An Analysis of Polynomial Composition Algorithms

Computer Science Department
University of Waterloo
Research Report CS-95-24

Stephen Mann and Wayne Liu

June 27, 1995

Abstract

An analysis is made of the runtime of a previously published algorithm for polynomial composition. Two new, more efficient algorithms are presented. One of these algorithms is optimal, while the other algorithm is numerically more stable than the optimal one.

Additionally, as a generalization of polynomial composition, we show how to compose a multiaffine function with a set of polynomials as an extension to an earlier algorithm for composing two polynomial functions. With this extension, we are able to perform degree raising with composition.

1 Introduction

Many CAGD operations can be implemented using polynomial composition. As discussed in [2], we can implement evaluation, subdivision, freeform deformation, conversion between tensor product and Bézier simplex forms using polynomial composition. In that paper, an efficient algorithm for composing two polynomials in Bézier simplex form was presented. No analysis was given for the runtime of this algorithm, however.

In Section 3, we analyze the runtime of this composition algorithm. In Section 4, we then present a variation on this algorithm that is more efficient. A second variation that is optimal is presented in Section 5. However, this optimal algorithm is numerically less stable than the non-optimal one, motivating the presentation of both techniques.

Further, while we can recast many CAGD problems as polynomial composition, we have been unable to implement degree raising using polynomial composition. That is, given the control points for a degree n polynomial function $\mathbf{F}$, find the control points for the degree $n+1$ representation of this function. In Section 8, we present a generalization of the composition algorithm for two Bézier simplices, and show how it can be used to degree raise a Bézier simplex.

All the pseudo-code used by the algorithms appears in Appendix B.

2 Notation and Background

In this section we briefly summarize the basic concepts of blossoming and we introduce a number of notational conventions that have been invented to simplify the manipulations needed in Section 8.

The notation used in this paper is a simple extension of the notation in the DeRose et al paper [2] and is summarized in Table 1.

Multi-indices and Hyperindices

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 $\vec{\imath} = (i_0, ..., i_k)$, $i_0, ..., i_k \geq 0$. We shall refer to such tuples as *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 $\vec{\imath} = (i_0, ..., i_k)$ 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 that is 1 in the jth component and zero in the rest of the components; $\vec{0}$ 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 $I = (\vec{\imath}_1, \ldots, \vec{\imath}_m) \in \mathbb{I}_k^{\vec{d},m}$, where $\mathbb{I}_k^{\vec{d},m}$ denotes the m-fold Cartesian product $\mathbb{I}_k^{d_1} \times \cdots \times \mathbb{I}_k^{d_m}$ (for notational convince, we ignore d_0). We shall refer to such tuples as *hyper-indices.* When $d_1 = \ldots = d_m = d$, we denote $\mathbb{I}_k^{\vec{d},m}$ as $\mathbb{I}_k^{d,m}$. Again we 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 $I \in \mathbb{I}_k^{\vec{d},m}$, the expression $||I||$ evaluates to the integer $|\vec{\imath}_1| + \cdots + |\vec{\imath}_m|$. Note that for most of the paper, we will use the $\mathbb{I}_k^{d,m}$ notation. We will not use the $\mathbb{I}_k^{\vec{d},m}$ notation until Section 8.

Bernsteins and Béziers

Using this indexing 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 $\vec{\imath} = (i_0, ..., i_k) \in \mathbb{I}_k^d$, $b_0, ..., b_k \in \mathbb{R}$, 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. It is known [1] that for every polynomial $Q : \mathcal{X} \to \mathcal{Y}$ of degree d, where $\mathcal{X}$ is an affine space of dimension k and $\mathcal{Y}$ is an affine space of arbitrary dimension, there exist unique points $\{\mathbf{V}_{\vec{\imath}}\}_{\vec{\imath} \in \mathbb{I}_k^d}$ in $\mathcal{Y}$ such that

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

where $b_0, ..., b_k$ are the barycentric coordinates of $\mathbf{u} \in \mathcal{X}$ relative to a simplex $\Delta = (\mathbf{x}_0, ..., \mathbf{x}_k)$ of points in $\mathcal{X}$. (For a description of affine spaces, simplices, and barycentric coordinates see, for instance, Farin[5] or DeRose[4].)

A polynomial Q, when expressed as in Equation 1, is called a Bézier simplex; the points $\mathbf{V}_{\vec{\imath}}$ are called the control net of Q with respect to the domain simplex Δ. Note that a degree d Bézier simplex with a domain of dimension k has $|\mathbb{I}_d^k| = \binom{d+k}{k}$ control points [2].

We shall often write $B_{\vec{\imath}}^d(\mathbf{u})$ with the understanding that $\mathbf{u}$ should be replaced with its barycentric coordinates relative to the appropriate domain simplex. If $I = (\vec{\imath}_1, ..., \vec{\imath}_m) \in \mathbb{I}_k^{\vec{d},m}$, we define $B_I^{\vec{d},m}(\mathbf{u})$ to be the product $B_{\vec{\imath}_1}^{d_1}(\mathbf{u}) \cdot \ldots \cdot B_{\vec{\imath}_m}^{d_m}(\mathbf{u})$. We shall make use of the following product relation satisfied by the Bernstein polynomials:

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

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

$$\mathcal{C}(I) = \frac{\begin{pmatrix} |\vec{\imath}_1| \\ \vec{\imath}_1 \end{pmatrix} \cdots \begin{pmatrix} |\vec{\imath}_m| \\ \vec{\imath}_m \end{pmatrix}}{\begin{pmatrix} ||I|| \\ |I| \end{pmatrix}}$$

This relation is easily proved using simple manipulation of the Bernstein polynomials (c.f. [3]).

Blossoms

Ramshaw [6] discovered how to exploit a connection between Bézier simplices and symmetric multi-affine maps. A map $q(\mathbf{u}_1, ..., \mathbf{u}_d)$ 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 $Q : \mathcal{X} \to \mathcal{Y}$ of degree d there is a unique, symmetric, d-affine map, $q : \mathcal{X}^d \to \mathcal{Y}$, that agrees with Q on its diagonal (the diagonal of a multi-affine map $q(\mathbf{u}_1, ..., \mathbf{u}_d)$ is the function obtained when all arguments are equal: $Q(\mathbf{u}) = q(\mathbf{u}, \mathbf{u}, ..., \mathbf{u})$). Ramshaw refers to this multi-affine map q as the *blossom* of Q.

Ramshaw 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 $\mathcal{X}$ is an affine space of dimension k, $Q : \mathcal{X} \to \mathcal{Y}$ a polynomial of degree d whose blossom is q, and $\Delta = (\mathbf{x}_0, ..., \mathbf{x}_k)$ a simplex in $\mathcal{X}$, then Ramshaw has shown that the Bézier control net of Q relative to Δ is given by

$$\mathbf{V}_{\vec{\imath}} = q(\overbrace{\underbrace{\mathbf{x}_0, ..., \mathbf{x}_0}_{i_0}, \underbrace{\mathbf{x}_1, ..., \mathbf{x}_1}_{i_1}, \cdots, \underbrace{\mathbf{x}_k, ..., \mathbf{x}_k}_{i_k}}^{d}).$$

for all $\vec{\imath} = (i_0, ..., i_k) \in \mathbb{I}_k^d$. Using Farin's blossom notation [5], this equation can be written as

$$\mathbf{V}_{\vec{\imath}} = q(\mathbf{x}_0^{<i_0>}, \mathbf{x}_1^{<i_1>}, \cdots, \mathbf{x}_k^{<i_k>}).$$

Polynomial Composition

The basic polynomial composition problem is the following:

> **Given:** Affine spaces $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ (of dimensions K_X, K_Y, and K_Z respectively), control points $\{\mathbf{G}_{\vec{i}}\}_{\vec{i}\in\mathbb{I}^{\ell}_{K_X}}$ defining a Bézier simplex $G : \mathcal{X} \to \mathcal{Y}$ of degree ℓ relative to a domain simplex $\Delta_{\mathcal{X}} \subset \mathcal{X}$, and control points $\{\mathbf{F}_{\vec{p}}\}_{\vec{p}\in\mathbb{I}^{m}_{K_Y}}$ defining a Bézier simplex $F : \mathcal{Y} \to \mathcal{Z}$ relative to a domain simplex $\Delta_{\mathcal{Y}} \subset \mathcal{Y}$.
>
> **Find:** The control points $\{\mathbf{H}_{\vec{j}}\}_{\vec{j}\in\mathbb{I}^{m\ell}_{K_X}}$ of the degree $m\ell$ Bézier simplex $H = F \circ G$ relative to $\Delta_{\mathcal{X}}$.
>
> **Solution:** If f denotes the blossom of $\mathbf{F}$, then
>
> $$\mathbf{H}_{\vec{j}} = \sum_{\substack{I\in\mathbb{I}^{\ell,m}_{K_X} \\ |I|=\vec{j}}} \mathcal{C}(I) f(\mathbf{G}_I), \qquad \vec{j} \in \mathbb{I}^{L}_{K_X} \tag{3}$$
>
> where $\mathbf{G}_I$ with $I = (\vec{i}_1, ..., \vec{i}_m)$ is an abbreviation for $(\mathbf{G}_{\vec{i}_1}, \ldots, \mathbf{G}_{\vec{i}_m})$.

A proof of this result can be found in the paper by DeRose et al [2], and a generalization of this proof can be found in Section 8.

In the following sections, we will make frequent use of the number of $\mathbf{G}$'s control points, which we will denote as $\#G$. Note that $\#G = \binom{\ell+K_X}{K_X}$, with $\mathbf{G}$, K_X, and ℓ defined above.

3 Implementation and Analysis

DeRose et al [2] used symmetry and reuse of intermediate results to reduce the computation cost of computing the control points of $\mathbf{H}$ using Equation 3. In this section, we discuss these two techniques and analyze their runtime behavior. Note that the code in Figures 10, 11, and 12 appeared in [2]. Further note that unless otherwise stated, our algorithms will evaluate the blossom $f(\mathbf{G}_I)$ one argument at a time, in the order given by I.

de Casteljau's Algorithm

Bézier curves and surfaces are commonly evaluated using de Casteljau's algorithm. This algorithm uses repeated linear interpolation to evaluate Bézier simplices. In addition to evaluating the Bézier simplex for position, the intermediate results can be used to compute derivatives. Further, the algorithm can be extended to evaluate a single argument of the blossom. See Farin's book [5] for more details on de Casteljau's algorithm.

Note that the intermediate values used in a de Casteljau evaluation of a degree d, dimension k Bézier simplex can be stored in simplex of one higher dimension (i.e., with $\binom{d+k+1}{k+1}$ storage). Further, each point in this simplex (other than the initial control points) is computed using one linear combination. Thus, a full de Casteljau evaluation requires $\binom{d+k}{k+1}$ linear combinations during its computation.

Notation	Description
$\mathbb{R}$	The set of real numbers.
$\mathbb{R}^n$	n-dimensional real space.
$a, ..., z$	Integers and real numbers.
$\mathcal{X}$, $\mathcal{Y}$, etc.	Affine spaces.
K_X	The dimension of the affine space $\mathcal{X}$
$\mathbf{a}, ..., \mathbf{z}$	Affine points.
Δ	Domain simplex.
$\vec{a}, ..., \vec{z}$	Multi-indices.
$\lvert\vec{\imath}\rvert$	$= i_0 + \cdots + i_k$ where $\vec{\imath} = (i_0, ..., i_k)$.
$\mathbb{I}_k^d$	The set of all multi-indices $\vec{\imath} = (i_0, ..., i_k)$ with $\lvert\vec{\imath}\rvert = d$.
I, J, I^1, etc.	Hyper-indices.
$\mathbb{I}_k^{d,m}$	The set of all hyper-indices $I = (\vec{\imath}_1, ..., \vec{\imath}_m)$ with $\vec{\imath}_1, ..., \vec{\imath}_m \in \mathbb{I}_k^d$.
$\mathbb{I}_k^{\vec{d},m}$	The set of all hyper-indices $I = (\vec{\imath}_1, ..., \vec{\imath}_m)$ with $\vec{\imath}_j \in \mathbb{I}_k^{d_j}$.
$\lvert I\rvert$	$= \vec{\imath}_1 + \cdots + \vec{\imath}_m$ where $I = (\vec{\imath}_1, ..., \vec{\imath}_m)$.
$\lvert\lvert I\rvert\rvert$	$= dm$ where $I \in \mathbb{I}_k^{d,m}$.
$\binom{\lvert\vec{\imath}\rvert}{\vec{\imath}}$	The multinomial coefficient $\frac{\lvert\vec{\imath}\rvert}{i_0! \cdots i_k!}$, $\vec{\imath} = (i_0, ..., i_k)$.
$B_{\vec{\imath}}^d(\mathbf{u})$	The $\vec{\imath}$-th Bernstein polynomial of degree d.
$B_I^{d,m}$	$= B_{\vec{\imath}_1}^d(\mathbf{u}) \cdots B_{\vec{\imath}_m}^d(\mathbf{u})$, $I = (\vec{\imath}_1, ..., \vec{\imath}_m)$.
$\mathcal{C}(I)$	A combinatorial constant given by $\frac{\binom{\lvert\vec{\imath}_1\rvert}{\vec{\imath}_1} \cdots \binom{\lvert\vec{\imath}_m\rvert}{\vec{\imath}_m}}{\binom{\lvert\lvert I\rvert\rvert}{\lvert I\rvert}}$.
$q(\mathbf{u}_1, ..., \mathbf{u}_d)$	Blossom of a Bézier simplex $Q(\mathbf{u})$ of degree d.
$f(\mathbf{G}_I)$	$= f(\mathbf{G}_{\vec{\imath}_1}, \ldots, \mathbf{G}_{\vec{\imath}_d})$ with $I \in \mathbb{I}_k^{d,m}$.
$\#G$	$= \binom{\ell + K_X}{K_X}$, the number of control points of $\mathbf{G}$.

Table 1: Summary of notation.

In the initial analysis of runtime, we will count the number of linear combinations performed by the composition algorithms. A more complete runtime analysis can be found in Section 6, where we show that the cost of the more sophisticated algorithms is proportional to the number of linear combinations performed.

In Figure 10, we present some code for performing partial blossom evaluations. All the linear combinations are performed in `EvalBlossomArgument()`, which evaluates one blossom argument by performing a single step of de Casteljau's algorithm.

3.1 Naive Algorithm

A naive implementation of the composition algorithm would compute each $\mathbf{H}_{\vec{i}}$ by performing a full evaluation of f for every possible hyperindex $I \in \mathbb{I}^{\ell,m}_{K_X}$ where $|I| = \vec{i}$. Note that this means we will be evaluating f exactly once for each hyperindex I. As mentioned earlier, a full de Casteljau evaluation requires $\binom{m+K_Y}{K_Y+1}$ linear combinations. Since the number of multi-indices of dimension K_X and degree ℓ is $\binom{\ell+K_X}{K_X} = \#G$, the number of hyper-indices will be $\#G^m$. Thus, this algorithm requires

$$\binom{m+K_Y}{K_Y+1}\#G^m \tag{4}$$

linear combinations. We shall refer to this algorithm as the *Naive Algorithm.*

3.2 One Permutation Algorithm

The first way to improve the Naive Algorithm is to use the symmetry property of blossoms. This property allows us to evaluate f only at hyperindices that are unique up to permutations. Using such hyperindices, we only care how many times each multi-index $\vec{i}_j$ appears in I, not where each $\vec{i}_j$ appears. Thus, we can convert the hyperindex I to a multi-index $\vec{i}$, where i_j is the number of times $\vec{i}_j$ appears in I. The dimension of $\vec{i}$ is one less than the number of $\mathbf{G}$'s control points, and the degree is the number of multi-indices in the hyperindex I (which, in the case we are considering, is m). Thus, $\vec{i} \in \mathbb{I}^m_{\#G-1}$. For example, if $\mathbf{G}$ has five control points, $G_{\vec{i}_0}, \ldots, G_{\vec{i}_4}$, and if $\mathbf{F}$ is fourth degree, then the hyperindex $I = (\vec{i}_0, \vec{i}_0, \vec{i}_0, \vec{i}_2)$ can be thought of as the multi-index $\vec{i} = (3,0,1,0,0) \in \mathbb{I}^4_4$.

Normally, we prefer to think of hyperindices and multi-indices as different, since they are used in different ways and if we care about the order of the multi-indices within the hyperindex, then the above correspondence fails to hold. However, such a correspondence is useful in counting the number of hyperindices that are unique up to permutations, which is

$$|\mathbb{I}^m_{\#G-1}| = \binom{\#G-1+m}{\#G-1}.$$

Thus, by evaluating at hyperindices that are unique up to permutations, the number of linear evaluations required by the composition algorithm is reduced to

$$\binom{m+K_Y}{K_Y+1}\binom{\#G-1+m}{\#G-1}. \tag{5}$$

Note that to compute the $\mathbf{H}_{\vec{\imath}}$, we will need to weight each $f(G_I)$ by the number of permutations of the multi-indices within I [2]. We shall refer to this algorithm that only evaluates $f(G_I)$ once for all permutations of I but makes a full de Casteljau evaluation for each of these G_Is as the *One Permutation Algorithm.*

Comparison of Naive and One Permutation Algorithms

For a comparison of the runtime of the two algorithms, we need to compare Equation 4 to Equation 5. Canceling the common factor of $\binom{m+K_Y}{K_Y+1}$, we have

$$\begin{aligned} \#G^m \quad vs \quad & \binom{\#G-1+m}{m} \\ \#G^m \quad vs \quad & \frac{[\#G+m-1]!}{m!(\#G-1)!} \\ \#G^m \quad vs \quad & \frac{\prod_{i=0}^{m-1}[\#G+i]}{m!} \end{aligned}$$

Alternatively, we could have chosen to cancel the $m!$ term, giving us

$$\#G^m \quad vs \quad \frac{\prod_{i=1}^{\#G-1}[m+i]}{(\#G-1)!}$$

If we fix K_Y and m and allow $\#G$ to increase, then the former is the appropriate equation, and we see that asymptotically we achieve a factor of $m!$ speedup. If we fix K_Y and $\#G$ and allow m to increase, then the latter is the appropriate formula, and we see that we asymptotically a speedup of

$$\frac{\#G^m(\#G-1)!}{m^{\#G}}$$

which is exponential in m.

Note that K_Y appears in neither the left or right side of our equation, so we have a "constant" speed-up factor of

$$\frac{\#G^m}{\binom{\#G-1+m}{m}}$$

as K_Y increases while m and $\#G$ remain constant.

3.3 The 1993 Algorithm

Now suppose we use the algorithm of DeRose et al (i.e., reusing partial blossom calculations). This code appears in Figure 12, with some supporting code (to initialize $\mathbf{H}$) in Figure 11. By construction, this algorithm iterates over hyperindices $I \in \mathbb{I}_{K_X}^{\ell,m}$ in lexicographical order by first selecting a multi-index $\vec{\imath}$ and partially evaluating f at $\mathbf{G}_{\vec{\imath}}$. This partial evaluation is used to compute $f(\mathbf{G}_I)$ for all hyperindices beginning with $\vec{\imath}$. The algorithm selects (and evaluates at) the remaining multi-indices recursively (each of which must be lexicographically equal to or larger than $\vec{\imath}$), until we have fully evaluated f. We shall refer to this algorithm as the *1993 Algorithm.*

We will now derive the runtime of the 1993 Algorithm. Suppose the algorithm has continued for k steps and it has selected multi-indices $\vec{i}_1, \ldots, \vec{i}_k$ where $\vec{i}_1 \leq \ldots \leq \vec{i}_k$. At this point the algorithm will have evaluated f at $\mathbf{G}_{\vec{i}_1}, \ldots, \mathbf{G}_{\vec{i}_k}$ and the calls to `EvalBlossomArgument()` will have computed the blossom values

$$f(\mathbf{G}_{\vec{i}_1}, \ldots, \mathbf{G}_{\vec{i}_k}, \mathbf{y}_0^{<j_0>}, \ldots, \mathbf{y}_{K_Y}^{<j_{K_Y}>}), \tag{6}$$

for all $\vec{j} \in \mathbb{I}_{K_Y}^{m-k}$. Note that `EvalBlossomArgument()` computes each of these blossom values with a single linear combination. If we relabel $\mathbf{G}$'s control points from 0 to $\#G-1$, then we can rewrite the blossom values of Equation 6 in the form

$$f(\mathbf{G}_0^{<i_0>}, \ldots, \mathbf{G}_{\#G-1}^{<i_{\#G-1}>}, \mathbf{y}_0^{<i_{\#G}>}, \ldots, \mathbf{y}_{K_Y}^{<i_{\#G+K_Y}>}), \tag{7}$$

where $\vec{i} = (i_0, \ldots, i_{\#G+K_Y}) \in \mathbb{I}_{\#G+K_Y}^m$. Further, the process of the 1993 Algorithm described in the previous paragraph ensures that all $\vec{i}_1, \ldots, \vec{i}_k$ for $k = 1..m$ where $\vec{i}_1 \leq \ldots \leq \vec{i}_k$ are selected exactly once. Therefore, the algorithm will compute the blossom values of Equation 7 exactly once for each $\vec{i}$ in $\mathbb{I}_{\#G+K_Y}^m$.

Thus, we can think of the blossom values of Equation 6 to be embedded in a simplicial array of dimension $\#G + K_Y$ and degree m. For each point in this simplicial array (except for the control points of $\mathbf{F}$), we perform one linear evaluation, and so the 1993 Algorithm performs

$$\binom{m+\#G+K_Y}{m} - \binom{m+K_Y}{m}. \tag{8}$$

linear combinations.

An example appears in Figure 1. In this figure, $m = 3$, $K_Y = 1$, and $\mathbf{G}$ has three control points (which we will subscript 0, 1, and 2). We embed the composition algorithm in a degree 3, dimension 4 simplicial array. Along one edge, we place $\mathbf{F}$'s control points (drawn as black dots). Each time we evaluate f at $\mathbf{G}_0$, we move "up" one simplex. For each argument we evaluate at $\mathbf{G}_1$, we move up one level within the current simplex. And for the remaining arguments (which are all $\mathbf{G}_2$), we finish evaluating the blossom in the remaining dimension, giving one of the contributing points to $\mathbf{H}$'s control points (drawn as a hollow circle in the figure). Note that the results we care about lie along one "face" of this high dimensional simplicial array.

Comparison of the One Permutation and 1993 Algorithms

We will now compare the number of linear combinations used by the One Permutation Algorithm (given by Equation 5) to the number of linear combinations used by the 1993 Algorithm (Equation 8):

$$\begin{aligned}
\binom{m+K_Y}{K_Y+1}\binom{\#G-1+m}{m} \quad &vs \quad \binom{m+\#G+K_Y}{m} - \binom{m+K_Y}{m} \\
\binom{m+K_Y}{K_Y+1}\frac{(m+\#G-1)!}{m!(\#G-1)!} \quad &vs \quad \frac{(m+\#G-1)!\prod_{i=0}^{K_Y}(m+\#G+i)}{m!(\#G-1)!\prod_{i=0}^{K_Y}(\#G+i)} - \binom{m+K_Y}{m} \\
\binom{m+K_Y}{K_Y+1} = \frac{(m+K_Y)!}{(m-1)!(K_Y+1)!} \quad &vs \quad \frac{\prod_{i=0}^{K_Y}(m+\#G+i)}{\prod_{i=0}^{K_Y}(\#G+i)} - \frac{\binom{m+K_Y}{m}}{\binom{\#G-1+m}{m}}
\end{aligned} \tag{9}$$

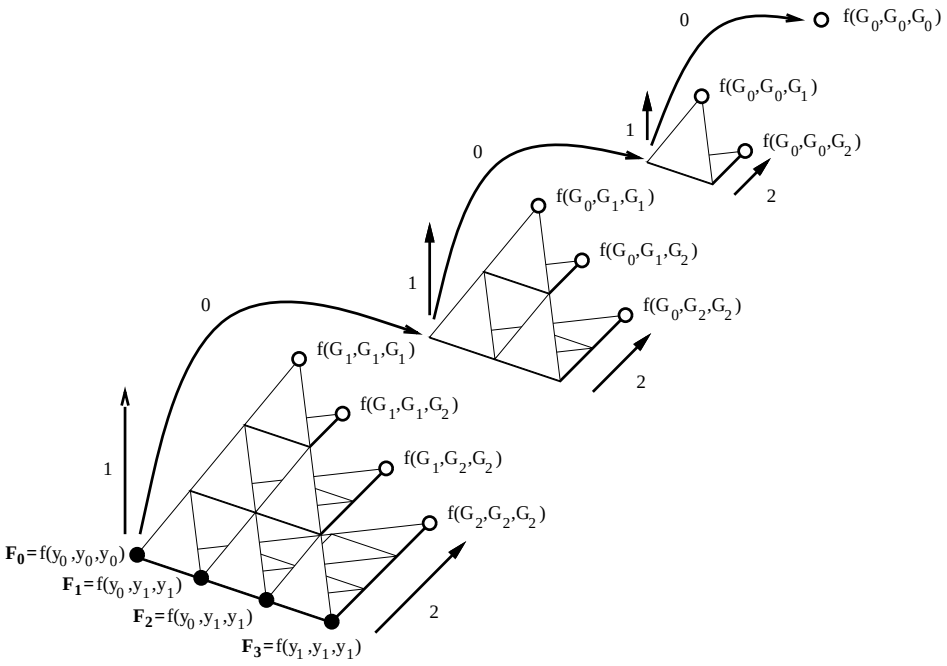

Figure 1: Evaluations performed by 1993 Algorithm

In a similar fashion, we can instead cancel a factor of $\binom{m+K_Y}{K_Y+1}$ from both sides, leaving

$$\binom{\#G-1+m}{m} = \frac{(\#G-1+m)!}{m!(\#G-1)!} \quad vs \quad \frac{\prod_{i=1}^{\#G}(m+K_Y+i)}{m\prod_{i=2}^{\#G}(K_Y+i)} - \frac{K_Y+1}{m} \tag{10}$$

We now consider the following three cases:

- $\#G$ grows while K_Y and m are fixed. In this case, we see that the limit of the right hand term of Equation 9 is 1. Thus, the 1993 Algorithm performs better than the One Permutation Algorithm by a factor of

  $$\binom{m+K_Y}{m-1}.$$

  This can be seen in the graph on the left in Figure 2. The theoretical asymptote is 15 for $K_Y = 3$ and $m = 3$. While the apparent asymptote in this figure appears to be about 14, note that the term on the right of Equation 9 is about 1.08. As $\#G$ increases further, the ratio of the runtimes becomes closer to 15.

- m grows while K_Y and $\#G$ are fixed. For large m, Equation 9 reduces to

  $$\frac{m^{K_Y+1}}{(K_Y+1)!} \quad vs \quad \frac{m^{K_Y+1}}{\prod_{i=0}^{K_Y}(\#G+i)} - \frac{(m+K_Y)!(\#G-1)!}{K_Y!(\#G-1+m)!}$$

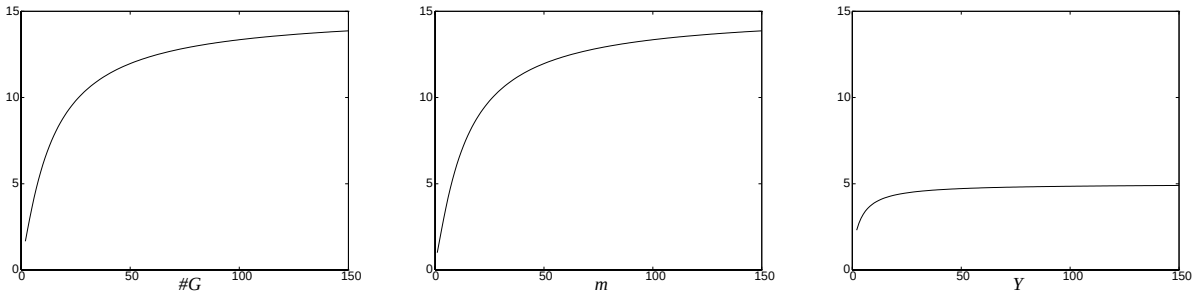

Figure 2: Comparison of the One Permutation Algorithm and the 1993 Algorithm. Graphs showing the ratio of the runtime of the One Permutation Algorithm to the runtime of the 1993 Algorithm. The graph on the left shows the ratio as we increase $\#G$ for $K_Y = 3$ and $m = 3$. The graph in the middle shows the ratio as we increase m for $\#G = 3$ and $K_Y = 3$. The graph on the right shows the ratio as we increase K_Y for $\#G = 4$ and $m = 3$.

Note that the subtrahend on the left side of the equation is never larger (in terms of m) than $O(m^{K_Y - \#G + 1})$, so this equation simplifies to

$$\frac{m^{K_Y+1}}{(K_Y+1)!} \quad vs \quad \frac{m^{K_Y+1}}{\prod_{i=0}^{K_Y}(\#G+i)}$$

Thus, the 1993 Algorithm is faster by a factor of

$$\frac{\prod_{i=0}^{K_Y}(\#G+i)}{(K_Y+1)!} = \binom{\#G+K_Y}{K_Y+1}$$

This effect can be seen in the middle graph of Figure 2. Again, our theoretical asymptote is 15, although m would have to be much larger for the graph to approach this asymptote. Note that this result can also be derived from Equation 10.

- K_Y grows while m and $\#G$ remain fixed. Here we work with Equation 10. To reduce this equation, we have to determine the limit of the right hand side of the equation as K_Y goes to infinity:

$$\begin{aligned}
&\lim_{K_Y\to\infty} \frac{\prod_{i=1}^{\#G}(m+K_Y+i)}{m\prod_{i=2}^{\#G}(K_Y+i)} - \frac{K_Y+1}{m} \\
&= \lim_{K_Y\to\infty} \frac{\prod_{i=1}^{\#G}(m+K_Y+i) - \prod_{i=1}^{\#G}(K_Y+i)}{m\prod_{i=2}^{\#G}(K_Y+i)} \\
&= \frac{\left[K_Y^{\#G} + (\#Gm + \sum_{i=1}^{\#G} i)K_Y^{\#G-1}\right] - \left[K_Y^{\#G} + (\sum_{i=1}^{\#G} i)K_Y^{\#G-1}\right]}{mK_Y^{\#G-1}} \\
&= \#G
\end{aligned}$$

Thus, 1993 Algorithm is faster than the One Permutation Algorithm by a factor of

$$\frac{\binom{m+\#G-1}{m}}{\#G}$$

An example is seen in the graph on the right in Figure 2 where our asymptote is 5.

Note that the above derivation means that as K_Y increases to infinity, the runtime of the 1993 Algorithm simplifies to

$$\#G \cdot \binom{m + K_Y}{K_Y + 1}$$

4 Improved Algorithm

In this section, we give an improvement on the 1993 Algorithm. Basically, we find an ordering of the multi-indices that makes better use of the partial blossom evaluations.

We begin by noting that any algorithm based on Equation 3 must compute $f(G_{\vec{i}}^{<m>})$ for all $\vec{i}$. Thus, we should try to maximize the use of these evaluations. If we consider de Casteljau's algorithm, the highest cost evaluations of the blossom arguments are the first ones. To optimize our computation, we should try to use these as much as possible. Therefore, we will order the evaluation of the blossom in nonincreasing order of the multiplicity of each argument. I.e, we only evaluate f at

$$f(G_{\vec{i}_1}^{<i_1>}, \ldots, G_{\vec{i}_k}^{<i_k>}),$$

where $i_1 \geq i_2 \geq \ldots \geq i_k$ and $i_1 + \ldots + i_k = n$. Further, to ensure that we only evaluate once for all permutations of a set of arguments, we will require that if $i_j = i_{j+1}$ then $\vec{i}_j < \vec{i}_{j+1}$. We shall refer to this algorithm as the *Improved Algorithm.*

Pseudo-code for this algorithm appears in Figure 13. We have to be careful with the minimum number of times we evaluate at an argument, otherwise we might perform extra evaluations. Also, left unspecified in our pseudo-code is how to compute '$ngt()$' and '$nlt()$' (which count the number of unmarked multi-indices in a set greater than and less than the given multi-index). We have to be careful how we compute these values or we will add to the Big-O runtime of the algorithm.

An example of the operation of this algorithm appears in Figure 3. Again, the initial control points of $\mathbf{F}$ are drawn in black. The control points needed by the composition algorithm are drawn as hollow dots. All of the lines appearing in Figure 1 are drawn in this figure, but ones in regions where no linear combinations were computed are drawn as dotted lines. Further, some additional lines between the simplices have been drawn to indicate linear combinations used by the Improved Algorithm that are not used by the 1993 Algorithm. Also note that the order of some of the arguments of these blossom values has changed to reflect the new order of evaluation. Finally, note that the Improved Algorithm does not compute four blossom values computed by the 1993 Algorithm ($f(\mathbf{G}_1, \mathbf{G}_2, \mathbf{y}_0)$, $f(\mathbf{G}_1, \mathbf{G}_2, \mathbf{y}_1)$, $f(\mathbf{G}_0, \mathbf{G}_2, \mathbf{y}_0)$, and $f(\mathbf{G}_0, \mathbf{G}_2, \mathbf{y}_0)$).

It is unclear what the run-time of this algorithm is. However, note that it has the same run time as the 1993 Algorithm if the degree of $\mathbf{F}$ is two or less. Empirical testing shows that while the Improved Algorithm is faster than the 1993 Algorithm, it is not significantly faster until m becomes large (Figures 6-9). It should be noted, though, that if $K_Y > \#G$ and m is large, then the Improved Algorithm performs much better than the 1993 Algorithm, and almost as good as the Optimal Algorithm described in the next section (see Figure 9).

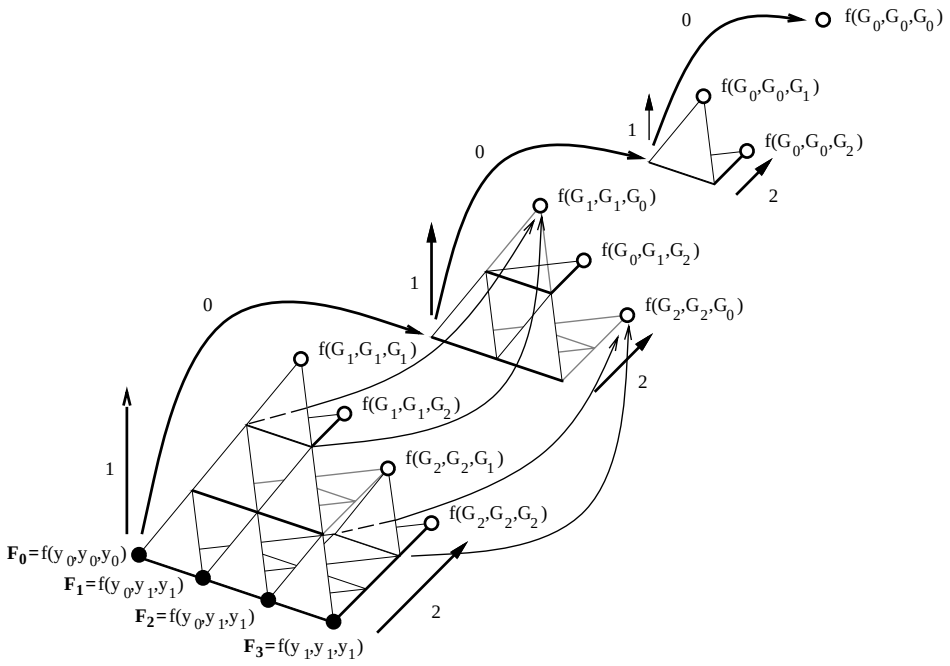

Figure 3: Evaluations performed by the Improved Algorithm

5 An Optimal Algorithm

In this section we present a composition algorithm that is optimal in a Big-O sense in that for each I at which Equation 3 requires us to evaluate $f(\mathbf{G}_I)$, this new algorithm performs one linear evaluation (with some additional linear evaluations required for an initial change of basis). The key observation is the following: If we reparameterize $\mathbf{F}$ over a simplex consisting of $\mathbf{G}$'s control points, then all of the control points of $\mathbf{F}$ are values of the form $f(\mathbf{G}_I)$ and can be used in computing $\mathbf{H}$'s control points. Working with a representation of $\mathbf{F}$ in this basis, every intermediate point computed by de Casteljau's algorithm when evaluating at a set of $\mathbf{G}$'s control points gives us one of the $f(\mathbf{G}_I)$ we need to compute the control points of $\mathbf{H}$.

Before presenting the algorithm, we first note that no algorithm based on Equation 3 can perform fewer than

$$\binom{\#G + m - 1}{m}$$

linear combinations, as this is the number of $I \in \mathbb{I}_{K_X}^{\ell,m}$ that are unique up to permutations (i.e., the number of times we must evaluate f when computing the control points of $\mathbf{H}$).

We can very nearly achieve this runtime by first converting $\mathbf{F}$ to a representation relative to a subset of $\mathbf{G}$'s control points (that lie in general position). Then, we iterate over $\mathbf{G}$'s remaining control points with an algorithm similar to the original composition algorithm. At each partial evaluation, every point in the de Casteljau diagram is used in the computation of $\mathbf{H}$'s control points (Figure 5, when evaluating at $\mathbf{G}_0$). Further, each such point appears exactly once in the

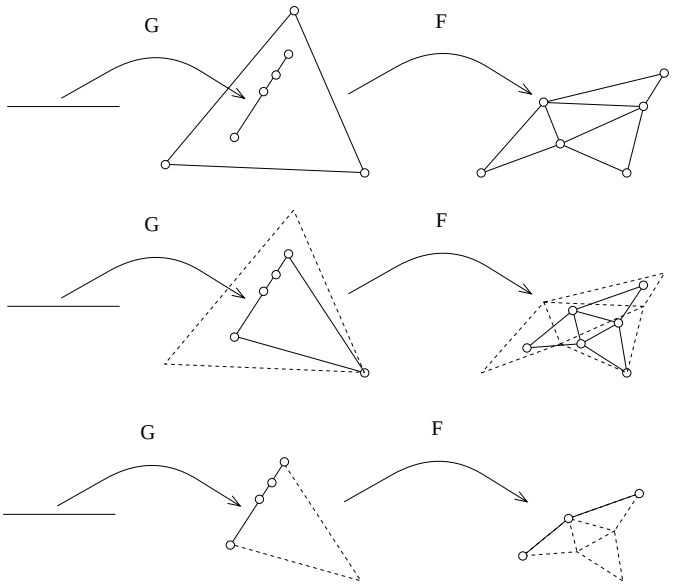

Figure 4: If **G** only spans a subdomain of **F**'s domain, we perform composition with the restriction of **F** to **G**'s range.

computation. The only points we compute that are not needed for computing **H** are those in the initial basis conversion. Pseudo-code for this algorithm appears in Figures 14, 15, and 16. Note that the changes to 1993 version of `RecursiveCompose()` are minor: A call to `ExtractCPs()` is inserted before the `if` statement, and the code inside the `if` statement is reduced to a `return` statement. We shall refer to this algorithm as the *Optimal Algorithm.*

As written, the routine `ConvertBasis()` is not always correct. Potentially, the subset of **G**'s control points selected by the routine might not form a proper simplex. A correct version must select $K_Y + 1$ of **G**'s control points that form a proper simplex in $\mathcal{Y}$, evaluate at these points, and continue processing from there. This adds a slight complication: The remainder of the composition algorithm needs to iterate over **G**'s remaining control points (i.e., those not used in the basis conversion).

There are a couple of approaches for handling this problem. One approach is to mark the control points selected for the new basis of **F** and skip them in the loop in `RecursiveCompose()` of Figure 14, with $\vec{j}_{\min}$ starting at **G**'s first control point. A second approach is to create a linear list of multi-indices into **G**'s control points. Then, before converting basis, a subset of **G**'s control points would be selected and moved to the beginning of the array. With this latter approach, the loop control of the `for` loop of `RecursiveCompose()` would change to be linear offsets into this list of multi-indices.

Also, note that if the control points of **G** do not span the domain of **F**, then we should partially reparameterize **F** over a subdomain spanned by **G**'s control points, extract the control points of **F** over this subdomain, and proceed with the algorithm from this point (Figure 4).

An example of the operation of the Optimal Algorithm appears in Figure 5. To maintain similarity to Figures 1 and 3, we have select $\mathbf{G}_2$ and $\mathbf{G}_1$ as our basis for **F**, and after converting to

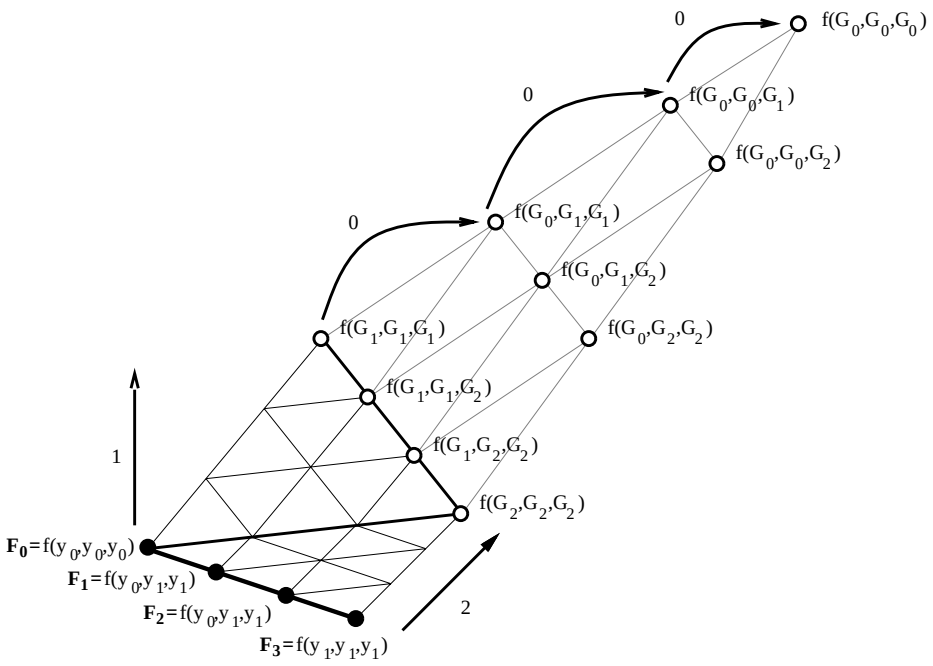

Figure 5: Evaluations performed by the Optimal Algorithm

this basis, we evaluate at $\mathbf{G}_0$. Again, the initial control points are drawn in black. In this figure the algorithm begins by fully evaluating $\mathbf{F}$ at $\mathbf{G}_2$. Then, starting from the points $f(\mathbf{y}_0, \mathbf{y}_0, \mathbf{y}_0)$, $f(\mathbf{y}_0, \mathbf{y}_0, \mathbf{G}_2)$, $f(\mathbf{y}_0, \mathbf{G}_2, \mathbf{G}_2)$, $f(\mathbf{G}_2, \mathbf{G}_2, \mathbf{G}_2)$, we fully evaluate at $\mathbf{G}_1$. This completes the basis conversion. Note that the linear combinations used for the basis convertion are indicated with solid lines. The Optimal Algorithm then performs the 1993 Algorithm with the remaining control point. In this case, we only have one more control points, so we get a third complete de Casteljau evaluation. Note, however, that every point computed in this de Casteljau evaluation is a point needed by the composition algorithm (drawn as hollow dots in the figure). The linear combinations used by the rest of the algorithm are drawn as dotted lines in the figure.

It should be cautioned that the example appearing in Figure 5 is a bit simplistic. If $\#G$ is larger than three, then the dotted line portion of the figure increases in dimension. For example, if $\#G = 4$, then while the solid line portion of the figure remains unchanged, the dotted portion becomes a degree three tetrahedral array. Note that the corresponding simplices for the 1993 and Improved algorithms also increase by one dimension, so the computation of the Optimal algorithm still fits in a smaller space than the other two algorithms.

The number of linear evaluations of the Optimal Algorithm is given by the number of locations at which we need to evaluate f plus the cost of the initial basis conversion:

$$\binom{\#G + m - 1}{m} + (K_Y + 1)\binom{m + K_Y}{K_Y + 1} - \binom{m + K_Y}{K_Y}. \tag{11}$$

The last term accounts for the representation of $\mathbf{F}$ in a basis of $\mathbf{G}$'s control points, a quantity that has been counted in both the other two terms. Note that if the number of $\mathbf{G}$'s control points is less

than or equal to the dimension of $\mathcal{Y}$, then the runtime simplifies to

$$\#G \cdot \binom{m + K_Y}{K_Y + 1},$$

since we will obtain all the values of f we need during the basis conversion of $\mathbf{F}$. In this case, the algorithm is no longer optimal, although it still has better runtime performance than the other algorithms.

Comparison of 1993 and Optimal Algorithms

To compare the runtime of the Optimal Algorithm to the 1993 Algorithm, we will look at what happens when $\#G > K_Y$ (in which case the first term of Equation 11 will dominate) and when $K_Y > \#G$ (in which case, the cost is entirely due to the basis conversion).

We begin by deriving an expression for the ratio of runtimes when $\#G > K_Y$ (which we normally expect). In this situation, the subtrahend of both equations for the runtime does not contribute to the asymptotic behavior for reasons given in Section 3. Thus, we compare $\binom{m+\#G+K_Y}{m}$ (the minuend of the runtime of the 1993 Algorithm) to $\binom{\#G+m-1}{m}$ (the first term of the runtime of the Optimal Algorithm):

$$\begin{aligned}
\binom{m + \#G + K_Y}{m} &= \frac{(m + \#G + K_Y)!}{m!(\#G + K_Y)!} \\
&= \frac{(\#G + m - 1)! \prod_{i=0}^{K_Y} (m + \#G + i)}{m!(\#G - 1)! \prod_{i=0}^{K_Y} (\#G + i)} \\
&= \binom{\#G + m - 1}{m} \frac{\prod_{i=0}^{K_Y} (m + \#G + i)}{\prod_{i=0}^{K_Y} (\#G + i)}.
\end{aligned}$$

Thus we see that the runtime of the 1993 Algorithm is a factor of

$$\frac{\prod_{i=0}^{K_Y} (m + \#G + i)}{\prod_{i=0}^{K_Y} (\#G + i)} \tag{12}$$

slower than the Optimal Algorithm.

To do a complete comparison of the runtimes, we consider four cases:

- If we fix $\#G$ and K_Y with $\#G > K_Y$ and let m vary, then the denominator of Equation 12 is constant and the dominant term in the numerator is m^{K_Y} (i.e., the Optimal Algorithm is a polynomial factor better than the 1993 Algorithm as we increase m). Graphs showing this behavior appear in Figure 6.

- If we fix K_Y and m and allow $\#G$ to vary, then again working with Equation 12 we see that asymptotically the two algorithms have the same runtime behavior. This behavior is visible in Figure 7, although we have to let $\#G$ become large before it is apparent in the graph.

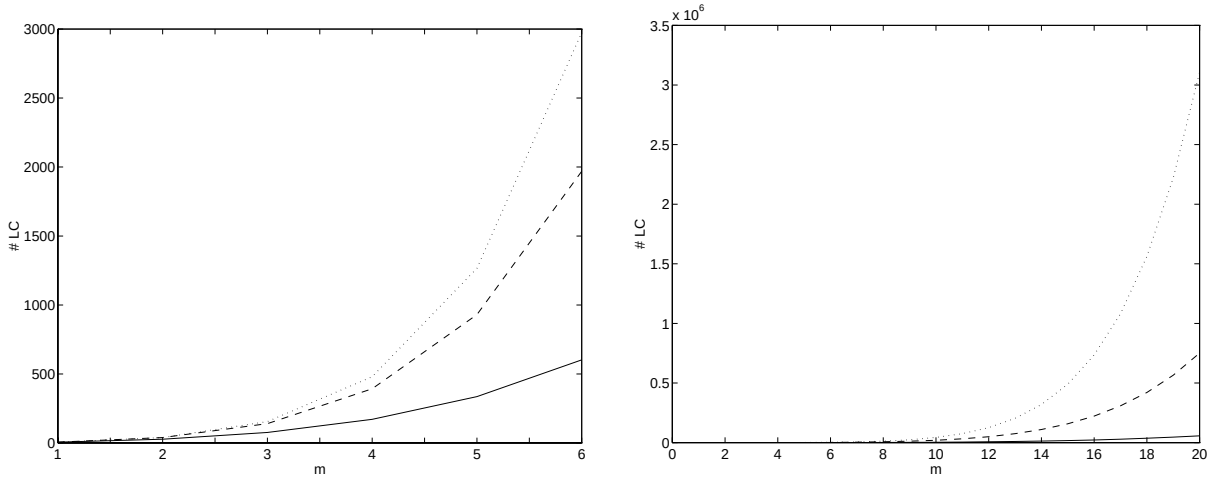

Figure 6: Comparison of Optimal (solid), Improved (dashed), and 1993 (dotted) Algorithms. In this figure, $\#G = 6$, and $K_Y = 2$.

- Looking at Equation 12, it would appear that the Optimal Algorithm is exponentially better than the 1993 Algorithm as we increase K_Y. Note, however, that as we increase K_Y above $\#G$, the runtime of the Optimal Algorithm is solely the cost of the basis conversion which is $\#G \cdot \binom{m+K_Y}{K_Y+1}$. But as K_Y goes to infinity, the runtime of the 1993 Algorithm also converges to $\#G \cdot \binom{m+K_Y}{K_Y+1}$. Thus, the two algorithms have the same behavior as K_Y goes to infinity with m and $\#G$ held fixed.

  This effect can be seen in Figure 8. When K_Y is less than $\#G$, the Optimal Algorithm has better performance than the 1993 Algorithm. But as K_Y increases beyond $\#G$, the two algorithms have the same runtime behavior. In the graph on the right of this figure, we see that as K_Y increases, the ratio of the runtimes tends to 1.

- Our final case is to consider the relative performance of the two algorithms as m increases when $K_Y > \#G$. Thus, we compare $\binom{m+\#G+K_Y}{m} - \binom{m+K_Y}{m}$ to $\#G \cdot \binom{m+K_Y}{K_Y+1}$:

$$\binom{m+\#G+K_Y}{m} - \binom{m+K_Y}{m} \quad vs \quad \#G \cdot \binom{m+K_Y}{K_Y+1}$$

$$\frac{\prod_{i=1}^{\#G}(m+K_Y+i)}{m\prod_{i=2}^{\#G}(K_Y+i)} - \frac{K_Y+1}{m} \quad vs \quad \#G$$

  In this case, we see that the Optimal Algorithm is asymptotically better than the 1993 Algorithm by a factor of

$$O(m^{\#G-1}).$$

  Graphs showing this behavior appear in Figure 9.

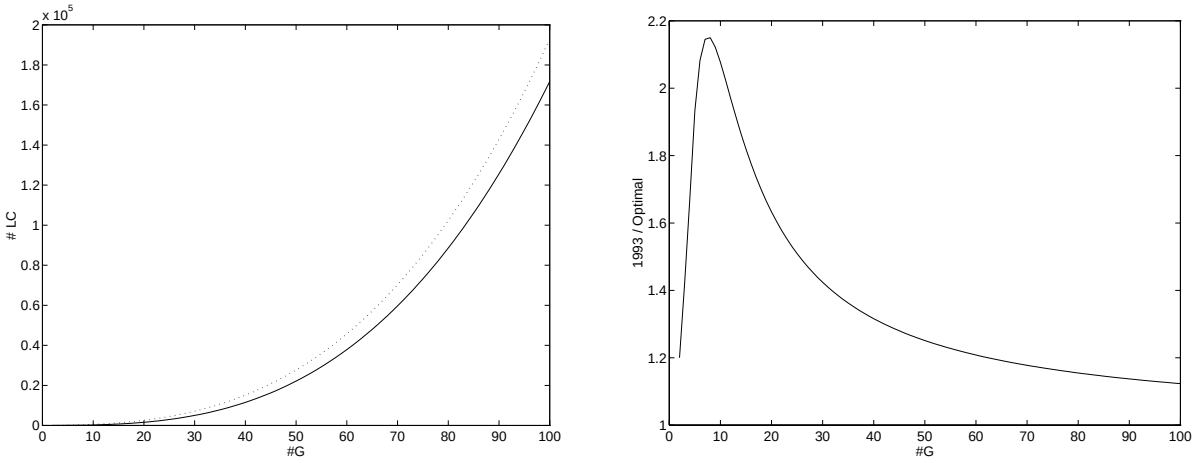

Figure 7: On the left, a comparison of Optimal (solid), and 1993 (dotted) Algorithms. In this figure, $K_Y = 3$ and $m = 3$. Note that the graph of the Improved Algorithm would fall between these two lines. On the right is a graph of the ratio of the runtime of the 1993 Algorithm to that of the Optimal Algorithm for this data.

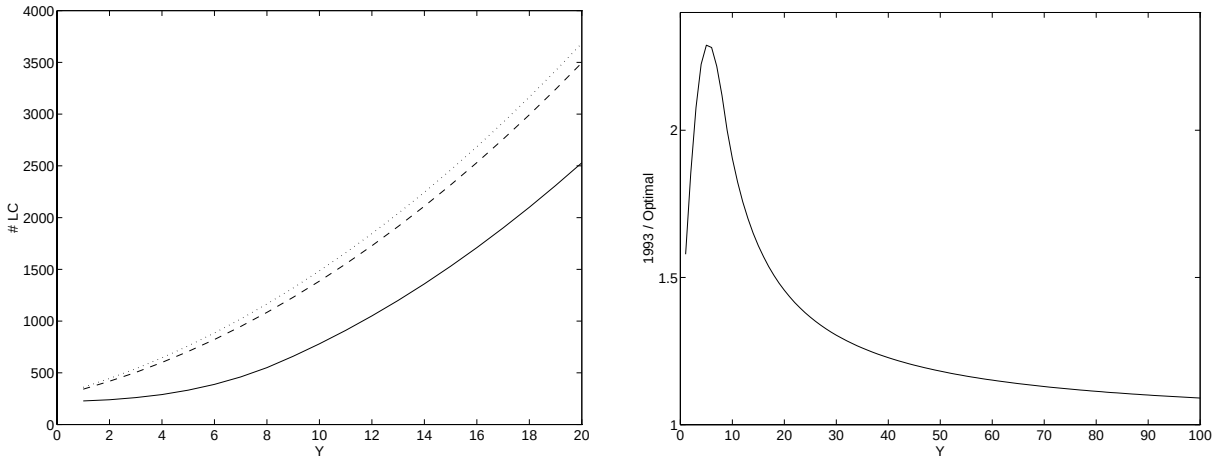

Figure 8: On the left, a comparison of Optimal (solid), Improved (dashed), and 1993 (dotted) Algorithms. In this figure, $\#G = 10$ and $m = 3$. On the right is a graph of the ratio of the runtime of the 1993 Algorithm to that of the Optimal Algorithm for this data over a larger range of K_Y.

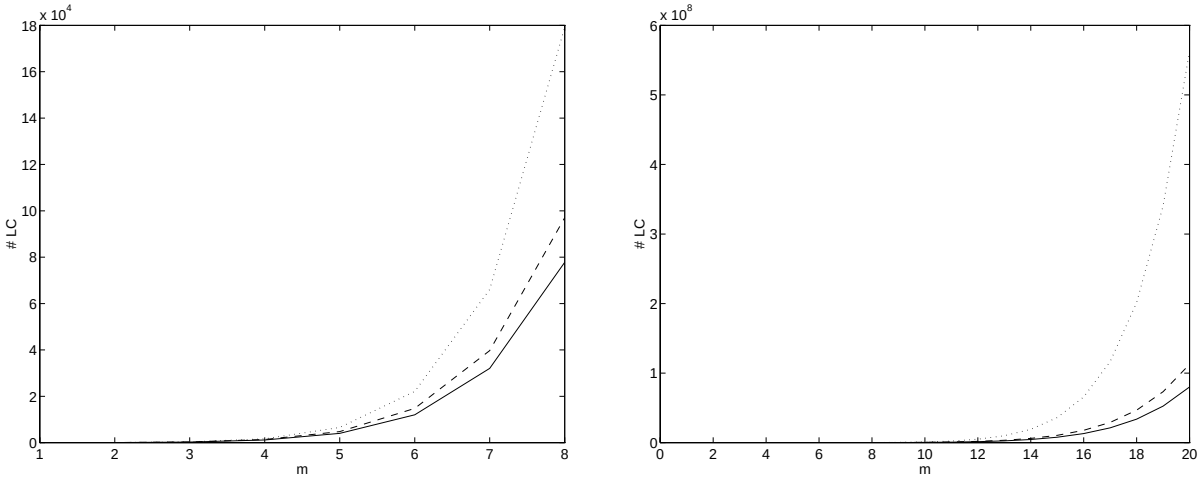

Figure 9: Comparison of Optimal (solid), Improved (dashed), and 1993 (dotted) Algorithms. In this figure, $\#G = 4$ and $K_Y = 9$.

6 Runtime Analysis

In the previous sections, we have analyzed the runtime of the various composition algorithms by counting the number of linear combinations made by the de Casteljau evaluations. However, this is only part of the total runtime cost of these algorithms. In this section we give a more complete analysis of the runtimes, and show how these costs can be reduced.

We begin by noting that each linear combination used by de Casteljau's algorithm to evaluate f requires $O(K_Y K_Z)$ operations, for a total operation count of

$$O\left(K_Y K_Z \binom{m+K_Y}{K_Y+1}\right).$$

However, you can also run de Casteljau's algorithm backwards to compute scalar weights for the control points. This variation of de Casteljau's algorithm takes $O(K_Y\binom{m+K_Y}{m-1})$ time to compute the weights, and $O(K_Z\binom{m+K_Y}{m})$ to combine the control points with these weights.

While we can use this technique to in the Naive and One Permutation Algorithms, note that it is computationally more expensive to use this method with the 1993 and the Improved Algorithms. Since we reuse the partially computed blossom values with these two algorithms, it is more efficient to run the forward de Casteljau algorithm as we will be reusing the evaluations that are expensive to compute (while with the backwards de Casteljau algorithm, we would be reusing the evaluations that are inexpensive to compute).

A second potential addition to our runtime cost is the conversion from a multi-index to a linear index. The conversion algorithm takes $O(K_Y)$ time, and we have to use it for $O(K_Y)$ domain points. Thus, for each linear combination in de Casteljau's algorithm, this would increase the runtime from

$O(K_Y K_Z)$ by an additive term of $O(K_Y^2)$ to $O(K_Y K_Z + K_Y^2)$ for a total of

$$O\left((K_Y K_Z + K_Y^2)\binom{m + K_Y}{K_Y + 1}\right)$$

However, Shoemake developed a version of de Casteljau's algorithm that computes the linear offsets incrementally, increasing the cost of each linear combination by an additive factor of only $O(K_Y)$ [7].

The more sophisticated composition algorithms also iterate through **G**'s control points, which requires iterating through **G**'s multi-indices, comparing these multi-indices, and converting these multi-indices into linear indexes into **G**'s control points. This would at most add an additive factor of $O(K_X)$ to runtime, which is easily dominated by the cost of a de Casteljau evaluation. Note however, that this extra cost can be reduced by precomputing an array of **G**'s multi-indices, each having an extra field that is the linear index corresponding to the multi-index.

Finally, there is an additive cost associated with the pre- and post-processing of **H**'s control points (Figure 11). However, this cost is proportional to the number of control points of **H** (times K_Z) and is dominated by other terms of the computation.

Thus, for the two naive algorithms, the full runtime cost is given by multiplying the number of linear combinations by a factor of $O(K_Y)$, while for the three sophisticated algorithms, we get an additional multiplicative cost of $O(K_Y K_Z)$.

7 Comparison of Algorithms

We have seen five algorithms for computing the composition of two polynomials. Three issues must be considered to decide which algorithm is appropriate for a particular application: Speed, numerical stability, and code complexity. We dismiss the Naive Algorithm and the One Permutation Algorithm since while they have the same numerical properties and roughly the same code complexity as the 1993 Algorithm, they are both significantly slower than the 1993 Algorithm. Therefore, you will always be better off implementing the 1993 Algorithm than either the Naive or One Permutation Algorithm.

This leaves us with three algorithms to compare: the 1993 Algorithm, the Improved Algorithm, and the Optimal Algorithm. We begin by considering code complexity. The 1993 Algorithm is far simpler than the Improved or Optimal Algorithms. While it is true that the Optimal Algorithm is essentially the 1993 Algorithm plus a change of basis and some code for `ExtractCPs`, a proper implementation of the change of basis code is difficult. The change of basis code given in Figure 16 is inadequate as the initial control points of **G** may not span the domain of **F**.

Difficulties arise when trying extend the change of basis code to select a "good" subsect of **G**'s control points to use for the basis. Further, if you select any subset other than the initial control points of **G**, then you have to keep track of which points you need to iterate over in the loop of `RecursiveCompose()`.

The code for the Improved Algorithm is clearly more complicated than the 1993 Algorithm. In addition to the complexities that are apparent in Figure 13, care must be taken when implementing `ngt()` and `nlt()` or you may lose the efficiency advantages over the 1993 Algorithm.

Moving to numerical stability, we begin by noting that the easist way to measure stability is by the maximum number of linear combinations separating the input data and any control point of $\mathbf{H}$. For the 1993 and the Improved Algorithms, it is easy to see that m linear combinations are used. Further, for many applications of composition, we would expect these all to be convex combinations (i.e., for the control points of $\mathbf{G}$ to lie inside the domain simplex of $\mathbf{F}$). This is the case, for example, for the problems of freeform deformation and triangular to tensor product basis conversion.

For the Optimal Algorithm, however, our longest path will be $m(K_Y + 1)$ affine combinations. The first mK_Y affine combinations are required for the basis conversion. While we would expect all of these to be convex combinations, the final m affine combinations will probably not be convex. Further, if the control points of $\mathbf{G}$ *nearly* lie in a subsimplex of $\mathbf{F}$'s domain simplex, we will have additional numerical difficulties. It should be noted, though, that not all of the paths from the input to control points of $\mathbf{H}$ will be of length $m(K_Y + 1)$. Some will be as short as m. Still, the numerical behavior of the Optimal Algorithm is potentially quite poor (although it should be acceptable in many situations).

Finally, we will consider efficiency. Although there is a proper ordering of the algorithms in terms of their runtime (i.e., the Optimal Algorithm is always the fastest, while the Improved Algorithm is faster than the 1993 Algorithm for $m > 2$ and equal otherwise), it is important to consider how much faster one algorithm is than another. While the Improved Algorithm is faster than the 1993 Algorithm, looking at Figures 6-9, the advantage is relatively small unless m is large. For example, with $m = 6$, the Improved Algorithm is only 25% to 33% faster than the 1993 Algorithm.

Likewise, the Optimal Algorithm's efficiency advantage is most apparent as m increases. While these advantages occur at lower values of m than they occur for the Improved Algorithm (giving, for example, a factor of 2 speed increase for $m = 3$ or 4), note that the numerical stability of the Optimal Algorithm decreases for larger values m. And as we increase $\#G$ or K_Y, the three algorithms asymptotically have the same runtime behaviour.

Thus, considering all three factors together (code complexity, efficiency, and numerical stability) you are probably best off implementing the 1993 Algorithm. There will be exceptions, of course. If speed is the primary concern, then the Optimal Algorithm is ideal. Likewise, if you plan to work with degrees of 3 or 4 and know that the initial control points of $\mathbf{G}$ span the domain of $\mathbf{F}$, then again the Optimal Algorithm is best. However, the only situation in which we can see the Improved Algorithm being the best choice is when stability is a concern and you expect m to be large.

8 Composition of a Blossom with a Set of Polynomials

In this section, we extend the notation of polynomial composition to the composition of a blossom with a set of polynomials. Mathematically, this introduces no problems – the proof of the theorem below is almost identical to the proof of the same theorem for the polynomial composition. Unfortunately, we will be unable to use most of the sophisticated algorithms developed in this paper. However, we will be able to use this generalization of composition to perform degree raising, and for this problem, we can use modified versions of the algorithms presented in this paper.

Given: Affine spaces $\mathcal{X}$ (of dimension K_X), $\mathcal{Y}$ (of dimension K_Y), and $\mathcal{Z}$ (of dimension K_Z), control points $\{\mathbf{G}^j_{\vec{i}}\}_{\vec{i}\in\mathbb{I}^{\ell_j}_{K_X}}$, $1 \le j \le m$ defining m Bézier simplices $\mathbf{G}^j : \mathcal{X} \to \mathcal{Y}$ of degree ℓ_j relative to a domain simplex $\Delta_{\mathcal{X}} \subset \mathcal{X}$, and control points $\{\mathbf{F}_{\vec{p}}\}_{\vec{p}\in\mathbb{I}^m_{K_Y}}$ defining an m-affine Bézier simplex $f : \mathcal{Y}^m \to \mathcal{Z}$ relative to a domain simplex $\Delta_{\mathcal{Y}} \subset \mathcal{Y}$.

Find: The control points $\{\mathbf{H}_{\vec{j}}\}_{\vec{j}\in\mathbb{I}^L_{K_X}}$ of the degree $L = \sum_{i=1}^m \ell_i$ Bézier simplex $\mathbf{H} = f \circ \mathbf{G} = f(\mathbf{G}^1, \ldots, \mathbf{G}^m)$ relative to $\Delta_{\mathcal{X}}$.

Solution: If f denotes the blossom of $\mathbf{F}$, then

$$\mathbf{H}_{\vec{j}} = \sum_{\substack{I\in\mathbb{I}^{\vec{\ell},m}_{K_X} \\ |I|=\vec{j}}} \mathcal{C}(I) f(\mathbf{G}_I), \qquad \vec{j} \in \mathbb{I}^L_{K_X} \tag{13}$$

where $\mathbf{G}_I$ with $I = (\vec{i}_1, ..., \vec{i}_m)$ is an abbreviation for $(\mathbf{G}^1_{\vec{i}_1}, \ldots, \mathbf{G}^m_{\vec{i}_m})$.

Proof: The proof is essentially the one appearing in DeRose et al [2]:

$$\begin{aligned}
\mathbf{H}(t) &= f(\mathbf{G}^1(t), \ldots, \mathbf{G}^n(t)) \\
&= f\left(\sum_{\vec{i}_1\in\mathbb{I}^{\ell_1}_{K_X}} \mathbf{G}^1_{\vec{i}_1} B^{\ell_1}_{\vec{i}_1}(t), \ldots, \sum_{\vec{i}_m\in\mathbb{I}^{\ell_m}_{K_X}} \mathbf{G}^m_{\vec{i}_m} B^{\ell_m}_{\vec{i}_m}(t) \right) \\
&= \sum_{I\in\mathbb{I}^{\vec{\ell},m}_{K_X}} f\left(\mathbf{G}^1_{\vec{i}_1}, \mathbf{G}^2_{\vec{i}_2}, \ldots, \mathbf{G}^m_{\vec{i}_m}\right) \prod_{j=1}^m B^{\ell_j}_{\vec{i}_j}(t) \\
&= \sum_{I\in\mathbb{I}^{\vec{\ell},m}_{K_X}} f(\mathbf{G}_I) B^L_I(t) \\
&= \sum_{I\in\mathbb{I}^{\vec{\ell},m}_{K_X}} f(\mathbf{G}_I) \mathcal{C}(I) B^L_{|I|}(t).
\end{aligned} \tag{14}$$

We can find the control points of $\mathbf{H}$ by grouping together terms in Equation 14 such that $|I| = \vec{j}$, yielding

$$\mathbf{H}_{\vec{j}} = \sum_{\substack{I\in\mathbb{I}^{\vec{\ell},m}_{K_X} \\ |I|=\vec{j}}} f(\mathbf{G}_I) \mathcal{C}(I). \tag{15}$$

Unfortunately, while the Naive Algorithm discussed in Section 3 can be used to compute the control points for this generalization of polynomial composition, in general, we can not use any of the speedups discusses in this paper since each $\mathbf{G}^i$ might have different control points. We can, however, run de Castlejau's algorithm backwards, and reduce the runtime cost by a factor of K_Z.

8.1 Special Cases

There are two special cases of particular interest. The first is when the functions $\mathbf{G}^i$ are all equal: $\mathbf{G}^1(t) = \ldots = \mathbf{G}^m(t) = \mathbf{G}(t)$. This gives us

$$\begin{aligned} f(\mathbf{G}^1(t), \ldots, \mathbf{G}^m(t)) &= f(\mathbf{G}(t), \ldots, \mathbf{G}(t)) \\ &= \mathbf{F}(\mathbf{G}(t)), \end{aligned}$$

where $F(t) = f(t, \ldots, t)$ is the degree m polynomial agreeing with f on f's diagonal. Thus, we have the composition of polynomial functions.

The second special case is when each $\mathbf{G}^i$ is the identity function. Let $I^n(t)$ be the nth degree representation of the identity function. Now consider

$$\mathbf{H}(t) = f(I^n(t), I^1(t), I^1(t), \ldots, I^1(t)),$$

where f is an m-affine function. The result $\mathbf{H}$ is a degree $m+n-1$ representation of $\mathbf{F}$. Thus, we have degree raising. Since most of f's arguments are linear representations of the identity function, we can simplify our equations somewhat, and in particular, we can simplify the combinatorial constant:

$$\begin{aligned} \mathbf{H}(t) &= f(I^n(t), I^1(t), I^1(t), \ldots, I^1(t)) \\ &= \sum_{I \in \mathbb{I}^{\vec{\ell},m}_{K_X}} f(I_I)\mathcal{C}(I)B^{m+1}_{|I|}(t) \\ &= \sum_{I \in \mathbb{I}^{\vec{\ell},m}_{K_X}} f(I_I)\frac{\binom{|\vec{i}_1|}{\vec{i}_1}}{\binom{m+1}{|I|}}B^{m+1}_{|I|}(t) \end{aligned}$$

Solving for $\mathbf{H}_{\vec{\jmath}}$,

$$\begin{aligned} \mathbf{H}_{\vec{\jmath}} &= \sum_{\substack{I \in \mathbb{I}^{(n,1,1,\ldots,1),m}_{K_X} \\ |I| = \vec{\jmath}}} f(I_I)\frac{\binom{|\vec{i}_1|}{\vec{i}_1}}{\binom{m+1}{\vec{\jmath}}} \\ &= \sum_{\substack{I \in \mathbb{I}^{(n,1,1,\ldots,1),m}_{K_X} \\ |I| = \vec{\jmath}}} f(I^n_{\vec{i}_1}, I^1_{\vec{i}_2}, I^1_{\vec{i}_3}, \ldots, I^1_{\vec{i}_m})\frac{\binom{|\vec{i}_1|}{\vec{i}_1}}{\binom{m+1}{\vec{\jmath}}} \end{aligned} \tag{16}$$

We can either use this formula directly to compute control points when raising the degree by an arbitrary amount n and k, or we can use it to derive more efficient formulae for any special cases of interest. As an example of this latter approach, in Appendix A the above formula is used to rederive the standard formula for degree raising a curve.

Note that we can use a modified form of any of the faster algorithms to compute the degree raised polynomial. Essentially, we evaluate all the control points from the degree one identity functions first (i.e., the I^1s), and then evaluate at the I^ns.

References

[1] C. de Boor. B-form basics. In G. Farin, editor, *Geometric Modeling: Algorithms and New Trends*, pages 131–148. SIAM, 1987.

[2] Tony DeRose, Ronald Goldman, Hans Hagen, and Stephen Mann. Composition again. *ACM Transactions on Graphics*, 12(2):113–135, 1993.

[3] Tony D. DeRose. Composing Bézier simplexes. *ACM Transactions on Graphics*, 7(3):198–221, July 1988.

[4] Tony D. DeRose. A coordinate-free approach to geometric programming. In *Math for Siggraph*. Siggraph Course Notes #23, 1989. Also available as Technical Report No. 89-09-16, Department of Computer Science and Engineering, University of Washington, Seattle, WA (September, 1989).

[5] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, second edition, 1990.

[6] Lyle Ramshaw. Blossoming: A connect-the-dots approach to splines. Technical Report 19, Digital Systems Research Center, Palo Alto, Ca, 1987.

[7] Ken Shoemake. Efficient de Casteljau indexing. In preparation, 1995.

A Derivation of curve degree raising via composition

We could perform degree raising by evaluating Equation 16 for the controls points of the degree raised curve. However, if we only want to degree raise for specific cases (e.g., curves by degree 1, or surfaces by degree 2) then it is more efficient to solve the equation analytically. In this appendix, we rederive the equations for degree raising curves by 1 degree.

We begin from working from Equation 16, with the additional knowledge that $n = 2$, $K_X = 1$, $\vec{\ell} = (2, 1, 1, \ldots, 1)$, $\vec{\imath}_1 \in \{(2,0), (1,1), (0,2)\}$, and for $1 < j \leq m$, $\vec{\imath}_j \in \{(1,0), (0,1)\}$:

$$
\begin{aligned}
\mathbf{H}_{(j,m+1-j)} &= \sum_{\substack{I \in \mathbb{I}_1^{(2,1,1,\ldots,1),m} \\ |I| = (j,m+1-j)}} f(\mathbf{G}_I) \frac{\binom{2}{\vec{\imath}_1}}{\binom{m+1}{(j,m+1-j)}} \\
&= \binom{m-1}{j-2} f((2,0), (0,1)^{<m+1-j>}, (1,0)^{<j-2>}) \frac{\binom{2}{(2,0)}}{\binom{m+1}{(j,m+1-j)}} + \\
&\quad \binom{m-1}{j-1} f((1,1), (0,1)^{<m+j>}, (1,0)^{<j-1>}) \frac{\binom{2}{(1,1)}}{\binom{m+1}{(j,m+1-j)}} + \\
&\quad \binom{m-1}{j} f((0,2), (0,1)^{<m-1-j>}, (1,0)^{<j>}) \frac{\binom{2}{(0,2)}}{\binom{m+1}{(j,m+1-j)}}
\end{aligned}
$$

$$
\begin{aligned}
&= \binom{m-1}{j-2} f((2,0),(0,1)^{<m+1-j>},(1,0)^{<j-2>}) \frac{1}{\binom{m+1}{(j,m+1-j)}} + \\
&\quad \binom{m-1}{j-1} f((1,1),(0,1)^{<m+j>},(1,0)^{<j-1>}) \frac{2}{\binom{m+1}{(j,m+1-j)}} + \\
&\quad \binom{m-1}{j} f((0,2),(0,1)^{<m-1-j>},(1,0)^{<j>}) \frac{1}{\binom{m+1}{(j,m+1-j)}} \\
&= \frac{j(j-1)}{m(m+1)} f((2,0),(0,1)^{<m+1-j>},(1,0)^{<j-2>}) + \\
&\quad \frac{2j(m+1-j)}{m(m+1)} f((1,1),(0,1)^{<m+j>},(1,0)^{<j-1>}) + \\
&\quad \frac{(m-j)(m-j+1)}{m(m+1)} f((0,2),(0,1)^{<m-1-j>},(1,0)^{<j>}) \\
&= \frac{j(j-1)}{m(m+1)} f((2,0),(0,1)^{<m+1-j>},(1,0)^{<j-2>}) + \\
&\quad \frac{2j(m+1-j)}{m(m+1)} \Big(f((2,0),(0,1)^{<m+1-j>},(1,0)^{<j-2>}) + \\
&\qquad\qquad f((0,2),(0,1)^{<m-1-j>},(1,0)^{<j>}) \Big)/2 + \\
&\quad \frac{(m-j)(m-j+1)}{m(m+1)} f((0,2),(0,1)^{<m-1-j>},(1,0)^{<j>}) \\
&= \frac{j}{m+1} f((2,0),(0,1)^{<m+1-j>},(1,0)^{<j-2>}) + \\
&\quad \frac{m-j+1}{m+1} f((0,2),(0,1)^{<m-1-j>},(1,0)^{<j>}) \\
&= \frac{j}{m+1} f((0,1)^{<m+1-j>},(1,0)^{<j-1>}) + \frac{m-j+1}{m+1} f((0,1)^{<m-j>},(1,0)^{<j>}),
\end{aligned}
$$

which is the standard degree raising formula for curves.

B Code

Pseudo-code for the algorithms presented in the paper appears on the following pages. We have not given code for the Naive or One Permutation Algorithms. Code for converting from multi-indices to linear indices appears in [2].

EvalBlossom(V, $\mathbf{u}_1, \ldots, \mathbf{u}_d$)
{ V is the control net for a Bézier simplex characterizing a blossom q
  returned is the point $q(\mathbf{u}_1, ..., \mathbf{u}_d)$}
begin
    $\bar{V} \leftarrow$ Prepare(V)
    for $\ell = 1$ to d
        EvalBlossomArgument($\bar{V}$, ℓ, $\mathbf{u}_\ell$)
    endfor
    return $\bar{V}.\mathbf{cp}_{\vec{0}}^{[d]}$
end

EvalBlossomArgument($\bar{V}$, ℓ, $\mathbf{u}$)
begin
    $d \leftarrow \bar{V}$.degree
    $(b_0, \ldots, b_k) \leftarrow$ Barycentric coordinates of $\mathbf{u}$ relative to $\bar{V}$.domain
    for all $\vec{\imath} \in \mathbb{I}_k^{d-\ell}$
        $\bar{V}.\mathbf{cp}_{\vec{\imath}}^{[\ell]} \leftarrow b_0 \bar{V}.\mathbf{cp}_{\vec{\imath}+\vec{e}_0}^{[\ell-1]} + \cdots + b_k \bar{V}.\mathbf{cp}_{\vec{\imath}+\vec{e}_k}^{[\ell-1]}$
    endfor
end

Prepare(V)
{ Initialize and return a structure into which the partial results of the
  blossom evaluation algorithm can be stored.}
begin
    $d \leftarrow \bar{V}$.degree $\leftarrow V$.degree
    for all $\vec{\imath} \in \mathbb{I}_k^d$
        $\bar{V}.\mathbf{cp}_{\vec{\imath}}^{[0]} \leftarrow V.\mathbf{cp}_{\vec{\imath}}$
    endfor
    return $\bar{V}$
end

Figure 10: 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 $V.\mathbf{cp}$ are the control points of Q relative to V.domain.

InitializeH($\mathbf{F}$, $\mathbf{G}$, $\mathbf{H}$)
begin
    $k \leftarrow$ Dimension($\mathbf{G}$.domain), $\ell \leftarrow \mathbf{G}$.degree, $m \leftarrow \mathbf{F}$.degree
    $\mathbf{H}$.degree $= \ell * m$
    $\mathbf{H}$.domain $= \mathbf{G}$.domain
    for all $\vec{j} \in \mathbb{I}_k^{\ell m}$
        $H.\mathbf{cp}_{\vec{j}} \leftarrow \mathbf{0}$
    endfor
end

PostProcessH($\mathbf{H}$)
begin
    for all $\vec{j} \in \mathbb{I}_k^{\ell m}$
        $H.\mathbf{cp}_{\vec{j}} \leftarrow H.\mathbf{cp}_{\vec{j}} / \binom{\ell m}{\vec{j}}$
    endfor
end

Figure 11: Preparation and postprocessing of $\mathbf{H}$.

Compose($\mathbf{F}$, $\mathbf{G}$, $\mathbf{H}$)
begin
    InitializeH($\mathbf{F}$, $\mathbf{G}$, $\mathbf{H}$) { See Figure 11}
    $\bar{F} \leftarrow$ Prepare($\mathbf{F}$) { See Figure 10 }
    $\vec{\jmath}_{\min} \leftarrow (0, ..., 0, G.\text{degree})$
    RecursiveCompose($\bar{F}$, $\mathbf{G}$, $\mathbf{H}$, 0, $\vec{\jmath}_{\min}$, $\vec{0}$, $\mathbf{F}$.degree!, 0)
    PostProcessH($\mathbf{H}$) { See Figure 11}
end

RecursiveCompose($\bar{F}$, $\mathbf{G}$, $\mathbf{H}$, n, $\vec{m}$, $\vec{s}$, c, μ)
{ n : recursion control variable }
{ $\vec{m}$: minimum multi-index value allowed for $\vec{\imath}_{n+1}$ }
{ $\vec{s}$: sum of multi-indices computed in I thus far }
{ c : scalar to use as weight at bottom of recursion }
{ μ : multiplicity of $\vec{m}$ in I thus far }
begin
    if $n = \bar{F}$.degree then
        $H.\mathbf{cp}_{\vec{s}} \leftarrow H.\mathbf{cp}_{\vec{s}} + c * \bar{F}.\mathbf{cp}_{\vec{0}}^{[n]}$
        return;
    endif
    for all $\vec{\imath}_{n+1} \in \mathbb{I}_k^\ell$ with $\vec{\imath}_{n+1} \geq \vec{m}$ in increasing order
        EvalBlossomArgument($\bar{F}$, $n+1$, $G.\mathbf{cp}_{\vec{\imath}_{n+1}}$) { compute $\bar{F}^{[n+1]}$ from $\bar{F}^{[n]}$}
        if $\vec{\imath}_{n+1} = \vec{m}$ then $\mu' \leftarrow \mu + 1$ else $\mu' \leftarrow 1$
        RecursiveCompose($\bar{F}$, $\mathbf{G}$, $\mathbf{H}$, $n+1$, $\vec{\imath}_{n+1}$, $\vec{s} + \vec{\imath}_{n+1}$, $c * \binom{|\vec{\imath}_{n+1}|}{\vec{\imath}_{n+1}} / \mu'$, μ')
    endfor
end

Figure 12: The Bézier composition algorithm.

```
Compose(F, G, H)
begin
    InitializeH(F, G, H);                                   { See Figure 11 }
    F̄ ← Prepare(F);                                         { See Figure 10 }
    Rec(F.Deg, F, G.cp, H, F.deg, G.first, 0⃗);
    PostProcessH(H);                                        { See Figure 11 }
end

Rec(n, F, G, H, M_max, prev, sum, c)
begin
    if (n = 0) then H.cp_sum += c * F_0^[n];
    M_min = ceil(|G|/n);                          { compute the number of times to eval}
    if (M_min > M_max) then return;               { May occur when e=1 at previous level}
    for p=G.first to G.last do                      { iterate through G's remaining CPs}
        if ( Marked(p) ) next;
        if ( p < prev ) then e = 1; else e = 0;     { If out of order, *must* evaluate 1 less time}
        { We may not be able to complete the evaluation for some points }
        if ( n − ngt(p,G) * (M_max − e) + nlt(p,G) * (M_max − e − 1) > M_max − e ) then next;
        { And some points may require a new minimum }
        L_min = ceil(M_min, (n + nlt(p,G))/(ngt(p,G) + 1 + nlt(p,G)));
        if ( L_min > min(M_max − e, n) ) then next;                    { can this happen? }
        F' =Eval(F, p, L_min − 1);
        s⃗ = (L_min − 1) * p.mi;
        c̄ = c * (|s⃗| choose s⃗)^(L_min − 1) / (L_min − 1)!;
        Mark(p)
        for i = L_min to M_max − e do
            F'=Eval(F', p);
            c̄ = c̄ * (|s⃗| choose s⃗)/i;
            Rec(n − i, F', G − p, H, min(i, n − i), p, sum + s⃗, c̄);
        end
        Unmark(p)
    end
end
```

Figure 13: Improved Algorithm

Compose($\mathbf{F}$, $\mathbf{G}$, $\mathbf{H}$)
begin
    InitializeH($\mathbf{F}$, $\mathbf{G}$, $\mathbf{H}$); { See Figure 11 }
    $\vec{\jmath}_{\min} \leftarrow$ ConvertBasis($\mathbf{F}$, $\mathbf{G}$); { Convert $\mathbf{F}$ to be w.r.t $\mathbf{G}$'s first control points; Figure 16 }
    $\bar{F} \leftarrow$ Prepare($\mathbf{F}$); { See Figure 10 }
    RecursiveCompose($\bar{F}$, $\mathbf{G}$, $\mathbf{H}$, 0, $\vec{\jmath}_{\min}$, $\vec{0}$, $\mathbf{F}$.degree!, 0);
    PostProcessH($\mathbf{H}$); { See Figure 11 }
end

RecursiveCompose($\bar{F}$, $\mathbf{G}$, $\mathbf{H}$, n, $\vec{m}$, $\vec{s}$, c, μ)
{ n : recursion control variable }
{ $\vec{m}$: minimum multi-index value allowed for $\vec{\imath}_{n+1}$ }
{ $\vec{s}$: sum of multi-indices computed in I thus far }
{ c : scalar to use as weight at bottom of recursion }
{ μ : multiplicity of $\vec{m}$ in I thus far }
begin
    ExtractCPs($\bar{F}$, H, n, $\vec{s}$, c)
    if $n = \bar{F}$.degree then return;
    for all $\vec{\imath}_{n+1} \in \mathbb{I}_k^{\ell}$ with $\vec{\imath}_{n+1} \geq \vec{m}$ in increasing order
        EvalBlossomArgument($\bar{F}$, $n+1$, $G.\mathbf{cp}_{\vec{\imath}_{n+1}}$) { compute $\bar{F}^{[n+1]}$ from $\bar{F}^{[n]}$ }
        if $\vec{\imath}_{n+1} = \vec{m}$ then $\mu' \leftarrow \mu + 1$ else $\mu' \leftarrow 1$
        RecursiveCompose($\bar{F}$, $\mathbf{G}$, $\mathbf{H}$, $n+1$, $\vec{\imath}_{n+1}$, $\vec{s} + \vec{\imath}_{n+1}$, $c * \binom{|\vec{\imath}_{n+1}|}{\vec{\imath}_{n+1}} / \mu'$, μ')
    endfor
end

Figure 14: The optimal Bézier composition algorithm.

ExtractCPs($\bar{F}$, H, n, $\vec{s}$, c)
begin
    for $\vec{\imath} \in \mathbb{I}_{K_Y}^{n}$ do
        for $j = 0$ to K_Y do
            $\vec{s} \leftarrow \vec{s} + i_j * F.\vec{simp}_j$;
            $c \leftarrow c * \binom{|F.\vec{simp}_j|}{F.\vec{simp}_j}^{i_j} / i_j!$;
        end
        $H.cp_{\vec{s}} \leftarrow H.cp_{\vec{s}} + c * \bar{F}_{\vec{\imath}}^{[n]}$
    end
end

Figure 15: Code to extract evaluations of f from the simplicial array.

```
ConvertBasis(F, G)
begin
    i⃗ ← (0, ..., 0, G.degree)
    for i = 0 to F.domain.dimension
        F.cp ← Subdivide(F, G.cp_i⃗, i)
        i⃗ ← i⃗.next
    endfor
    return i⃗
end
```

Figure 16: Basis conversion