Functional Composition Algorithms via Blossoming

TONY D. DEROSE
University of Washington

RONALD N. GOLDMAN
Rice University

HANS HAGEN
Universitaet Kaiserlautern
and
STEPHEN MANN
University of Washington

In view of the fundamental role that functional composition plays in mathematics, it is not surprising that a variety of problems in geometric modeling can be viewed as instances of the following composition problem: given representations for two functions F and G, compute a representation of the function $H = F \circ G$. We examine this problem in detail for the case when F and G are given in either Bézier or B-spline form. Blossoming techniques are used to gain theoretical insight into the structure of the solution which is then used to develop efficient, tightly codable algorithms. From a practical point of view, if the composition algorithms are implemented as library routines, a number of geometric-modeling problems can be solved with a small amount of additional software.

Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*curve, surface, solid, and object representations*; J.6 [**Computer Applications**]: Computer-Aided Engineering—*computer-aided design*

General Terms: Algorithms

Additional Key Words and Phrases: B-Splines, Bézier curves, computer-aided geometric design, free-form deformations, triangular Bézier surface patches, tensor-product surface patches

This work was supported in part by Xerox Corporation, IBM, Hewlett-Packard, Digital Equipment Corporation, and the National Science Foundation under grants CCR-8957323 and DMC-8802949.

Authors' addresses: T. D. DeRose, Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195; S. Mann, Computer Science Department, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1; R. N. Goldman, Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251; H. Hagen, Universitaet Kaiserslautern, FB Informatik, Postfach 3049, 6750 Kaiserslautern, West Germany.

1. INTRODUCTION AND MOTIVATION

Many problems in curve and surface modeling can be viewed as instances of functional composition. In [6] two procedures for composing multivariate polynomials given in Bézier simplex form were derived: one called the product algorithm, the other called the blossom algorithm. The product algorithm was found to be more efficient for machine implementation, whereas the blossom algorithm was found to be geometrically more intuitive. Particular problems that can be solved using these composition algorithms include: evaluation, subdivision, and nonlinear reparameterization of Bézier simplexes (i.e., curves, triangular patch surfaces, etc.); exact representations of shapes modeled in Bézier simplex form that have undergone free-form deformation [17]; and the joining of two Bézier curves with geometric continuity of arbitrary order.

The purpose of this paper is to extend the work begun in [6] to tensor products, rational functions, and B-splines. Roughly speaking, we develop efficient algorithms for solving problems of the following form: Given polynomial or rational functions $G : \mathscr{X} \to \mathscr{Y}$ and $F : \mathscr{Y} \to \mathscr{Z}$ in either Bézier simplex or tensor-product B-spline form,[1] find a representation of the function $H = F \circ G$. As will be shown, the most natural form of the representation for H depends on the form in which G is given, irrespective of the form of F. For example, if G is given in tensor-product form, then H will be represented most naturally in tensor-product form. Instead of directly generalizing the proofs given in [6], we have taken a new approach that considerably simplifies the derivation of the blossom algorithm. In fact, the new proof technique is powerful enough to allow the extensions to be addressed with very little additional effort. We also provide an optimized recursive version of the algorithm that exploits certain symmetries in the formulae, resulting in an efficient, tightly codable implementation of composition.

Our extended algorithms are capable of solving a broadened class of problems, including: finding exact B-spline representations of trim curves on B-spline surfaces (see Figure 1 and Color Plate 1); computing exact representations of NURB curves and surfaces that have undergone free-form deformation (when the deformation is represented in Bézier form); and converting between the tensor-product and Bézier simplex forms. These algorithms can be used to practical advantage by implementing them as a set of library routines. Higher-level software designed to solve specific modeling problems such as those listed above could then call on these library routines, thereby encapsulating the bulk of the computation. Moreover, the simple recursive form of the algorithms allows them to be written rapidly and compactly, resulting in a small amount of powerful code.

Although the algorithms we present could be considered as specializations of a single, unifying algorithm, the notational complexity of stating and

[1] For reasons given in Section 4.4, one restriction is that F and G cannot both be in tensor-product B-spline form.

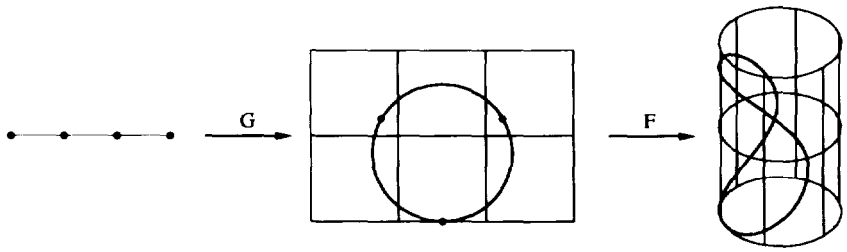
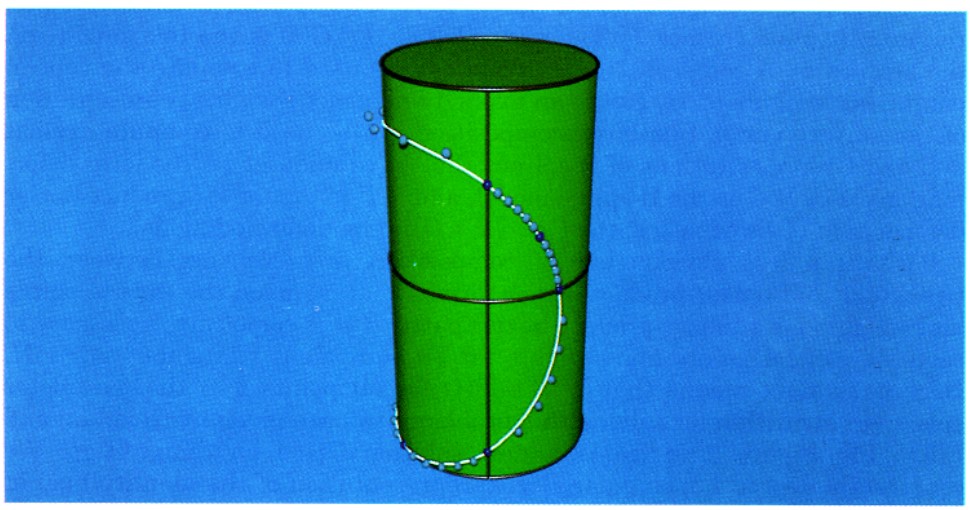


Fig. 1. G represents a B-spline trim curve, in this case a circle, in the parameter space of a tensor-product B-spline surface F, in this case a cylinder. See also Color Plate 1.



Color Plate 1. The B-spline representation of a rational B-spline curve (shown in white) on a rational B-spline surface (shown in green). Balls denote the control points of the curve, with the dark balls indicating where B-spline segments abut. Red lines denote the knot lines of the B-spline surface.

deriving the general algorithm is prohibitive. We have therefore chosen to develop the various algorithms separately. In Section 2, we begin by presenting several problems that can be solved with composition. In Section 3, we present background material. The basic proof techniques used are developed in Section 4.1 by rederiving the formula given in [6] on which the blossom algorithm is based. In Section 4.2, we generalize these ideas to the composition of tensor products. The results in Section 4.1 and 4.2 are readily extended to rational functions, as shown in Section 4.3. However, the extension to B-splines developed in Section 4.4 is inherently more difficult for the reasons explained in that section. Finally, in Section 5, we give optimized

pseudocode for the composition of polynomials to indicate how the composition algorithms may be implemented efficiently.

2. APPLICATIONS

Throughout the paper, affine spaces are denoted by upper-case characters set in a script typeface, such as $\mathscr{X}$, $\mathscr{Y}$, etc. Points in affine spaces are set in bold face. Other objects, including scalars, functions, indices, and sets of points, are set in italics. The introduction of additional notational conventions will be postponed until Section 3 (see Table I for a summary of the notation).

Before presenting the details of the algorithms for accomplishing composition, we examine more closely the use of composition in a variety of geometric-modeling applications. Consider for instance the situation shown in Figure 1 where a rational B-spline curve G is used to trim a tensor-product rational B-spline surface F. The curve $H(\mathbf{t}) = F \circ G(\mathbf{t})$ is the image of G on the surface F. A composition algorithm can be used to compute the control points and weights of H, giving an explicit and exact B-spline representation of the surface curve. Similarly, composition can be used to compute explicit and exact representations of B-splines that have undergone free-form deformations [17]. If G is the B-spline model, and if F is the tensor-product Bézier deformation of three-space, then $H = F \circ G$ is the deformed model.

Perhaps a less obvious use of composition is to convert between the triangular and tensor-product Bézier forms. That is, given the control points and weights for a tensor-product Bézier patch F it is sometimes necessary to compute control points and weights for an equivalent triangular patch H. Here equivalent means that $H(\mathbf{t}) = F(\mathbf{t})$ for all points $\mathbf{t}$ in the parameter space. An algorithm for solving this problem has been given by Goldman and Filip [10]; Figure 2 indicates a new solution based on composition. We construct a degree-1 triangular Bézier representation of the identity map of the parameter plane onto itself. That is, we construct G such that $G(\mathbf{t}) = \mathbf{t}$. This is easily done by setting the domain triangle of G to **abc**, and setting the control points of G to **a**, **b** and **c**. We now use the composition algorithm to compute the control points of $H(\mathbf{t}) = F \circ G(\mathbf{t})$. Since G is in triangular form, so is H, and since $G(\mathbf{t}) = \mathbf{t}$, $H(\mathbf{t}) = F(\mathbf{t})$. Conversion from a triangular patch F to an equivalent tensor product H can be accomplished using the method of Brueckner [3]; alternatively, the problem can be recast as composition by constructing G to be a bilinear representation of the identity map. The control points of H are then the control points of $F \circ G$.

One situation where the conversion between rational tensor-product form and rational triangular form may be desirable is in the construction of surfaces of revolution. If Q is a (typically planar) Bézier curve to be revolved around some axis, A it is relatively straightforward to construct a rational tensor-product control net for the surface of revolution F. However, if, as shown in Figure 3, an endpoint of Q lies on A, then an edge of F degenerates to a point, leading to a patch of triangular shape. It is therefore natural to seek a triangular representation of the patch. This can be done using composition together with a (rational) quadratic transformation that maps the

Table I. Summary of Notation

Notation	Description
$\mathbb{R}$	The set of real numbers.
$\mathbb{R}^n$	n-dimensional real space.
$a, \ldots, z$	Integers and real numbers.
$\mathscr{A}, \mathscr{A}_1, \mathscr{Y}$, etc.	Affine spaces.
$\mathbf{a}, \mathbf{A}, \ldots, \mathbf{z}, \mathbf{Z}$	Affine points.
Δ	Domain simplex.
$\Delta_{\mathscr{A}_1} \times \Delta_{\mathscr{A}_2}$	Domain simploid.
$\vec{a}, \ldots, \vec{z}$	Multi-indices.
$\vec{0}$	Multi-index whose components are all zero.
$\vec{e}_j$	Multi-index with a one in the jth component and zeros elsewhere.
$\lvert\vec{\imath}\rvert$	$= i_0 + \cdots + i_k$ where $\vec{\imath} = (i_0, \ldots, i_k)$.
$\mathbb{I}_k^d$	The set of all multi-indices $\vec{\imath} = (i_0, \ldots, i_k)$ with $\lvert\vec{\imath}\rvert = d$.
I, J, I^1, etc.	Hyperindices.
$\mathbb{I}_k^{d,m}$	The set of all hyperindices $I = (\vec{\imath}_1, \ldots, \vec{\imath}_m)$ with $\vec{\imath}_1, \ldots, \vec{\imath}_m \in \mathbb{I}_k^d$.
$S_k^{\mathscr{I},m}$	A subset of $\mathbb{I}_k^{\mathscr{I},m}$ containing exactly one permutation of each hyperindex in $\mathbb{I}_k^{\mathscr{I},m}$.
$\lvert I\rvert$	$= \vec{\imath}_1 + \cdots + \vec{\imath}_m$ where $I = (\vec{\imath}_1, \ldots, \vec{\imath}_m)$.
$\lVert I\rVert$	$= dm$ where $I \in \mathbb{I}_k^{d,m}$.
$\binom{\lvert\vec{\imath}\rvert}{\vec{\imath}}$	The multinomial coefficient $(\lvert\vec{\imath}\rvert)/(i_0!, \cdots, i_k!)$, $\vec{\imath} = (i_0, \ldots, i_k)$.
$B_{\vec{\imath}}^d(\mathbf{u})$	The $\vec{\imath}$th Bernstein polynomial of degree d.
$B_I^{d,m}$	$= B_{\vec{\imath}_1}^d(\mathbf{u}) \cdots B_{\vec{\imath}_m}^d(\mathbf{u})$, $I = (\vec{\imath}_1, \ldots, \vec{\imath}_m)$.
$\mathscr{C}(I)$	A combinatorial constant given by $\left(\binom{\lvert\vec{\imath}_1\rvert}{\vec{\imath}_1} \cdots \binom{\lvert\vec{\imath}_m\rvert}{\vec{\imath}_m}\right) \Big/ \binom{\lVert I\rVert}{\lvert I\rvert}$.
$q(\mathbf{u}_1, \ldots, \mathbf{u}_d)$	Blossom of a Bézier simplex $Q(\mathbf{u})$ of degree d.
$\hat{Q}(\mathbf{u})$	Homogenization of a rational Bézier simplex $Q(\mathbf{u})$.
$\hat{q}(\mathbf{u}_1, \ldots, \mathbf{u}_d)$	Multilinear blossom of $\hat{Q}(\mathbf{u})$.
Proj( )	The map from $\mathbb{R}^{K+1}$ to $\mathbb{R}^{K+1}$ defined by $\text{Proj}(y_0, \ldots, y_K) = (y_0, \ldots, y_K)/(y_0 + \cdots + y_K)$.

interior of a triangle so as to cover the interior of a square. This is done by "blowing up" one of the vertices of the triangle into a line.[2]

The quadratic transformation operates as follows: Let b_0, b_1, b_2 be the barycentric coordinates on the triangle, and let u, v be a Cartesian coordinates on the square. The transformation $G:(b_0, b_1, b_2) \rightarrow (u, v)$ is given by:

$$u = \frac{b_1}{b_1 + b_2}$$

$$v = b_0$$

[2] Quadratic transformations have been used elsewhere in CAGD. For example, Warren [19] has used them recently to develop n-sided patch representations.

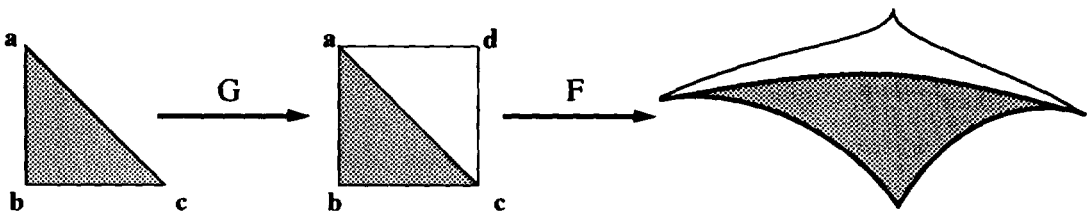


Fig. 2. The conversion of a tensor-product Bézier patch into triangular Bézier form.

To express G in rational Bézier form, we must put u and v over a common denominator,

$$G(b_0, b_1, b_2) = \frac{(b_1, b_0(b_1 + b_2))}{b_1 + b_2}$$

from which the control points and weights can be obtained by a simple change of basis [9]. G maps the line $b_0 = 0$ to the line $v = 0$, the line $b_1 = 0$ to the line $u = 0$, the line $b_2 = 0$ to the line $u = 1$, but the image of point (1, 0, 0) under G is the entire $v = 1$ line. The vertex (1, 0, 0) is therefore "blown up." Since G is in triangular patch form, the control points and weights of $H = F \circ G$ will form a rational triangular representation for the surface of revolution.

Composition is also a useful tool when dealing with S-patches [12, 13]. S-patches are rational generalizations of Bézier surfaces that admit any number n of boundary curves. In [12] it is shown how to use composition to convert an n-sided S-patch into an m-sided S-patch, for arbitrary n and m. Another problem that can be solved using composition is finding the representation of a trimmed tensor-product patch as a collection of untrimmed S-patches. Figure 4 depicts a simple instance. The map G is an S-patch that carries the domain pentagon (left) into the domain square (center) so that the boundaries of G match the boundaries of the trimmed patch F. The control points of $H = F \circ G$ are therefore the S-patch control points for the trimmed portion of the surface.

3. NOTATION AND BACKGROUND

We briefly summarize the basic notions of blossoming and introduce a number of notational conventions that have been invented to simplify the implementations needed in Sections 4 and 5. As is often the case with compact notation, ours is something of a two-edged sword. On one hand the notation reduces the visual complexity of the derivation so as to expose its simple underlying structure, but on the other hand some effort is required to become familiar with the notation. On the whole, we feel that the investment required to learn the notation is time well spent; see Table I for a summary.

Most of our notation deals with the indexing of multivariate polynomials. While univariate Bernstein polynomials are commonly indexed with integers, multivariate Bernstein polynomials are most easily indexed using tuples of non-negative integers such as $\vec{i} = (i_0, \ldots, i_k)$, $i_0, \ldots, i_k \geq 0$. Such tuples are

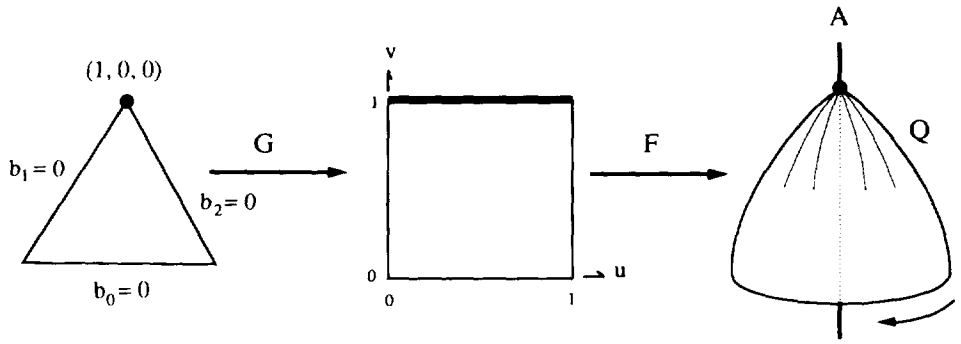


Fig. 3. A degenerate tensor-product formed as a surface of revolution.

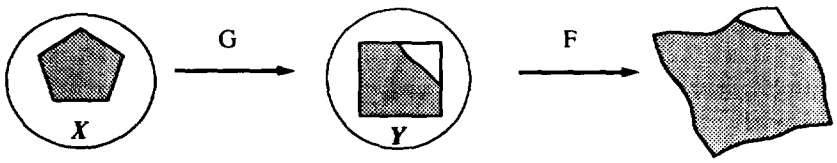


Fig. 4. Representing a trimmed tensor-product patch as an S-patch.

called *multi-indices*. We denote the norm of a multi-index $\vec{\imath}$ by $|\vec{\imath}|$ and define it to be the sum of the components of $\vec{\imath}$. We use the symbol $\mathbb{I}_k^d$ to stand for the set of all multi-indices $\vec{\imath} = (i_0, \ldots, i_k)$ with $|\vec{\imath}| = d$. For example, $\mathbb{I}_2^2 = \{(0, 0, 2), (0, 1, 1), (0, 2, 0), (1, 0, 1), (1, 1, 0), (2, 0, 0)\}$. The symbol $\vec{e}_j$ will be used to denote a multi-index whose components are all zero except for the jth component, which is one; $\vec{0}$ denotes a multi-index whose components are all zero. Addition and subtraction of multi-indices is defined componentwise. We shall also have occasion to use tuples of multi-indices, such as $I = (\vec{\imath}_1, \ldots, \vec{\imath}_m) \in \mathbb{I}_k^{d,m}$, where $\mathbb{I}_k^{d,m}$ denotes the m-fold Cartesian product $\mathbb{I}_k^d \times \cdots \times \mathbb{I}_k^d$. We shall refer to such tuples as *hyper-indices*, and we again use the notation $|I|$ to denote the sum of the components of I. Notice that whereas $|\vec{\imath}|$ is an integer, $|I|$ is a multi-index. Therefore, for $I \in \mathbb{I}_k^{d,m}$, the expression $\|I\|$ evaluates to the integer $|\vec{\imath}_1| + \cdots + |\vec{\imath}_m| = dm$. Finally, juxtaposition of hyper-indices denotes concatenation, as in $IJ = (\vec{\imath}_1, \ldots, \vec{\imath}_m, \vec{\jmath}_1, \ldots, \vec{\jmath}_p)$.

Using this notation, the k-variate Bernstein polynomials of degree d can be defined by

$$B_{\vec{\imath}}^d(b_0, \ldots, b_k) = \binom{d}{\vec{\imath}} b_0^{i_0} b_1^{i_1} \cdots b_k^{i_k}$$

where $\vec{\imath} = (i_0, \ldots, i_k) \in \mathbb{I}_k^d$, $b_0, \ldots, b_k \in \mathbb{R}$, and where

$$\binom{d}{\vec{\imath}} = \frac{d!}{i_0! i_1! \cdots i_k!}$$

is the multinomial coefficient. Notice that each of the Bernstein polynomials is a *homogeneous polynomial* of degree d, a fact that will become important when considering rational functions. It is known [5] that for every polynomial $Q : \mathscr{X} \to \mathscr{Y}$ of degree d, where $\mathscr{X}$ is an affine space of dimension k and where $\mathscr{Y}$ is an affine space of dimension K, there exist unique points $\{\mathbf{V}_{\vec{i}}\}_{\vec{i} \in \mathbb{I}_k^d}$ in $\mathscr{Y}$ such that

$$Q(\mathbf{u}) = \sum_{\vec{i} \in \mathbb{I}_k^d} \mathbf{V}_{\vec{i}} B_{\vec{i}}^d(b_0, \ldots, b_k), \tag{1}$$

where $b_0, \ldots, b_k$ are the barycentric coordinates of $\mathbf{u} \in \mathscr{X}$ relative to a simplex $\Delta = (\mathbf{x}_0, \ldots, \mathbf{x}_k)$ of points in $\mathscr{X}$. (For a description of affine spaces, simplexes, and barycentric coordinates see, for instance, Farin [9] or DeRose [7].)

A polynomial Q, when expressed as in Eq. (1), is called a Bézier simplex; the points $\mathbf{V}_{\vec{i}}$ are called the control net of Q with respect to the domain simplex Δ.

We shall often write $B_{\vec{i}}^d(\mathbf{u})$ with the understanding that $\mathbf{u}$ should be replaced with its barycentric coordinates relative to the appropriate domain simplex. If $I = (\vec{i}_1, \ldots, \vec{i}_m) \in \mathbb{I}_k^{d,m}$, we define $B_I^{d,m}(\mathbf{u})$ to be the product $B_{\vec{i}_1}^d(\mathbf{u}) \cdot \cdots \cdot B_{\vec{i}_m}^d(\mathbf{u})$. We shall make considerable use of the following product relation satisfied by the Bernstein polynomials:

$$B_I^{d,m}(\mathbf{u}) = \mathscr{C}(I) B_{|I|}^{dm}(\mathbf{u}) \tag{2}$$

where $\mathscr{C}(I)$ is a combinatorial constant given by

$$\mathscr{C}(I) = \frac{\binom{|\vec{i}_1|}{\vec{i}_1} \cdots \binom{|\vec{i}_m|}{\vec{i}_m}}{\binom{\|I\|}{|I|}}.$$

This relation is easily proved using simple manipulation of the Bernstein polynomials (c.f., [6]).

Ramshaw [15] has discovered how to exploit a connection between Bézier simplexes and symmetric multi-affine maps. A map $q(\mathbf{u}_1, \ldots, \mathbf{u}_d)$ is said to be multi-affine if it is affine when all but one of its arguments are held fixed; it is said to be symmetric if its value does not depend on the ordering of the arguments. Associated with every polynomial $Q : \mathscr{X} \to \mathscr{Y}$ of degree d there is a unique, symmetric, d-affine map, $q : \mathscr{X}^d \to \mathscr{Y}$, that agrees with Q on its diagonal (the diagonal of a multi-affine map $q(\mathbf{u}_1, \ldots, \mathbf{u}_d)$ is the function obtained when all arguments are equal: $Q(\mathbf{u}) = q(\mathbf{u}, \mathbf{u}, \ldots, \mathbf{u})$). Ramshaw refers to this multi-affine map q as the *blossom* of Q. As a simple univariate example where $\mathscr{X} = \mathscr{Y} = \mathbb{R}^1$, if

$$Q(u) = 1 + u - 3u^2, \quad u \in \mathbb{R}^1 \tag{3}$$

then

$$q(u_1, u_2) = 1 + \frac{u_1 + u_2}{2} - 3u_1u_2, \quad u_1, u_2 \in \mathbb{R}^1. \tag{4}$$

Ramshaw [15] has shown that the Bézier control net for a polynomial relative to a domain simplex can be obtained by evaluating the polynomial's blossom at the vertices of the simplex. More precisely, if $\mathscr{X}$ is an affine space of dimension k, $Q : \mathscr{X} \to \mathscr{Y}$ a polynomial of degree d whose blossom is q, and $\Delta = (\mathbf{x}_0, \ldots, \mathbf{x}_k)$ a simplex in $\mathscr{X}$, then Ramshaw shows that the Bézier control net of Q relative to Δ is given by

$$\mathbf{V}_{\vec{i}} = q\left(\overbrace{\underbrace{\mathbf{x}_0, \ldots, \mathbf{x}_0}_{i_0}, \underbrace{\mathbf{x}_1, \ldots, \mathbf{x}_1}_{i_1}, \ldots, \underbrace{\mathbf{x}_k, \ldots, \mathbf{x}_k}_{i_k}}^{d}\right)$$

for all $\vec{i} = (i_0, \ldots, i_k) \in \mathbb{I}_k^d$. For example, if we apply this result to the polynomial $Q(u)$ given in Eq. (3) using as the domain simplex the interval [0, 1], we find

$$\mathbf{V}_{(2,0)} = q(0,0) = 1$$

$$\mathbf{V}_{(1,1)} = q(0,1) = \frac{3}{2}$$

$$\mathbf{V}_{(0,2)} = q(1,1) = -1$$

A straightforward approach to evaluate an arbitrary value $q(\mathbf{u}_1, \ldots, \mathbf{u}_d)$ of a blossom would be to explicitly compute an expression similar to Eq. (4) for the blossom as a multivariate polynomial, and then to evaluate it using the arguments $\mathbf{u}_1, \ldots, \mathbf{u}_d$. This procedure is generally to be avoided for computational reasons since an explicit expression for a general d-variate polynomial of degree one in each variable can have as many as 2^d terms. Fortunately, symmetry of blossoms can be exploited to make the computation tractable. The computation of an arbitrary blossom value $q(\mathbf{u}_1, \ldots, \mathbf{u}_d)$ can be made particularly efficient if Q is known in Bézier form. Specifically, the generalization of de Casteljau's algorithm given in Figure 5 can be used. The routine *EvalBlossom()* takes as input the control net V of a Bézier simplex Q of degree d having as its domain an affine space $\mathscr{X}$ of dimension k and points $\mathbf{u}_1, \ldots, \mathbf{u}_d$ in $\mathscr{X}$. Returned is the blossom of Q, evaluated at $\mathbf{u}_1, \ldots, \mathbf{u}_d$. Notice that if $\mathbf{u}_1, \ldots, \mathbf{u}_d$ are equal to a common point $\mathbf{u}$, the algorithm reduces to de Casteljau's algorithm. A data structure called a *simplicial array*, used for storing control points and intermediate results, is described in Section 5.

A variation of blossoming can also describe rational functions.[3] Although it is possible to develop the following material in a coordinate-free manner, a number of ideas not central to composition would have to be introduced first.

[3] The remainder of this section can be skipped on first reading. It can be skipped also by readers interested only in the composition of polynomials.

```
EvalBlossom(V, u_1, ..., u_d)
{V is the control net for a Bézier simplex characterizing a blossom q
 returned is the point q(u_1, ..., u_d)}
begin
        V̄ ← Prepare(V)
        for p = 1 to d
            EvalBlossomArgument(V̄, p, u_p)
        endfor
        return V̄.cp_0^[d]
end

EvalBlossomArgument(V̄, p, u)
begin
        d ← V̄.degree
        (b_0, ..., b_k) ← Barycentric coordinates of u relative to
        V̄.domain
        for all ī ∈ I_k^(d-p)
            V̄.cp_ī^[p] ← b_0 V̄.cp_(ī+e_0)^[p-1] + ··· + b_k V̄.cp_(ī+e_k)^[p-1]
        endfor
end

Prepare(V)
{Initialize and return a structure into which the partial results of the blossom
 evaluation algorithm can be stored.}
begin
      d ← V̄.degree ← V.degree
      for all ī ∈ I_k^d
          V̄.cp_ī^[0] ← V.cp_ī
      endfor
      return V̄
end
```

Fig. 5. Evaluation of an arbitrary blossom value. If V is a control net for a Bézier simplex Q, V.degree is the degree of Q; V.domain denotes the domain simplex; and V.**cp** are the control points of Q relative to V.domain.

To simplify the discussion, we shall therefore leave the coordinate-free framework when dealing with rational functions.[4]

Rather than speak of a rational function Q of degree d as mapping points in an affine space $\mathscr{X}$ (of dimension k) into points in an affine space $\mathscr{Y}$ (of dimension K), we associate with each point $\mathbf{u} \in \mathscr{X}$ the point $(b_0, \ldots, b_k) \in \mathbb{R}^{k+1}$, where $(b_0, \ldots, b_k)$ are the barycentric coordinates of $\mathbf{u}$ relative to some simplex in $\mathscr{X}$. (To avoid a proliferation of symbols, we write $\mathbf{u} = (b_0, \ldots, b_k)$, although, in a strict sense, this is an abuse of notation.) This process effectively embeds the affine space $\mathscr{X}$ as the affine subset in $\mathbb{R}^{k+1}$ whose equation is

$$x_0 + \cdots + x_k = 1, \quad (x_0, \ldots, x_k) \in \mathbb{R}^{k+1}. \tag{5}$$

Readers familiar with projective geometry will note that this is a rather nonstandard embedding; the more common embedding, based on Cartesian

[4] Ramshaw has given a purely coordinate-free development [15].

coordinates, corresponds to the hyperplane $x_k = 1$. The embedding given in Eq. (5) is more convenient for our purposes because of the close relationship between the blossom evaluation algorithm and barycentric coordinates. The principle consequence of this embedding is that it allows the composition algorithm for polynomial Bézier simplexes, developed in Section 4.1, to be used virtually without change to compose rational Bézier simplexes.

By embedding $\mathscr{Y}$ analogously onto the affine subset $y_0 + \cdots + y_K = 1$ in $\mathbb{R}^{K+1}$, a rational function Q of degree d may be treated as a map from $\mathbb{R}^{k+1}$ to $\mathbb{R}^{K+1}$ expressed as the ratio of two polynomials $\hat{Q}: \mathbb{R}^{k+1} \to \mathbb{R}^{K+1}$ and $D: \mathbb{R}^{k+1} \to \mathbb{R}^1$, each of degree d:

$$Q(\mathbf{u}) = \frac{\hat{Q}(\mathbf{u})}{D(\mathbf{u})} = \frac{(\hat{Q}_0(\mathbf{u}), \ldots, \hat{Q}_K(\mathbf{u}))}{D(\mathbf{u})},$$

where $\hat{Q}_0, \ldots, \hat{Q}_K$ refer to the component functions of $\hat{Q}$. Since Q maps into the affine subset of $\mathbb{R}^{K+1}$, it must be that

$$\frac{\hat{Q}_0(\mathbf{u})}{D(\mathbf{u})} + \cdots + \frac{\hat{Q}_K(\mathbf{u})}{D(\mathbf{u})} = 1,$$

or equivalently, that

$$\hat{Q}_0(\mathbf{u}) + \cdots + \hat{Q}_K(\mathbf{u}) = D(\mathbf{u}).$$

Thus, the point $Q(\mathbf{u})$ can be obtained as the projection of $\hat{Q}(\mathbf{u})$ onto the affine subset $y_0 + \cdots + y_K = 1$. More precisely, if this projection is denoted by $\mathrm{Proj}: \mathbb{R}^{K+1} \to \mathbb{R}^{K+1}$ and defined by

$$\mathrm{Proj}(y_0, \ldots, y_K) = \frac{(y_0, \ldots, y_K)}{y_0 + \cdots + y_K}, \tag{6}$$

then $Q(\mathbf{u}) = \mathrm{Proj}(\hat{Q})$.

If Q is given in rational Bézier simplex form with control points $\mathbf{V}_i$ and weights w_i, then $\hat{Q}$ is given by

$$\hat{Q}(\mathbf{u}) = \sum_{i \in \mathbb{I}_k^d} w_i V_i B_i^d(\mathbf{u})$$

where each of the control points $\mathbf{V}_i$ is represented by barycentric coordinates $(v_{i,0}, \ldots, v_{i,K})$. It is generally convenient to set $\hat{\mathbf{V}}_i = w_i \mathbf{V}_i = (w_i v_{i,0}, \ldots, w_i v_{i,K})$. The fact that the Bernstein polynomials are homogeneous polynomials of degree d means that $\hat{Q}$, the *homogenization* of Q, is a homogeneous polynomial of degree d. It has been established in fields such as projective geometry that, associated with every homogeneous polynomial of degree d, there is a symmetric multilinear function that agrees with the polynomial on its diagonal. (Although symmetric multilinear functions are classically known as *polar forms*, the term multilinear blossom is more in keeping with Ramshaw's terminology.) Thus, associated with the rational function Q is the multilinear blossom $\hat{q}(\mathbf{u}_1, \ldots, \mathbf{u}_d)$ of $\hat{Q}$. It is precisely the homogeneity of $\hat{Q}$ that makes the embedding given in Eq. (5) so useful.

The evaluation algorithm for multilinear blossoms is nearly identical to the algorithm of Figure 5 for evaluating multi-affine blossoms. The only modifications are ones that trade an affine framework for a linear one. Specifically,

—$\mathbf{u}_1, \ldots, \mathbf{u}_d$ become elements of the linear space $\mathbb{R}^{k+1}$ instead of the affine space $\mathscr{X}$ of dimension k.

—V.domain becomes a basis for $\mathbb{R}^{k+1}$ instead of a simplex in $\mathscr{X}$. A computationally advantageous basis is of course the canonical basis, the ith element, which is a tuple consisting of all zero components except for the ith entry, which is a one.

—The statement in the routine *EvalBlossomArgument()* of Figure 5 that sets $(b_0, \ldots, b_k)$ to barycentric coordinates of $\mathbf{u}$ is removed. By choosing the canonical basis for $\mathbb{R}^{k+1}$, $(b_0, \ldots, b_k)$ are simply the components of $\mathbf{u} \in \mathbb{R}^{k+1}$.

We shall refer to the algorithm thus created as the *multilinear blossom evaluation algorithm*.

An important consequence of the homogeneous form of $\hat{Q}(\mathbf{u})$ is that

$$Q(a\mathbf{u}) = Q(\mathbf{u}), \quad \text{for all} \quad a \neq 0. \tag{7}$$

This follows from two facts:

(1) $\hat{Q}(a\mathbf{u}) = a^d \hat{Q}(\mathbf{u})$ since $\hat{Q}$ is homogeneous of degree d.

(2) $\text{Proj}(a\mathbf{y}) = \mathbf{y}$ for all $a \neq 0$ and all $\mathbf{y} \in \mathbb{R}^{K+1}$.

Thus, $Q(a\mathbf{u}) = \text{Proj}(\hat{Q}(a\mathbf{u})) = \text{Proj}(a^d \hat{Q}(\mathbf{u})) = \text{Proj}(\hat{Q}(\mathbf{u})) = Q(\mathbf{u})$. A more complete investigation of Eq. (7) and its connection to projective spaces is provided in [8]. Equation (7) will be exploited in Section 4.3 to develop a composition algorithm for rational functions.

4. ALGORITHM DEVELOPMENT

The problem considered in this section is, generally: given the coefficients of two polynomial (or rational) functions, $G: \mathscr{X} \to \mathscr{Y}$ and $F: \mathscr{Y} \to \mathscr{Z}$, find the coefficients of $H = F \circ G$. We will begin by rederiving the result for polynomials using a new proof technique. Then, this result is generalized to tensor products, to rational functions, and to B-splines.

4.1 Composing Bézier Simplexes

There are several ways to address the problem of polynomial composition. The approach in [6] is to express both polynomials in Bernstein form and manipulate the Bernstein polynomials. Another approach is to manipulate the blossoms of both polynomials using only the symmetry property of the blossom (personal communication, L. Ramshaw, Digital Systems Research Center). Both methods yield the same formula as the one given below in Eq. (8). The approach taken here is a hybrid of the above two methods that results in a concise proof. The blossom is used as the representation for one of the polynomials, whereas the Bernstein representation is used for the other polynomial.

To introduce our method, we first rederive the blossom algorithm of [6] using the hybrid proof technique. Specifically, the problem we wish to reexamine is:

Given. Affine spaces $\mathscr{X}$ (of dimension k), $\mathscr{Y}$ (of dimension K), and $\mathscr{Z}$ (of arbitrary dimension), control points $\{\mathbf{G}_{\vec{i}}\}_{\vec{i}\in\mathbb{I}_k^\ell}$ defining a degree ℓ Bézier simplex $G:\mathscr{X}\to\mathscr{Y}$ relative to a domain simplex $\Delta_{\mathscr{X}}\subset\mathscr{X}$ and control points $\{\mathbf{F}_{\vec{p}}\}_{\vec{p}\in\mathbb{I}_K^m}$ defining a degree m Bézier simplex $F:\mathscr{Y}\to\mathscr{Z}$ relative to a domain simplex $\Delta_{\mathscr{Y}}\subset\mathscr{Y}$.

Find. The control points $\{\mathbf{H}_{\vec{\cdot}}\}_{\vec{\cdot}\in\mathbb{I}_k^{\ell m}}$ of the degree ℓm Bézier simplex $H=F\circ G$ relative to $\Delta_{\mathscr{X}}$.

Solution. If f denotes the blossom of F, then we have

$$\mathbf{H}_{\vec{\cdot}}=\sum_{\substack{I\in\mathbb{I}_k^{\ell,m}\\ |I|=\vec{\cdot}}}\mathscr{C}(I)f(\mathbf{G}_I),\qquad \vec{\cdot}\in\mathbb{I}_k^{\ell m} \tag{8}$$

where $\mathbf{G}_I$ with $I=(\vec{i}_1,\ldots,\vec{i}_m)$ is an abbreviation for $(\mathbf{G}_{\vec{i}_1},\ldots,\mathbf{G}_{\vec{i}_m})$.

PROOF. Into the equation $H(\mathbf{t})=F(G(\mathbf{t}))=f(G(\mathbf{t}),\ldots,f(G(\mathbf{t}))$ substitute the Bernstein form of G to obtain

$$H(\mathbf{t})=f\Bigg(\sum_{\vec{i}_1\in\mathbb{I}_k^\ell}\mathbf{G}_{\vec{i}_1}B_{\vec{i}_1}^\ell(\mathbf{t}),\ldots,\sum_{\vec{i}_m\in\mathbb{I}_k^\ell}\mathbf{G}_{\vec{i}_m}B_{\vec{i}_m}^\ell(\mathbf{t})\Bigg).$$

The Bernstein polynomials sum to one, implying that the first argument to f is an affine combination of the points $\mathbf{G}_{\vec{i}_1}$. The fact that f is affine in its first argument can now be used to pull the summation outside of f's argument list, yielding

$$H(\mathbf{t})=\sum_{\vec{i}_1\in\mathbb{I}_k^\ell}f\Bigg(\mathbf{G}_{\vec{i}_1},\sum_{\vec{i}_2\in\mathbb{I}_k^\ell}\mathbf{G}_{\vec{i}_2}B_{\vec{i}_2}^\ell(\mathbf{t}),\ldots,\sum_{\vec{i}_m\in\mathbb{I}_k^\ell}\mathbf{G}_{\vec{i}_m}B_{\vec{i}_m}^\ell(\mathbf{t})\Bigg)B_{\vec{i}_1}^\ell(\mathbf{t}).$$

The same process can be applied to the summations in the remaining arguments; the resulting expression is

$$H(\mathbf{t})=\sum_{\vec{i}_1\in\mathbb{I}_k^\ell}\cdots\sum_{\vec{i}_m\in\mathbb{I}_k^\ell}f(\mathbf{G}_{\vec{i}_1},\ldots,\mathbf{G}_{\vec{i}_m})B_{\vec{i}_1}^\ell(\mathbf{t})\cdot\cdots\cdot B_{\vec{i}_m}^\ell(\mathbf{t}),$$

which can be written more compactly using hyperindex notation as

$$H(\mathbf{t})=\sum_{I\in\mathbb{I}_k^{\ell,m}}f(G_I)B_I^{\ell,m}(\mathbf{t}). \tag{9}$$

Using Eq. (2) for the product of Bernstein polynomials, we can write Eq. (9) as

$$H(\mathbf{t})=\sum_{I\in\mathbb{I}_k^{\ell,m}}f(G_I)\mathscr{C}(I)B_{|I|}^{\ell m}(\mathbf{t}). \tag{10}$$

The $\vec{\cdot}$th Bézier coefficient of H is the sum of all terms that multiply $B_{\vec{\cdot}}^{\ell m}(\mathbf{t})$. That is, we seek the points $\mathbf{H}_{\vec{\cdot}}$ such that

$$H(\mathbf{t})=\sum_{\vec{\cdot}\in\mathbb{I}_k^{\ell m}}H_{\vec{\cdot}}B_{\vec{\cdot}}^{\ell m}(\mathbf{t}). \tag{11}$$

These points can be found by grouping together terms in Eq. (10) such that $|I| = \vec{\imath}$, yielding

$$H(\mathbf{t}) = \sum_{\vec{\imath} \in \mathbb{I}_k^{\ell m}} \left(\sum_{\substack{I \in \mathbb{I}_k^{\ell m} \\ |I| = \vec{\imath}}} f(G_I)\mathscr{C}(I) \right) B_{\vec{\imath}}^{\ell m}(\mathbf{t}). \tag{12}$$

The proof is completed by comparing Eqs. (11) and (12) and using the linear independence of the Bernstein polynomials. □

Notice that while F's control points do not appear explicitly in Eq. (8), they are used in the evaluation of f. Since we do not need to know F's representation explicitly, we will generally omit its control points in the statement of the remaining problems. In fact, all that is required for F is the ability to compute arbitrary values of its blossom.

4.2 Composing Bézier Simploids

We now extend the results of the previous section to the three cases where either F or G or both are given in tensor product Bézier form. Rather than considering these cases separately, we shall unify them by formulating and solving a single general problem, which is based on the notion of a *simploid*, a polyhedron formed as the Cartesian product of simplexes [4]. A square is a simple example of a simploid, formed as the Cartesian product of two line segments (i.e., two one-dimensional simplexes). A three-dimensional simploid can be obtained as the Cartesian product of a triangle and a line segment, resulting in a triangular prism. We say that a map Q is a Bézier simploid of degree $d_1 \times d_2$ if its domain is a simploid $\Delta_{\mathscr{X}_1} \times \Delta_{\mathscr{X}_2}$ and if Q is of the form

$$Q(\mathbf{u}, \mathbf{v}) = \sum_{\substack{\vec{\imath} \in \mathbb{I}_{k_1}^{d_1} \\ \vec{\jmath} \in \mathbb{I}_{k_2}^{d_2}}} \mathbf{V}_{\vec{\imath},\vec{\jmath}} B_{\vec{\imath}}^{d_1}(\mathbf{u}) B_{\vec{\jmath}}^{d_2}(\mathbf{v}), \quad (\mathbf{u}, \mathbf{v}) \in \mathscr{X}_1 \times \mathscr{X}_2$$

where $\mathscr{X}_1$ and $\mathscr{X}_2$ are affine spaces of dimension k_1 and k_2, respectively. The points $\mathbf{V}_{\vec{\imath},\vec{\jmath}}$ comprise the control net of Q relative to a domain simploid $\Delta_{\mathscr{X}_1} \times \Delta_{\mathscr{X}_2}$, where $\Delta_{\mathscr{X}_1}$ and $\Delta_{\mathscr{X}_2}$ are simplexes in $\mathscr{X}_1$ and $\mathscr{X}_2$, respectively. Although we shall not do so here, it is straightforward to generalize Bézier simploids further by allowing n-fold Cartesian products, leading to domains such as a simploid $\mathscr{X}_1 \times \cdots \times \mathscr{X}_n$.

Since simploids generalize both simplexes and hypercubes, Bézier simploids generalize both Bézier simplexes and Bézier tensor products. The usual Bézier tensor-product surfaces are obtained when $\mathscr{X}_1$ and $\mathscr{X}_2$ are of dimension one. The Bézier simplexes are obtained when $\mathscr{X}_2$ is of dimension zero.[5]

Many algorithms for tensor products proceed by applying curve algorithms to the rows and columns of the tensor-product control net. For instance, a Bézier tensor-product patch can be degree raised by degree raising each

[5] Strictly speaking, this requires the identification of points in $\mathscr{X}_1 \times \mathscr{X}_2$ with points in $\mathscr{X}_1$ according to $(\mathbf{u}, \mathbf{p}) \mapsto \mathbf{u}$ where $\mathbf{p}$ is the sole point in $\mathscr{X}_2$.

column of the control net using the Bézier curve algorithm, then degree raising each of the resulting rows [9]. A similar, although slightly more complex, procedure holds for composing Bézier simploids F and G. However, the resulting algorithm does not reuse partial results as efficiently as the one we now develop.

Our method uses the tensor-product variant of the blossom [15]. The form of the blossom appropriate for a Bézier simploid Q is a multivariate function q whose arguments are partitioned into two sets, $q(\mathbf{u}_1, \ldots, \mathbf{u}_{d_1}; \mathbf{v}_1, \ldots, \mathbf{v}_{d_2})$, such that: q is multi-affine in each argument; q is symmetric in $\mathbf{u}_1, \ldots, \mathbf{u}_{d_1}$; q is symmetric in $\mathbf{v}_1, \ldots, \mathbf{v}_{d_2}$; and $Q(\mathbf{u}, \mathbf{v}) = q(\mathbf{u}, \ldots, \mathbf{u}; \mathbf{v}, \ldots, \mathbf{v})$. Control points are again obtained by evaluation at vertices of domain simplexes; in particular,

$$\mathbf{V}_{i_\cdot} = q\Big(\underbrace{\mathbf{x}^1_0, \ldots, \mathbf{x}^1_0}_{i_0}, \ldots, \underbrace{\mathbf{x}^1_{k_1}, \ldots, \mathbf{x}^1_{k_1}}_{i_{k_1}}; \underbrace{\mathbf{x}^2_0, \ldots, \mathbf{x}^2_0}_{j_0}, \ldots, \underbrace{\mathbf{x}^2_{k_2}, \ldots, \mathbf{x}^2_{k_2}}_{j_{k_2}}\Big)$$

where $\Delta_{\mathscr{X}_1} = (\mathbf{x}^1_0, \ldots, \mathbf{x}^1_{k_1})$ and $\Delta_{\mathscr{X}_2} = (\mathbf{x}^2_0, \ldots, x^2_{k_2})$.

Since Bézier simploids generalize both tensor products and Bézier simplexes, our unifying problem is to consider the composition of two Bézier simploids F and G.

Given. Affine spaces $\mathscr{X}_1$ (of dimension k_1), $\mathscr{X}_2$ (of dimension k_2), $\mathscr{Y}_1$, $\mathscr{Y}_2$, $\mathscr{Z}$ (of arbitrary dimension), a degree $\ell_1 \times \ell_2$ Bézier simploid $G : \mathscr{X}_1 \times \mathscr{X}_2 \to \mathscr{Y}_1 \times \mathscr{Y}_2$

$$G(\mathbf{u}, \mathbf{v}) = \sum_{\substack{i \in \mathbb{I}^{\ell_1}_{k_1} \\ \cdot \in \mathbb{I}^{\ell_2}_{k_2}}} \mathbf{G}_{i_\cdot} B^{\ell_1}_i(\mathbf{u}) B^{\ell_2}_\cdot(\mathbf{v}) = (G^1(\mathbf{u}, \mathbf{v}), G^2(\mathbf{u}, \mathbf{v})),$$

relative to a domain simploid $\Delta_{\mathscr{X}_1} \times \Delta_{\mathscr{X}_2}$ where $\mathbf{G}_{i_\cdot} = (\mathbf{G}^1_{i_\cdot}, \mathbf{G}^2_{i_\cdot}) \in \mathscr{Y}_1 \times \mathscr{Y}_2$, and a degree $m_1 \times m_2$ Bézier simploid $F : \mathscr{Y}_1 \times \mathscr{Y}_2 \to \mathscr{Z}$.

Find. The control points of the degree $\ell_1(m_1 + m_2) \times \ell_2(m_1 + m_2)$ Bézier simploid $H = F \circ G$ relative to $\Delta_{\mathscr{X}_1} \times \Delta_{\mathscr{X}_2}$.

Solution. If $f(\mathbf{r}_1, \ldots, \mathbf{r}_{m_1}; \mathbf{s}_1, \ldots, \mathbf{s}_{m_2})$ denotes F's blossom, the control points of H are given by

$$H_{i_\cdot} = \sum_{\substack{I^1 \in \mathbb{I}^{\ell_1, m_1}_{k_1}, I^2 \in \mathbb{I}^{\ell_1, m_2}_{k_1} \\ |I^1 I^2| = i_\cdot}} \sum_{\substack{J^1 \in \mathbb{I}^{\ell_2, m_1}_{k_2}, J^2 \in \mathbb{I}^{\ell_2, m_2}_{k_2} \\ |J^1 J^2| = \cdot}} \mathscr{C}(I^1 I^2) \mathscr{C}(J^1 J^2) \cdot f(\mathbf{G}^1_{I^1, J^1}; \mathbf{G}^2_{I^2, J^2}), \tag{13}$$

where $\mathbf{G}^1_{I^1, J^1}$ with $I^1 = (\vec{\imath}^1_1, \ldots, \vec{\imath}^1_{m_1})$ and $J^1 = (\vec{\cdot}^1_1, \ldots, \vec{\cdot}^1_{m_1})$ is an abbreviation for $(\mathbf{G}^1_{i_1 \cdot_1}, \mathbf{G}^1_{i_2 \cdot_2}, \ldots, \mathbf{G}^1_{i_{m_1} \cdot_{m_1}})$; and similarly for $\mathbf{G}^2_{I^2, J^2}$.

PROOF. The proof proceeds by applying the technique of the previous section simultaneously to each of the sets of f's arguments. Into

$$H(\mathbf{u}, \mathbf{v}) = f(G^1(\mathbf{u}, \mathbf{v}), \ldots, G^1(\mathbf{u}, \mathbf{v}); G^2(\mathbf{u}, \mathbf{v}), \ldots, G^2(\mathbf{u}, \mathbf{v})) \tag{14}$$

we substitute the Bernstein form for $G^1(\mathbf{u}, \mathbf{v})$:

$$H(\mathbf{u},\mathbf{v}) = f\left(\sum_{\substack{\vec{i}_1^1 \in \mathbb{I}_{k_1}^{\ell_1} \\ \vec{\ }_1^1 \in \mathbb{I}_{k_2}^{\ell_2}}} \mathbf{G}^1_{\vec{i}_1^1, \vec{\ }_1^1} B^{\ell_1}_{\vec{i}_1^1}(\mathbf{u}) B^{\ell_2}_{\vec{\ }_1^1}(\mathbf{v}), \ldots, \right.$$

$$\left. \sum_{\substack{\vec{i}_{m_1}^1 \in \mathbb{I}_{k_1}^{\ell_1} \\ \vec{\ }_{m_1}^1 \in \mathbb{I}_{k_2}^{\ell_2}}} \mathbf{G}^1_{\vec{i}_{m_1}^1, \vec{\ }_{m_1}^1} B^{\ell_1}_{\vec{i}_{m_1}^1}(\mathbf{u}) B^{\ell_2}_{\vec{\ }_{m_1}^1}(\mathbf{v}); \quad G^2(\mathbf{u},\mathbf{v}), \ldots, G^2(\mathbf{u},\mathbf{v}) \right).$$

Once again, the multi-affine property of f can be exploited to pull the summations outside of f's argument list. Using the hyperindex notation, the resulting expression can be written as

$$H(\mathbf{u},\mathbf{v}) = \sum_{\substack{I^1 \in \mathbb{I}_{k_1}^{\ell_1, m_1} \\ J^1 \in \mathbb{I}_{k_2}^{\ell_2, m_1}}} C(I^1) C(J^1) f(\mathbf{G}^1_{I^1, J^1}; G^2(\mathbf{u},\mathbf{v})) B^{\ell_1 m_1}_{|I^1|}(\mathbf{u}) B^{\ell_2 m_1}_{|J^1|}(\mathbf{v}).$$

A similar substitution is now done for $G^2(\mathbf{u}, \mathbf{v})$. Finally, the control point $\mathbf{H}_{\vec{i},\vec{\ }}$ of H is determined by grouping together all terms in Eq. (14) that multiply $B^{\ell_1(m_1+m_2)}_{\vec{i}}(\mathbf{u}) B^{\ell_2(m_1+m_2)}_{\vec{\ }}(\mathbf{v})$. A straightforward calculation then gives Eq. (13). □

Since the domain of H is the domain of G, H is a Bézier simploid of the same type as G. That is, if G is a simplex (i.e., $\mathscr{X}_2$ is of dimension zero), then H will be in simplex form, regardless of the form of F. Similarly, if G is a tensor-product surface, then the tensor-product form of H will be produced.

4.3 Composing Rational Functions

We now wish to consider composition of rational functions. Recall that, as mentioned in Section 3, we prefer not to work in a coordinate-free framework when dealing with rational functions. The specific problem we address in this section may be stated as follows.

Given. A degree ℓ rational Bézier simplex $G : \mathbb{R}^{k+1} \to \mathbb{R}^{K+1}$ where

$$G(u) = \frac{\hat{G}(\mathbf{t})}{D(\mathbf{t})}.$$

with control points $\{\mathbf{G}_{\vec{i}}\}$ and weights $w_{\vec{i}}$ and a degree m rational Bézier simplex $F : \mathbb{R}^{K+1} \to \mathbb{R}^{L+1}$ with homogenization $\hat{F}$.

Find. The control points and weights of the degree ℓm rational Bézier simplex $H = F \circ G$.

Solution. Let $\hat{f}$ denote the multilinear blossom of $\hat{F}$, and let

$$\hat{H}_{\vec{\ }} = \sum_{\substack{I \in \mathbb{I}_k^{\ell, m} \\ |I| = \vec{\ }}} \mathscr{C}(I) \hat{f}\left(\hat{G}_I\right) = (h_{\vec{\ }0}, \ldots, h_{\vec{\ }L}), \tag{15}$$

where $\cdot \in \mathbb{N}_k$. The weights of H are then given by

$$w_\cdot = h_{\cdot,0} + \cdots + h_{\cdot,L},$$

and the control points are given by

$$\mathbf{H}_\cdot = \mathrm{Proj}(\hat{\mathbf{H}}_\cdot) = \frac{\hat{\mathbf{H}}_\cdot}{w_\cdot}.$$

PROOF. Since F is the ratio of homogeneous polynomials, Eq. (7) can be used to show that

$$H(\mathbf{t}) = F(G(\mathbf{t})) = F\left(\frac{1}{D(\mathbf{t})}\hat{G}(\mathbf{t})\right) = F(\hat{G}(\mathbf{t})).$$

Using the projection operator defined in Eq. (6), it can then be seen that $H(\mathbf{t}) = \mathrm{Proj}(\hat{F}(\hat{G}(\mathbf{t}))$. The function $\hat{H}(\mathbf{t}) = \hat{F}(\hat{G}(\mathbf{t}))$ is a (homogeneous) polynomial of degree ℓm resulting from the composition of the (homogeneous) polynomials $\hat{F}$ and $\hat{G}$. In complete analogy with the proof of Eq. (8), the control points $\hat{H}_\cdot$ can be computed from the multilinear blossom $\hat{f}$ of $\hat{F}$ and the control points $\hat{\mathbf{G}}_{i,\cdot} = w_{i,\cdot}\mathbf{G}_{i,\cdot}$ of $\hat{G}$ according to Eq. (15). The control points $\mathbf{H}_\cdot$ and weights $w_\cdot$ of H can then be obtained from the control points of $\hat{H}$ as shown above. □

To summarize the results of this section, to compose two rational functions F and G, use the polynomial algorithm (together with the multilinear blossom evaluation algorithm) to compute the control points of $\hat{F} \circ \hat{G}$, where $\hat{F}$ and $\hat{G}$ are then homogenizations of F and G. The control points and weights of $H = F \circ G$ can then be recovered by projection.

4.4 Composition of B-splines

In this section we shall sketch the extension of the composition algorithm to B-spline curves and tensor-product B-spline surfaces. These extensions are sufficient to implement the various B-spline applications listed in Section 1. Unfortunately, for the reasons given below, composing B-splines is inherently more complicated than composing Bézier forms.

We begin by considering the composition of B-spline curves. In the following, if Q is a B-spline curve, then Breakpoints(Q) denotes the set of breakpoints of Q, that is, the set of distinct knots of Q.

Given. Two polynomial (or rational) B-spline curves $G : \mathbb{R}^1 \to \mathbb{R}^1$ and $F : \mathbb{R}^1 \to \mathcal{X}$.

Find. The knot vector, B-spline control points (and weights) for $H = F \circ G$.

Solution. One of the complications introduced when dealing with B-splines is that the knot vector appropriate for H must be constructed by suitably merging the knot vectors of G and F. The knot vectors are merged by "pulling back" each of the knots of F into G's domain by finding their pre-images under G. That is, for each knot t_i of F, denote by Pullback(t_i) the

set of all real roots of the equation $t_i = G(u)$. The set Breakpoints(H) is formed by computing the union of Breakpoints(G) with the set

$$\bigcup_{t \in \text{Breakpoints}(F)} \text{Pullback}(t).$$

The breakpoints of H segment H into a sequence of polynomial curve segments. One way to proceed is to:

(1) Convert both F and G to piecewise Bézier form by inserting knots to full multiplicity. For G, knots of full multiplicity must be inserted at each of the breakpoints of H constructed above.
(2) Compute the Bézier control polygon of each segment of H using the Bézier composition algorithm.
(3) Meld the Bézier control polygons (and weights) into a single B-spline representation for H [2, 14, 16].

This is essentially our approach, except that F does not need to be converted to Bézier form. As noted in Section 4.1, the composition algorithm does not use F's Bézier control points directly; it requires only the evaluation of the blossom of F at G's Bézier control points. Fortunately, arbitrary blossom values can be computed directly from the B-spline control points using an algorithm similar to the Cox-deBoor algorithm (cf., [1, 18]). Thus, Eq. (8) can be applied on a segment-by-segment basis to implement step 2 above without converting F to Bézier form.

A similar procedure can be used to find the image of a B-spline curve G on a tensor-product B-spline surface F, as indicated in Figure 1. Once again, the idea is to pull back the knot lines of F to construct the knot vector for H, convert G to piecewise Bézier form, apply Eq. (8) segment by segment, and then use Sablonniere's algorithm [16] to recover the B-spline representation of H. In this case, the knot lines of F are pulled back by finding all parameter values u where $G(u)$ intersects a knot line of F. (If a segment of G coincides with a knot line, that segment can be composed using the univariate technique above.) An example of this procedure is shown in Color Plate 1, where F is a rational B-spline representation of a cylinder, and G is a rational B-spline representation of a full circle in F's domain. The blue and gray spheres denote the B-spline control polygon for the surface curve H (the blue spheres denote control points where segments abut).

It is also desirable to compose two tensor-product B-splines; for instance, one might wish to deform a tensor-product B-spline model G using a tensor-product B-spline deformation F. Unfortunately, the deformed model $H = F \circ G$ cannot, in general, be found in tensor-product B-spline form since it is possible for H to be segmented into a nonrectangular mesh of surface patches. Simply stated, tensor-product B-splines are not closed under composition. For instance, the shaded surface patch shown in Figure 6 has five boundary curves, a situation that is impossible with the tensor-product B-spline form.

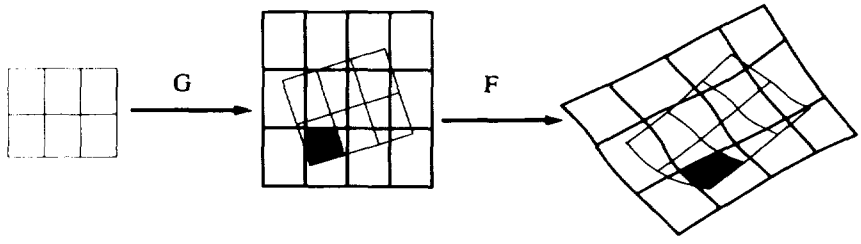


Fig. 6. The composition of two tensor-product B-splines surfaces results in a piecewise surface composed of nonrectangular patches.

To summarize the results of this section, the composition of two B-spline curves F and G can be accomplished by converting G to Bézier form (F does not require conversion), executing the composition algorithm, then using Sablonniere's algorithm to recover the B-spline representation of $F \circ G$.

5. IMPLEMENTATION

An implementation of the polynomial composition algorithm based directly on Eq. (8) is highly inefficient. However, symmetries in the blossom arguments can be exploited to reduce the number of computations, and intermediate results may be reused to further reduce the computational cost. In this section, we give a more efficient, easily codable algorithm for computing the composition of two polynomial functions in Bézier simplex form.

To compute $\mathbf{H}$, Eq. (8) requires the evaluation of $\mathscr{E}(I)f(G_I)$ for all hyperindices $I \in \mathbb{I}_k^{\ell,m}$ where $|I| = \vec{\cdot}$. However, if the tuple J is a permutation of the tuple I, then $\mathscr{E}(I)f(G_I) = \mathscr{E}(J)f(G_J)$ since both $\mathscr{E}$ and f are symmetric. Thus, only one of $\mathscr{E}(I)f(G_I)$ and $\mathscr{E}(J)f(G_J)$ needs to be computed. If $S_k^{\ell,m} \subset \mathbb{I}_k^{\ell,m}$ denotes a set of hyperindices where exactly one permutation of each hyperindex of $\mathbb{I}_k^{\ell,m}$ appears in the set, then Eq. (8) can be rewritten as

$$\mathbf{H} \cdot = \sum_{\substack{I \in S_k^{\ell,m} \\ |I| = \cdot}} P(I)\mathscr{E}(I)f(G_I), \tag{16}$$

where $P(I)$ denotes the number of distinct permutations of hyperindex I in the set $\mathbb{I}_k^{\ell,m}$. If all $\vec{\imath}$'s in $I = (\vec{\imath}_1, \ldots, \vec{\imath}_m)$ are unique, then $P(I) = m!$. A multi-index $\vec{\imath}$ that occurs with multiplicity μ will reduce this by a factor of $\mu!$. Thus, the number of distinct permutations of I is

$$P(I) = \frac{m!}{\Pi_{\vec{\imath}} \mu(I, \vec{\imath})!},$$

where $\mu(I, \vec{\imath})$ is the number of occurrences of $\vec{\imath}$ in I.

In order to iterate through the set $S_k^{\ell,m}$, we impose an ordering on multi- and hyperindices. The ordering we use is the *lexicographic* ordering of tuples [11]: $\vec{\imath} = (i_0, \ldots, i_k) < \vec{\cdot} = (j_0, \ldots, j_k)$ if there is some a, $0 \le a \le k$ such that $i_b = j_b$ for $0 \le b < a$ but $i_a < j_a$. For example, the lexicographic ordering of

the members of $\mathbb{I}_2^2$ is (0, 0, 2) < (0, 1, 1) < (0, 2, 0) < (1, 0, 1) < (1, 1, 0) < (2, 0, 0). The lexicographic ordering on hyperindices is defined similarly. With every $\vec{\imath} \in \mathbb{I}_k^d$, we denote by $\mathrm{Ord}(\vec{\imath})$ the rank order of $\vec{\imath}$, beginning at zero, in the set $\mathbb{I}_k^d$. Thus, Ord(0, 2, 0) = 2 and Ord(0, 0, 2) = 0. By evaluating $C(I)f(G_I)$ only on hyperindices whose multi-indices are in nondecreasing order, i.e., $I = (\vec{\imath}_1, \ldots, \vec{\imath}_m)$ where $\vec{\imath}_j \le \vec{\imath}_{j+1}$, $0 \le j < m$, we avoid re-evaluation on permutations of the same hyperindex.

A second way to reduce the number of computations is to reuse the blossom values of the blossom evaluation algorithm given in Figure 5. For example, the values $f(\mathbf{G}_{i_1}, \mathbf{G}_{i_2}, \mathbf{G}_{i_3}, \mathbf{G}_{i_4})$ and $f(\mathbf{G}_{i_1}, \mathbf{G}_{i_2}, \mathbf{G}_{i_3}, \mathbf{G}_{i_4})$ can both be computed from the control points of the quadratic polynomial $E(\mathbf{u}) = f(\mathbf{G}_{i_1}, \mathbf{G}_{i_2}, \mathbf{u}, \mathbf{u})$. The control points of E are precisely the ones computed by applying *EvalBlossomArgument()* twice, once with the argument $\mathbf{G}_{i_1}$ and once with $\mathbf{G}_{i_2}$. Reusing the results of *EvalBlossomArgument()* can therefore lead to substantial savings. In the code of Figure 7, these results are efficiently reused by simultaneously computing all the control points of H.

The scalar function $\mathscr{C}(I)$, $I = (\vec{\imath}_1, \ldots, \vec{\imath}_m)$, is a ratio of two terms, $\mathscr{C}^N(I) = \binom{|\vec{\imath}_1|}{\vec{\imath}_m} \cdots \binom{|\vec{\imath}_m|}{\vec{\imath}_m}$, and $\binom{\|I\|}{|I|}$. The latter depends on $|I|$ but not on the multi-indices of I. Thus, we can get a further improvement by computing the sum

$$\binom{|\vec{\cdot}|}{\vec{\cdot}}\mathbf{H}_{\vec{\cdot}} = \sum_{\substack{I \in S_k^{\vec{\cdot},m} \\ |I| = \vec{\cdot}}} P(I)\mathscr{C}^N(I)f(G_I). \tag{17}$$

The procedure *Compose()* in Figure 7 incorporates the above optimizations. *Compose()* first calls *RecursiveCompose()*, which recursively computes the values $\binom{|\vec{\cdot}|}{\vec{\cdot}}\mathbf{H}_{\vec{\cdot}}$. After returning from *RecursiveCompose()*, *Compose()* divides by the combinatorial factors $\binom{|\vec{\cdot}|}{\vec{\cdot}}$ to obtain the control points of H.

The procedure *RecursiveCompose()* operates by recursively computing the values $P(I)\mathscr{C}^N(I)f(G_I)$. Its operation is most easily understood if we consider the computation of $I = (\vec{\imath}_1, \ldots, \vec{\imath}_m)$. On entering *RecursiveCompose()*, a prefix of I, $(\vec{\imath}_1, \ldots, \vec{\imath}_n)$, $0 \le n \le m$, will have been computed. The parameter $\vec{m}$ will contain the previous multi-index computed for I. The remaining multi-indices may be no less than this multi-index. *RecursiveCompose()* therefore iteratively sets $\vec{\imath}_{n+1}$ to all multi-indices greater than or equal to $\vec{m}$. For each $\vec{\imath}_{n+1}$, a recursive call is made to compute the remaining multi-indices in I. $\vec{F}$ and the scalar weight are both computed recursively as shown in the code. At the bottom level of the recursion, $P(I)\mathscr{C}^N(I)f(G_I)$ is added to $\mathbf{H}_{|I|}$. The sum $|I|$ is also computed recursively, with the intermediate results being passed in the parameter $\vec{s}$.

Aside from geometric calculations, two things are needed to implement the above scheme: a data structure for storing the control points and a method of iterating over multi-indices in order.

Our data structure for storing the control points of a Bézier simplex consists of a linear array in which control points are stored in lexicographic order. That is, a control point $\mathbf{V}_{\vec{\imath}}$, $\vec{\imath} \in \mathbb{I}_k^d$ is stored at location $\mathrm{Ord}(\vec{\imath})$ within the array. The pseudocode of Figure 8 can be used to compute $\mathrm{Ord}(\vec{\imath})$. The

```
Compose(F, G, H)
begin
        {Initialize H}
        k ← Dimension(G.domain), ℓ ← G.degree, m ← F.degree
        H.degree = ℓ * m
        H.domain = G.domain
        for all ī ∈ I_k^{ℓm}
            H.cp_ī ← 0
        endfor

        F̄ ← Prepare(F) {See Figure 5}
        ī_min ← (0, ..., 0, H.degree)
        RecursiveCompose(F̄, G, H, 0, ī_min, 0, F.degree!, 0)
        for all ī ∈ I_k^{ℓm}
            H.cp_ī ← H.cp_ī / C(ℓm, ī)
        endfor
end

RecursiveCompose(F̄, G, H, n, m̂, ŝ, c, μ)
{n    : recursion control variable}
{m̂    : minimum multi-index value allowed for i_{n+1}}
{ŝ    : sum of multi-indices computed in I thus far}
{c    : scalar to use as weight at bottom of recursion}
{μ    : multiplicity of m̂ in I thus far}
begin
        if n = F̄.degree then
            H.cp_ŝ ← H.cp_ŝ + c * F̄.cp_0^{[n]}
        else
            for all i_{n+1} ∈ I_k^ℓ with i_{n+1} ≥ m̂ in increasing order
                EvalBlossomArgument(F̄, n + 1, G.cp_{i_{n+1}}) {compute F̄^{[n+1]} from F̄^{[n]}}
                if i_{n+1} = m̂ then μ' ← μ + 1 else μ' ← 1
                RecursiveCompose(F̄, G, H, n + 1, i_{n+1}, ŝ + ī_{n+1}, c * C(i_{n+1}, i_{n+1})/μ', μ')
            endfor
        endif
end
```

Fig. 7. The Bézier composition algorithm.

function Size(d, k) referred to in Figure 8 returns the value $\binom{d+k}{k}$, which is the number of elements in $\mathbb{I}_k^d$, or equivalently, the dimension of the space of polynomials in k variables of degree less than or equal to d.

An array used to store points $\mathbf{V}_{\vec{\imath}}$, $\vec{\imath} \in \mathbb{I}_k^d$ is called a simplicial array of dimension k, degree d. In the pseudocode of Figure 5, the control points referred to as $V.\mathbf{cp}_{\vec{\imath}}$ can obviously be stored in a simplicial array of dimension k, degree d. Moreover, a simplicial array of dimension $k + 1$, degree d can be used to store the points $\bar{V}.\mathbf{cp}_{\vec{\imath}}^{[p]}$. In particular, the point $\bar{V}.\mathbf{cp}_{\vec{\imath}}^{[p]}$ can be stored at location Ord(p, $\vec{\imath}$), meaning that the routine *Prepare()* returns a suitably initialized simplicial array.

The procedure *RecursiveCompose()* of Figure 7 requires iteration over multi-indices in increasing lexicographic order. The index succeeding a given multi-index $\vec{\imath} = (i_0, \ldots, i_k)$ can be computed as follows. The indices of $\vec{\imath}$ are scanned from right to left to find the last nonzero index i_p. Next, index i_{p-1}

```
Ord(i⃗)
{Return the rank order of the multi-index i⃗ = (i_0, ..., i_k) ∈ 𝕀_k^d}
begin
        d ← |i⃗|
        ord ← Size(k, d)
        for j = 0 to k − 1 do
            d ← d − i_j
            ord ← ord − Size(k − j, d − 1)
        endfor
        return ord
end
```

Fig. 8. Computation of the offset associated with a multi-index.

```
Successor(i⃗)
{Return the multi-index that succeeds i⃗ = (i_0, ..., i_k).}
begin
        p ← RightMostNonZeroIndex(i⃗)
        if    p = 0 then
              return NoSuccessor
        else
              v ← i_p
              i_{p-1} ← i_{p-1} + 1
              i_p ← 0
              i_k ← v − 1
        endif
        return i⃗
end
```

Fig. 9. Determining the successor of a multi-index.

is incremented; index i_k is set to $i_p - 1$; and index i_p is set to zero. For example, the multi-index following $(i_0, i_1, i_2, i_3, 0, 0, 0)$ is $(i_0, i_1, i_2 + 1, 0, 0, 0, i_3 - 1)$. This process is summarized by the pseudocode of Figure 9.

Note that reusing the partial blossom evaluations is a time-space trade-off. The speed increase comes at the price of higher memory usage. It should also be noted that the number of blossom evaluations performed by the above algorithm (i.e., the number of calls to *EvalBlossomArgument()*) is not minimal. That is, there exist cases where another evaluation order of the values $C(I)f(G_I)$ results in less computation. We chose to use the lexicographical ordering because it simplifies the code for Successor.

The composition algorithm given in Figure 7 is appropriate for composing polynomial functions. As mentioned in Section 4.3, it may also be applied to compose rational functions by using the multilinear blossom evaluation algorithm. The pseudocode of Figure 7 may also be extended to tensor products, although the control flow is more complicated as there are four hyperindices to compute (see Eq. (13)).

6. SUMMARY

We have developed efficient algorithms for computing the Bézier or B-spline representations of $F \circ G$ from the corresponding representations of F and G. A wide variety of modeling problems including evaluation, subdivision, repa-

rameterization, conversion between triangular and tensor-product forms, and explicitly representing models that have undergone free-form deformations can be solved by these algorithms. Since the algorithms can be tightly and efficiently coded, a large amount of software reuse can be achieved by providing these algorithms as library routines.

REFERENCES

1. BARRY, P., AND GOLDMAN, R. Algorithms for progressive curves: Extending B-splines and blossoming techniques to the monomial, power, and Newton dual forms of a curve. In *Knot Insertion and Deletion Algorithm for B-spline Curves and Surfaces*. SIAM, to appear.
2. BARRY, P., AND GOLDMAN, R. Knot insertion algorithms. In *Knot Insertion and Deletion Algorithm for B-spline Curves and Surfaces*. SIAM, to appear.
3. BRUECKNER, I. Construction of Bézier points of quadrilaterals from those of triangles. *Comput. Aided Des. 12*, 1 (1980), 21–24.
4. DAHMEN, W. A., AND MICCHELLI, C. A. On the linear independence of multivariate B-splines, I. Triangulations of simploids. *SIAM J. Numer. Anal. 19*, 5 (Oct. 1982), 993–1012.
5. DE BOOR, C. B-form basics. In *Geometric Modeling: Algorithms and New Trends*, SIAM, 1987, 131–148.
6. DEROSE, T. D. Composing Bézier simplexes. *ACM Trans. Graph. 7*, 3 (July 1988), 198–221.
7. DEROSE, T. D. A coordinate-free approach to geometric programming. In *Math for Siggraph*. ACM Siggraph Course Notes 23, 1989. Also available as Tech. Rep. 89-09-16, Dept. of Computer Science and Engineering, Univ. of Washington, Seattle, Sept. 1989.
8. DEROSE, T. D. Rational Bézier curves and surfaces on projective domains. In *NURBS for Curve and Surface Design*. SIAM, 1991, 35–45.
9. FARIN, G. *Curves and Surfaces for Computer Aided Geometric Design*, 2nd ed. Academic Press, New York, 1990.
10. GOLDMAN, R. N., AND FILIP, D. Conversion from Bézier rectangles to Bézier triangles. *Comput. Aided Des. 19*, 1 (1987), 25–28.
11. KNUTH, D. E. *The Art of Computer Programming*. Vol. 1, *Sorting and Searching*. Addison-Wesley, Reading, Mass., 1969.
12. LOOP, C. T., AND DEROSE, T. D. A multisided generalization of Bézier surfaces. *ACM Trans. Graph. 8*, 3 (July 1989), 204–234.
13. LOOP, C. T., AND DEROSE, T. D. Generalized B-spline surfaces of arbitrary topology. In *SIGGRAPH '90 Proceedings* (1990). ACM, New York, 347–356.
14. LYCHE, T., AND MORKEN, K. Knot removal for parametric B-spline curves and surfaces. *Comput. Aided Geom. Des. 4*, 3 (1987), 217–230.
15. RAMSHAW, L. Blossoming: A connect-the-dots approach to splines. Tech. Rep. 19, Digital Systems Research Center, Palo Alto, Calif., 1987.
16. SABLONNIERE, P. Spline and Bézier polygons associated with a polynomial spline curve. *Comput. Aided Des. 10*, 4 (1978), 257–261.
17. SEDERBERG, T. W., AND PARRY, S. R. Free-form deformation of solid geometric models. In *Proceedings of SIGGRAPH '86* (Aug. 1986). ACM, New York, 151–160.
18. SEIDEL, H.-P. Knot insertion from a blossoming point of view. *Comput. Aided Geom. Des. 5*, 1 (1988), 81–86.
19. WARREN, J. Creating multisided rational Bézier surfaces using base points. *ACM Trans. Graph. 11*, 2 (Apr. 1992), 127–139.

Received May 1991; revised June 1992; accepted August 1992