < x_1 < \cdots < x_{m+n+1}$, the B-spline control points are given by $f(x_{i+1}, x_{i+2}, \ldots, x_{i+n})$, for $0 \leq i < m+1$. (Ramshaw [Ram88] discusses some subtleties that occur with the blossoms of B-splines.) For simplicity, we will only consider uniform B-splines.

Given the blossom f of an n-th degree polynomial F, we can calculate the derivatives of $F(p)$ by evaluating f at p $n-1$ times, leaving an affine function

$$f^*(t) := f(\underbrace{p, \ldots, p}_{n-1}, t).$$

We can evaluate either f or f^* to obtain the derivative of F:

$$\begin{aligned} \frac{\partial F}{\partial t}(p) &= n[f(1, p, \ldots, p) - f(0, p, \ldots, p)] \\ &= n[f^*(1) - f^*(0)]. \end{aligned}$$

Note that, for a Bézier curve defined on the interval $[a, b]$, the derivative is also given by

$$\frac{\partial F}{\partial t}(p) = n\frac{f(b, p, \ldots, p) - f(a, p, \ldots, p)}{b - a}.$$

For notational simplicity, we will use the former varient of the derivative. The implementer should use the latter, however, as these values can be obtained directly from the intermediate points of de Casteljau's algorithm.

The blossom of a tensor product is formed by blossoming each of its arguments independently: For a d-variate tensor product $F(t^1; \cdots; t^d)$ of degree $n_1 \times n_2 \times \cdots \times n_d$, the blossom of F is a function $f(t_1^1,\ldots,t_{n_1}^1;\ \cdots\ ;t_1^d,\ldots,t_{n_d}^d)$ that is symmetric in each block of arguments and agrees with F on the diagonal, i.e., $F(p^1;p^2;\cdots;p^d) = f(p^1\ldots,p^1;\ \cdots\ ;p^d,\ldots,p^d)$.

For a tensor product $F(t^1;\cdots;t^d)$, the partial derivatives of F can be determined by evaluating all but one of the arguments of its blossom f, resulting in a curve. The i-th partial of F is the derivative of the i-th such curve:

$$\frac{\partial F}{\partial t^i}(p^1;\cdots;p^d) = n_i\Big[f(p^1,\ldots,p^1;\ \cdots\ ;\ \ 1,p^i,\ldots,p^i;\ \ \cdots\ ;\ p^d,\ldots,p^d) - f(p^1,\ldots,p^1;\ \cdots\ ;\ \ 0,p^i,\ldots,p^i;\ \ \cdots\ ;\ p^d,\ldots,p^d)\Big].$$

The problem we are solving can be stated as follows:

Given: A d-variate tensor product Bézier function $F(t^1;\cdots;t^d)$, and a point $(p^1,\ldots,p^d)$ in its parameter space.

Find: The value and gradient of F at $F(p^1;\cdots;p^d)$.

As indicated earlier, our approach appeals to properties of f, the blossom of F. From f, we build the function

$$L(t^1;\cdots;t^d) = f(t^1,p^1,\ldots,p^1;\ \cdots\ ;t^d,p^d,\ldots,p^d).$$

The value and derivatives of F are then computed by evaluation of L, using the identities

$$F(p^1;\cdots;p^d) = L(p^1;\cdots;p^d)$$

and

$$\frac{\partial F}{\partial t^i}(p^1;\cdots;p^d) = n_i\Big[L(p^1;\cdots;p^{i-1};\quad 1\quad ;p^{i+1};\cdots;p^d) - L(p^1;\cdots;p^{i-1};\quad 0\quad ;p^{i+1};\cdots;p^d)\Big].$$

3 The Algorithm

There are a variety of ways to compute the function L. For efficiency, we evaluate f by operating on each of the d blocks of arguments in turn (using de Casteljau's algorithm) but always stopping "one level short of completion." The function L is obtained when all argument blocks have been processed. To make these ideas more precise, after i blocks have been processed, we have computed the control points representing the function

$$f(t^1,p^1,\ldots,p^1;\ \cdots\ ;t^i,p^i,\ldots,p^i;\ \ t_1^{i+1},\ldots,t_{n_{i+1}}^{i+1};\ \cdots\ ;t_1^d,\ldots,t_{n_d}^d).$$

We denote this partial evaluation of f as

$$L_i(t^1;\ \cdots\ ;t^i).$$

Thus, after all d blocks have been processed we have constructed a representation of L_d, which by construction is equal to L.

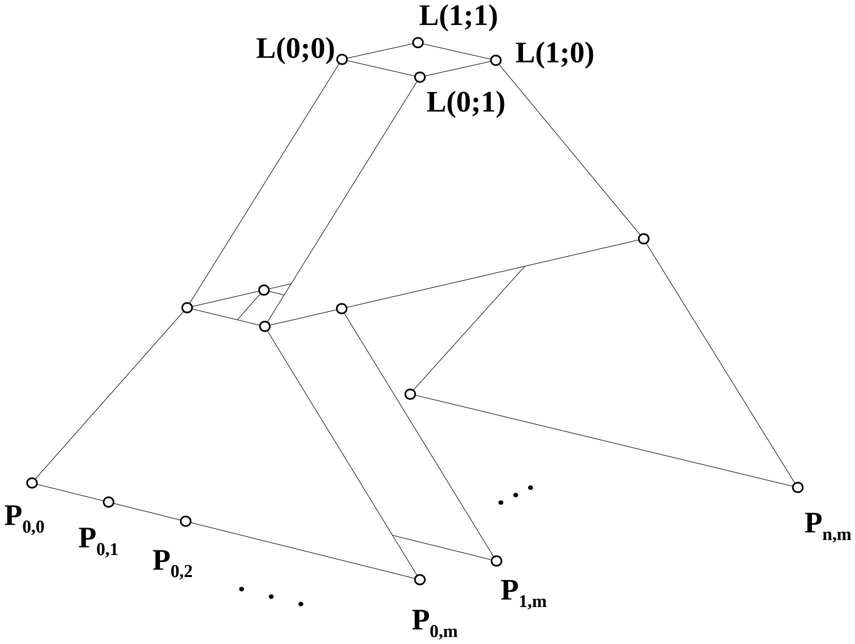

Figure 3: Data Flow Diagram of Algorithm

The steps needed to compute L are illustrated in Figure 3 for a tensor product surface (i.e., for $d = 2$). We first run the de Casteljau algorithm on all the rows of control points of F, but stop one step short of the top. This leaves us with two columns of control points that describe the function L_1. We then run the de Casteljau algorithm on each of these two columns, again stopping one evaluation short of the top, thereby obtaining four control points describing the functions $L_2(t^1;t^2) = L(t^1;t^2)$. This bilinear surface osculates with $F(t^1;t^2)$ to first order. Thus, $F(t^1;t^2)$ as well as both partials of F can be computed from L. Pictorially, the algorithm is illustrated in Figure 3. Code for the algorithm appears in Figure 4.

The algorithm readily generalizes to d-variate tensor products, and a similar approach (based on the Cox-de Boor algorithm) may be used to evaluate tensor products B-splines. As an example of both generalizations, we give code to compute L for a triquadratic, uniform, tensor product B-spline (Figure 5).

3.1 Comparison of Algorithms

In this section we compare our algorithm to the bilinear algorithm. We do not compare algorithms such as forward differencing [Roc87, LSP87] since they are not designed to evaluate the surface at arbitrary parametric values on demand.

We can use either our algorithm or the bilinear algorithm to evaluate an $n \times n$ tensor product surface F. If the blossom of F is f and we wish to evaluate it at the point (p^1, p^2) then the bilinear algorithm proceeds by setting one argument in each argument block. E.g., from the initial blossom

$$f(t_1^1, \ldots, t_n^1;\ t_1^2, \ldots, t_n^2),$$

```
/* Given the control points P_{i,j} for a n × m tensor product
      Bezier patch F and a pair of parameter values p^1 and p^2,
      return the value and partial derivatives of F at p^1, p^2 */

EvalTensorProduct(IN: P_{i,j}, n, m, p^1, p^2; OUT: F, ∂F/∂p^1, ∂F/∂p^2)

  /* Run de Casteljau's curve algorithm on the rows of the net */
  for (i = 0; i ≤ n; i++) { /* for each row */
     for (k = 1; k < m; k++) {
        for (j = 0; j ≤ m − k; j++) {
           P_{i,j} = (1 − p^2) * P_{i,j} + p^2 * P_{i,j+1}
        }
     }
  }

  /* Now run de Casteljau on the first two columns of the net */
  for (j = 0; j ≤ 1; j++){ /* for each column */
     for (k = 1; k < n; k++) {
        for (i = 0; i ≤ n − k; i++) {
           P_{i,j} = (1 − p^1) * P_{i,j} + p^1 * P_{i+1,j}
        }
     }
  }

  /* now we have a bilinear patch in P_{i,j} */
  L_{p^1,0} = (1 − p^1) * P_{0,0} + p^1 * P_{1,0}
  L_{p^1,1} = (1 − p^1) * P_{0,1} + p^1 * P_{1,1}
  L_{0,p^2} = (1 − p^2) * P_{0,0} + p^2 * P_{0,1}
  L_{1,p^2} = (1 − p^2) * P_{1,0} + p^2 * P_{1,1}

  /* Compute the normal by computing partial derivatives; ignore scale */

  ∂F/∂p^1 = L_{1,p^2} − L_{0,p^2}
  ∂F/∂p^2 = L_{p^1,1} − L_{p^1,0}

  /* finally, evaluate the function...*/
  F = (1 − p^2) L_{p^1,0} + p^2 L_{p^1,1}
```

Figure 4: Code to Evaluate Bézier Tensor Product Patch.

```
/* Given the control points P_{i,j,k} for a triquadratic tensor product
      B-spline volume F and a triplet of parameter values p^1, p^2, p^3,
      return the value of the gradient L of F at p^1, p^2, p^3. */

EvalTriquadradic(IN: P_{i,j}, p^1, p^2, p^3; OUT: F, L)

   for (j = 0; j <= 2; j++) {
      for (k = 0; k <= 2; k++) {
         for (i = 0; i <= 1; i++) {
            P_{i,j,k} = ((2+i-p^1)P_{i,j,k} + (p^1-i)P_{i+1,j,k}) / 2
         }
      }
   }

   for (i = 0; i <= 1; i++) {
      for (k = 0; k <= 2; k++) {
         for (j = 0; j <= 1; j++) {
            P_{i,j,k} = ((2+j-p^2)P_{i,j,k} + (p^2-j)P_{i,j+1,k}) / 2
         }
      }
   }

   for (i = 0; i <= 1; i++) {
      for (j = 0; j <= 1; j++) {
         for (k = 0; k <= 1; k++) {
            L_{i,j,k} = ((2+k-p^3)P_{i,j,k} + (p^3-k)P_{i,j,k+1}) / 2
         }
      }
   }
```

Figure 5: Code to Evaluate Uniform Tensor Product B-spline Volume.

the bilinear algorithm computes control points representing the function

$$f(t^1, \ldots, t^1_{n-1}, p^1;\ t^2_1, \ldots, t^2_{n-1}, p^2),$$

as illustrated in Figure 2. In contrast, our algorithm evaluates one argument block at at time. Thus, from the initial blossom, our algorithm instead computes control points for

$$f(t^1, p^1, \ldots, p^1;\ t^2_1, \ldots, t^2_n)$$

as illustrated in Figure 3.

In the bilinear algorithm, successive levels of control points are computed performing bilinear interpolation on sets of four control points on the previous level, typically using three linear interpolations, requiring six multiplications and three additions. For an $n \times n$ tensor product surface, there are $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ such combinations.

For our algorithm, control points for a new level are a linear combination of two points, each requiring two multiplications and one addition. When evaluating the first block of arguments, there are $(n+1)\sum_{i=1}^{n-1} i = \frac{(n+1)n(n-1)}{2}$ combinations. For the second block, there are $2\sum_{i=1}^{n-1} i = n(n-1)$ combinations. After these first two evaluations, we have reduced the control net to a bilinear patch, and we need an additional six multiplications and three additions to compute the point on the surface.

Thus, the total number of multiplications used by the bilinear algorithm is

$$n(n+1)(2n+1),$$

while our algorithm uses

$$(n+1)(n+1)n + 2(n+1)n$$

multiplications. (Both algorithms require half the number of additions as multiplications.) While both algorithms have the same asymptotic complexity ($O(n^3)$), our algorithm has a constant factor about half that of the bilinear algorithm. In particular, our algorithm is computationally more efficient for surfaces of bi-degree $n \geq 2$ (as expected, both algorithms perform identically for a bilinear surface).

Note that the number of multiplications at each step of the bilinear algorithm can be reduced from six to four, although three additions would still be required. With this variant of the bilinear algorithm, our algorithm is slightly slower when evaluating a bilinear patch, but is still more efficient when $n \geq 2$, and asymptotically our algorithm only requires two-thirds the number of multiplications and only half the additions.

4 Conclusions

We have presented an efficient algorithm for computing values and first derivatives of multivariate tensor product Bézier and B-spline functions.

A few notes are in order. First, it is computationally advantageous to evaluate the tensor product first in the parametric direction of lowest degree. Second, we stopped at the next to last step of the de Casteljau algorithm since we were only interested in first derivatives. Higher order derivatives can be computed if the evaluation is stopped at an earlier stage. For example, to compute second derivatives of a tensor product Bézier surface, we could stop two levels early in the de Casteljau algorithm (when computing both the rows and the columns). The nine points thus constructed then form a biquadratic Bézier surface from which all second derivatives can be calculated.

References

[Far93] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design.* Academic Press, San Diego, third edition, 1993.

[LSP87] Sheue-Ling Lien, Michael Shantz, and Vaughan Pratt. Adaptive forward differencing for rendering curves and surfaces. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):111–118, July 1987.

[Ram88] Lyle Ramshaw. Béziers and B-splines as multiaffine maps. In R. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 757–776. Springer Verlag, 1988.

[Roc87] Alyn P. Rockwood. A generalized scanning technique for display of parametrically defined surfaces. *IEEE Computer Graphics and Applications*, 7(8):15–26, August 1987.