Programming Support for Blossoming

Wayne Liu
wbliu@vlsi.uwaterloo.ca
Electrical and Computer Engineering

Stephen Mann
smann@cgl.uwaterloo.ca
Computer Science Department

University of Waterloo
Waterloo, Ontario, N2L 3G1 CANADA

Abstract

A C++ library has been created to facilitate prototyping of curve and surface modeling techniques. The library provides blossoming datatypes to support creation of modeling techniques based on blossoming analysis. The datatypes have efficient operations that are generalizations of important CAGD algorithms and can be used to implement many algorithms. Most importantly, the library is able to interoperate with user-supplied datatypes or routines to create complex modeling techniques.

Résumé

Une librairie de fonctions C++ a été créée pour faciliter le prototypage de techniques de modélisation de courbes et de surfaces. La librairie donne accès à des structures de données de type "formes polaires" qui aident à la création de techniques de modélisation basées sur une analyse "blossoming" (de forme polaire). Les opérations, efficaces, sur les structures de données sont des généralisations d'algorithmes importants en Conception Géométrique Assistée par Ordinateur (CAGD); plusieurs types d'algorithmes peuvent être basés sur ces opérations. L'aspect le plus important de cette librairie est la possibilité de fonctionner avec des structures de données ou des routines fournies par l'utilisateur, pour créer des techniques de modélisation complexes.

Keywords: Blossoms, Curves and Surfaces, Graphics data structures and data types, Software

1 Introduction

Computer Aided Geometric Design (CAGD) is concerned with modeling curves and surfaces on computers. Research focuses on finding various techniques of representing curves and surfaces in computer-compatible form, and algorithms for manipulating these representations. This research has applications in CAD/CAM. For a general introduction to CAGD, see [6].

The most successful techniques represent curves and surfaces with piecewise polynomial functions, such as Bézier patches and NURBS. Many properties of such techniques are most easily studied using blossoming analysis. Blossoming analysis was introduced into CAGD in the mid-1980's [12, 3]. Since that time, it has proven to be a simple and powerful mathematical tool. It does not require advanced mathematical concepts, yet it reveals the properties of important modeling techniques. Researchers continue to apply blossoming analysis to find new modeling.

In addition to analyzing new ideas mathematically, researchers must also implement prototypes, computer programs that test the practicality of these ideas. Thus, programming is an essential step in CAGD research.

The task of programming involves translating from the mathematical analysis into computer code. This translation is often difficult. The problem lies in translating mathematical concepts, such as piecewise polynomials, into computer language concepts, such as floating-point arithmetic. It is unclear how to translate from one to the other, and in general, they bear no resemblance to each other. Ideally, the programmer should be able to manipulate the same concepts in the code as in the analysis. Then, the translation process would be straight-forward, and the programming would be simple. The solution is to create datatypes for blossoming.

A datatype is simply an abstract set of objects with operations that can be performed on these objects. In this case, the objects correspond to concepts used in the blossoming analysis, such as blossoms, blossom arguments, or spaces. The operations

perform meaningful actions on the objects in terms of blossoming analysis. Thus, the programmer can use the operations to manipulate the mathematical concepts in the code.

In this article, we discuss the Blossom Classes, a C++ library we developed to support programming with blossoming datatypes. The library is designed to be useful for many applications. The library provides a set of general datatypes that can be used to code many modeling techniques. In creating operations for the datatypes, we discovered generalizations of important CAGD algorithms. The resulting operations are efficient building blocks for many algorithms.

In this paper, we only give an introduction to the Blossom Classes. For a more detailed description, the interested reader is referred to [8].

1.1 Previous Work

DeRose and Goldman [5] first proposed the approach of using blossoming datatypes. Their proposed system extends a coordinate-free geometry programming package [4] with a blossom datatype. Their datatype supports the creation and manipulation of Bézier curves and patches.

The major work implemented using the blossoming datatype approach was by Dahl. He provided a blossom datatype in *Weyl* [2], a language for CAGD research. The Weyl language provides datatypes that closely mimic the corresponding mathematical concepts. Weyl also has an interactive, graphical environment. The goal of Weyl was to provide an environment where researchers can manipulate mathematical objects in familiar ways and receive graphical feedback.

However, Weyl is unsuitable for creating complex modeling techniques. The biggest drawback is that the Weyl language environment is "closed": it cannot be used as part of another system, nor can other tools be easily integrated with it. This is a serious problem as the Weyl environment does not provide all the facilities that researchers need. For example, if a surface fitting scheme requires the use of singular value decomposition, that function would have to be created in the Weyl language.

Finally, although object oriented programming has been used for other purposes in CAGD [1, 14], it has not been used to develop a blossoming package (although Bartels mentions the idea [1]).

1.2 Overview

In Section 2, we review the technique of blossoming analysis. In Section 3, we describe the design of the Blossom Classes. In Section 4, we evaluate the usefulness of the library by using it to implement different techniques and algorithms.

2 Background on Blossoming

Blossoming analysis is based on the *blossom*, which is defined as a symmetric and multi-affine (affine in each argument) map of n arguments. The following theorem [12] states that polynomials and blossoms are essentially the same.

Theorem 2.1 (The Blossoming Principle) There is a one-to-one correspondence between the degree n polynomials, $F : X \to Y$, and the n–affine blossoms, $f : X^n \to Y$, such that

$$F(u) = f(\underbrace{u, \ldots, u}_{n}),$$

where X and Y are spaces of arbitrary dimension.

If f is a blossom of three arguments, then the symmetric property implies $f(x, y, z) = f(y, x, z) = f(\text{any permutation of } x, y, z)$. The multi-affine property implies

$$f(\alpha w + (1-\alpha)x, y, z) = \alpha f(w, y, z) + (1-\alpha) f(x, y, z).$$

If we know f at certain sets of arguments, then the symmetric and multi-affine properties allow us to compute f at any set of arguments. For example, as illustrated in Figure 1, given the values of $f(0, 1, 2)$, $f(1, 2, 3)$, $f(2, 3, 4)$, and $f(3, 4, 6)$, it is possible to compute $f(2.7, 1.8, 3.4)$. First, by the symmetric property,

$$f(0, 1, 2) = f(1, 2, 0).$$

Then, by the multi-affine property,

$$\begin{aligned} f(1, 2, \underline{2.7}) &= f(1, 2, \underline{.1 \times 0 + .9 \times 3}) \\ &= \underline{.1} f(1, 2, \underline{0}) + \underline{.9} f(1, 2, \underline{3}). \end{aligned}$$

Thus, from the first pair of values of f, the new value $f(1, 2, \underline{2.7})$ can be computed. In a similar manner, using the next pair and the last pair, $f(2, 3, \underline{2.7})$ and $f(3, 4, \underline{2.7})$ can be computed. These three new values combine to give $f(2, \underline{2.7}, \underline{1.8})$ and $f(3, \underline{2.7}, \underline{1.8})$, which in turn combine to give $f(\underline{2.7}, \underline{1.8}, \underline{3.4})$. This example evaluated the blossom of a curve. Since the concept of a blossom is independent of dimension, the same approach can evaluate surfaces.

The blossom can also be used to evaluate the derivatives of a polynomial. If we distinguish points

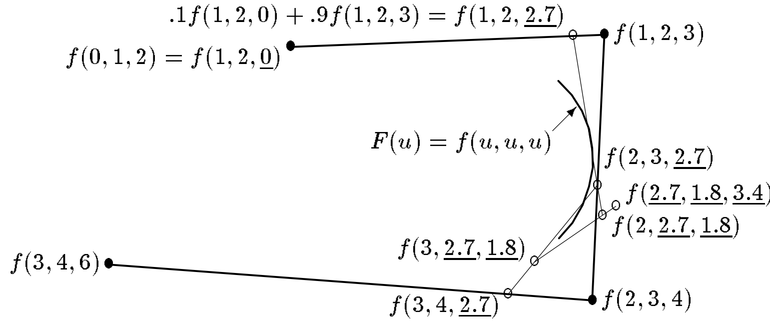

Figure 1: Evaluating a blossom

from vectors in an affine geometry sense, then the evaluation of a blossom at an argument set consisting only of points yields a point. However, if one or more of the arguments are vectors, then the evaluation is proportional to a derivative of the polynomial. Ramshaw has shown [12] that the directional derivative of a polynomial F in direction $\vec{v}$ is given by

$$\frac{\partial}{\partial \vec{v}} F(x) = n f(x^{n-1}\vec{v}). \tag{1}$$

If more than one argument is a vector, then we obtain higher order derivatives. Note that in this equation, we have used the tensor form of the blossom (i.e., there are no commas separating the blossom arguments). For an introduction to tensors, see for example [12, 8]. We will continue to use tensor notation for the remainder of this paper.

2.1 Blossom Knots

The arguments of the known values of the blossom must follow a pattern to allow the computation of new blossom values. The arguments are generated by the *knot net* for the blossom.

A one-dimensional knot net for a degree n blossom is the set of points in the domain

$$\{a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{n-1}\}$$

The blossom values generated from this knot net are

$$\begin{aligned} P_0 &= f(a_{n-1}\cdots a_0), \\ P_1 &= f(a_{n-2}\cdots a_0 b_0), \\ &\cdots \\ P_k &= f(a_{n-1-k}\cdots a_0 b_0 \cdots b_k), \\ &\cdots \\ P_n &= f(b_0 \cdots b_{n-1}). \end{aligned}$$

The values of the blossom, P_k, are called the *coefficients* of the blossom. Thus, the knot net for the previous one dimensional example is $\{0, 1, 2, 3, 4, 6\}$; for cubic Bézier curves over the interval $[0, 1]$ the knot net is $\{0, 0, 0, 1, 1, 1\}$.

Note that modeling applications are only interested in using a piece of the polynomial. For 1-D blossoms, that piece is the segment of the polynomial over $[a_0, b_0)$. Thus, Figure 1 shows the segment of the curve over the interval $[3, 4)$. This segment is related to a segment of a longer B-spline.

A two-dimensional knot net for a degree n blossom is the set of points in the domain

$$\{a_0, a_1, \ldots, a_{n-1}, b_0, b_1, \ldots, b_{n-1}, c_0, c_1, \ldots, c_{n-1}\}$$

The blossom values generated from this knot net are

$$\begin{aligned} P_{0,0} &= f(a_{n-1}\cdots a_0), \\ P_{1,0} &= f(a_{n-2}\cdots a_0 c_0), \\ P_{0,1} &= f(a_{n-2}\cdots a_0 b_0), \\ &\cdots \\ P_{i,j} &= f(a_0\cdots a_{n-1-i-j} b_0 \cdots b_i c_0 \cdots c_j), \\ &\cdots \\ P_{0,n} &= f(c_0 \cdots c_{n-1}). \end{aligned}$$

For cubic Bézier patches over the domain triangle $\{a, b, c\}$, the knot net is $\{a, a, a, b, b, b, c, c, c\}$. Modeling applications are interested in the piece of the polynomial over the domain triangle $\{a_0, b_0, c_0\}$.

In higher dimensions, the knot net consists of points $\{t_{k,l} : k = 0..d, l = 0..n-1\}$. For blossoms of arbitrary dimensions, it is more convenient to index the coefficients using *multi-indices*, which are tuples of integers. For a blossom of dimension d and degree n, its coefficients are $P_{\vec{\imath}}$ where $\vec{\imath}$ is a multi-index of $d{+}1$ integers, $(i_0, \ldots, i_d)$, and the sum of its elements is $i_0 + \cdots + i_d = n$.

Specifying blossom values generated by a knot net is equivalent to specifying a polynomial in *B-basis*

form [10]. B-bases are the most important class of polynomial bases for CAGD: Bézier curves and surfaces, segments of B-splines, and B-patches are all special cases of polynomials in B-basis form. The familiar monomial basis is also closely related to the B-basis.

Although we will not be discussing B-splines in detail in this paper, it is worth noting that the blossom illustrates the connection between Bézier curves and B-splines. A degree n B-spline with knot vector $\{t_0, \ldots, t_{m-1}\}$ has control points $f(t_i, \ldots, t_{i+n-1})$ for $i = 0..m-n$. Thus, if we have the blossom, we can extract either Bézier or B-spline control points.

3 Blossoming Library

Blossoms can be made into datatypes very naturally because of two properties of blossoming analysis. First, blossoming gives a unified representation of different modeling paradigms: the representation of a curve or surface is given by a set of blossom arguments and a set of coefficients. How these two sets of parameters are assigned depends on the application. For example, they may be set interactively or as a result of filtering data. Second, blossoming gives a unified view of the operations on different modeling paradigms. Operations extract various kinds of information from the representation. For example, an application may want to obtain the location of various points on a surface, or derivatives, or a bounding box. Since a blossom is conceptually a function, obtaining any kind of information corresponds to evaluating the blossom at certain arguments.

3.1 Overview of Design

We wrote the Blossom Classes as a C++ library. Because it is a library, users can easily incorporate the Blossom Classes into their applications, and can use it with other tools, like matrix libraries. The Blossom Classes library is part of the Waterloo Computer Graphics Lab Splines project [1], and works with the datatypes in the Splines library. In addition, the Blossom Classes library is specifically designed to work with the Standard Template Library (STL) [15], which is a library of generic algorithms designed to work with many different classes. STL is part of the new C++ standard, and thus will be available to all C++ users.

The outstanding feature of the Blossom Classes library is that it works with user-supplied classes. Like STL, it is designed so that a datatype can have many implementations by different classes. The idea is to specify a datatype by an abstract interface, i.e., as a set of functions that a class must provide in order to implement that datatype. The library code manipulates the datatype using only the functions defined in the abstract interface. In our library, this abstract interface was implemented using the C++ template facility. While we provide one implementation of these abstract data types, users can integrate their own classes with the library simply by providing the functions required in the abstract interface. Users can also use this facility to create specialized implementations that are more efficient for certain applications.

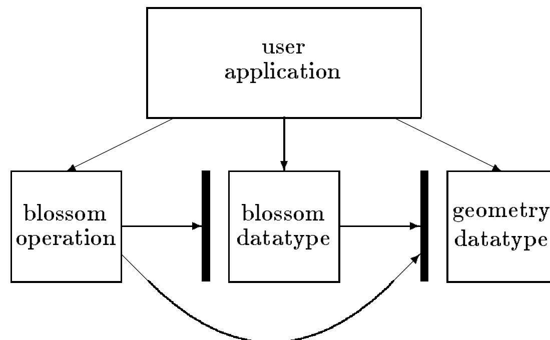

Figure 2: Components of Blossom Classes.

The overall design of the the library is shown in Figure 2. It has three components: the blossom datatypes, the geometric datatypes, and the operations on blossoms. The heart of the library is the blossom operations. These operations manipulate blossom objects using the abstract interfaces (shown as black bars in the diagram) for blossom and geometry datatypes. Blossom datatypes store geometry objects as arrays of knots and coefficients, also accessing them through their abstract interface. On the other hand, the user application directly uses any functionality provided by the actual classes (shown as boxes), rather than being limited to the functions in the abstract interface.

In the following sections, we describe each of these components in more detail. First, we look at the blossoming datatypes, and describe the functions that it must support in order for the blossom operations to work. In addition, we present an actual implementation of a blossoming datatype included in the library. Next, we give the same information for the geometry datatypes. Finally, we describe the blossom operations, which applications use to manipulate the blossoms. In the interests of brevity, we will only give the details about functionality used in

our examples; a full description can be found in [8].

3.2 Blossoming Datatypes

A blossom is simply an array of coefficients over a knot net. The user works with blossoms in a natural way by setting the knots and the coefficients directly. This design means the library only supports polynomials represented in B-basis. As arbitrary B-bases are supported, many useful paradigms can be implemented: Bézier curves or surfaces, B-splines, B-patches, monomials. However, other useful bases, like Lagrange bases, are not directly supported.

The abstract interface of blossom datatypes consists of the following:

- the definition for the following four classes: the class for the array of blossom coefficients, the class for the array of knots, the class for the geometry datatype for the domain and the geometry class for the range
- the functions `getDegree`, `getSpace`, `getCoeffs`, `getKnotNet`, and `makeBlossom`. The most important of these are the functions `getCoeffs` and `getKnotNet` which return the knots and coefficients of a blossom. Details about these two functions are given in the Blossom Operations section.

The library supplies an implementation of a blossom datatype in the `Blossom` class. The class provides blossoms of arbitrary dimension and degree. In addition, the class is a templated class with two template arguments: `Blossom<domain,range>`. The first argument is the geometry type to use as the domain of the blossom, while the second argument is the type to use as the range. Of course, other blossom classes can be used instead. For example, when the application only deals with curves, more efficient versions of blossoms can be implemented.

Section 4 contains some example applications that use the `Blossom` class, so here we describe the functionality of the class that is used in the examples.

The class provides several constructors. The constructor `Blossom<domain,range>(n)` creates a degree `n` blossom. The knots are initialized to be the Bézier knots over the standard basis in the domain. The coefficients are initially created using the default constructor for their class. The constructor `Blossom<domain,range>(b,n)` creates a blossom in Bézier basis form using the basis `b`. Other functions manipulate the knots and coefficients, such as `getCoeffs()`, `getCoeff(i)`, `setCoeff(i,pt)`, and `setKnot(k,l,pt)`.

Since the `Blossom` class implements arbitrary dimension blossoms, it indexes its coefficients using the `MultiIndex` class, which implements arbitrary dimension multi-indices. Thus, the variable `i` in the previous paragraph refers to a `MultiIndex` object. Operations on `MultiIndex` objects include addition (`i+j`), multiplication by an integer (`3*i`), `E(d)` which creates the multi-index $(0,\ldots,1,\ldots,0)$, where all elements are 0 except the `d`th element is 1.

Geometry Datatypes

The geometry component contains datatypes for working with domain spaces, range spaces, and scalars. We made these datatypes separate from the blossoming datatypes so that the actual implementation of geometric datatypes can change without any change in the blossoming datatypes. Different modeling paradigms, such as polynomial, homogeneous polynomial, or rational polynomial curves and surfaces, can be obtained by using different types for the geometry component,

The abstract interface of geometry datatypes consists of the following:

- the definition for the five classes fundamental to affine geometry: the geometric space, the basis, the points, the scalars, and the array used to extract coordinates
- the standard operations of affine geometry: `getDimension`, `getStdBasis`, `getCoordsFromBasis`, `getSpace`, `getBasisElement`, `setBasisElement`, and `makeBasis`, and `combination`. The most important of these are the operations `getCoordsFromBasis` and `combination`, which compute the linear combinations required by the blossom operations.

The library provides the `PtDomain` class, which implements linearized spaces of arbitrary dimension. The class can also be used as a projective range space. The library also supports the built-in type `double` as an efficient one-dimensional space.

The examples of Section 4 use the `PtDomain` class. The `PtDomain` class uses the `Pt` class as the point type. The `Pt` class provides convenient constructors for one, two, and three dimensional points by specifying their homogeneous coordinates with respect to the standard basis: `Pt(x,w)`, `Pt(x,y,w)`, `Pt(x,y,z,w)`. The function `cross` returns the cross product of two 3-D vectors. The functions `getStdFrame(d)` returns the standard Cartesian frame in `d`-dimensional space (e.g. (0,0,1),

(0,1,0), and (1,0,0) for 2-D space). The function `getStdSimplex(d)` returns the standard simplex (e.g. (0,0,1), (0,1,1), and (1,0,1) for 2-D space). (By convention, in homogeneous coordinates a vector has last coordinate of 0, and a point has last coordinate of 1.)

3.3 Blossom Operations

The heart of the Blossom Classes are the blossom operations. The blossom operations are all templated functions. They operate on blossoming datatypes through their abstract interfaces. The operations are useful basic building-block operations that can be used to implement different algorithms (see Section 4).

We present the operations in three parts: the operations for defining blossoms, the operations that evaluate blossoms, and the operations that manipulate the knots and coefficients of a blossom.

Defining Blossoms

To get a representation of a curve or surface one must assign the knots and the coefficients. These functions are not actually part of the blossom operations component, but rather are provided by the blossom datatype. They are described here since they are used in conjunction with the blossom operations.

The operation `f.setCoeff(i,P)` will set the `i`th coefficient of the blossom `f` to the range point `P`, where `i` is an index into the coefficient array. The operation `f.setKnot(k,l,x)` sets the `(k,l)` knot of `f` to the domain point `x`. The arguments `k` and `l` are `int`s; in the one dimensional case, if `k` is 0, we are setting a_l to `x`, and if `k` is 1, we are setting b_l to `x`.

Figure 3 shows the effect of these operations on a curve segment. The left diagram shows the original blossom; the middle diagram illustrates moving one coefficient; and the right diagram illustrates moving one knot. By convention, the segment is drawn over the interval $[a_0, b_0)$. Thus, in the left two pictures, the segment is over $[2, 3)$, but in the right-hand picture, the segment is over $[2, 4)$.

Evaluating Blossoms

Evaluating the blossom extracts information from the representation. Partial evaluation is also provided for reusing intermediate results of an evaluation.

The evaluation routines implement the algorithm illustrated in Figure 1. The algorithm is a generalization of the de Casteljau algorithm for Bézier curves and surfaces, and the de Boor algorithm for B-spline curves. It extends these algorithms to evaluate arbitrary blossom arguments for arbitrary dimension polynomials in B-bases form. In the same way, partial evaluation extends the Boehm knot-insertion algorithm.

The operation `P = eval(f,args)` evaluates the blossom `f` at the list of range points stored in `args` and return a point in the range. The `f` parameter is any blossom datatype object, and the `args` parameter is any STL *forward iterator* object that returns points in the domain.

The operation `P = diagonalEval(f,x)` evaluates a blossom at a single argument, `x`, n times.

The operation `f2 = partialEval(f, args, args_end, (blossom*)0)` returns a new blossom that is the partial evaluation of a blossom. The operation takes a blossom object `f` and returns an object of type `blossom`, where the type is indicated by the `(blossom*)0` parameter. The return type can be different from the type of `f`. The `args` parameter is an iterator that marks the beginning of the list of arguments, while `args_end` marks the end. With the variant `f2 = partialEval(f, pt, (blossom*)0)`, the blossom `f` is evaluated at the single argument `pt`.

Figure 4 shows the evaluation of a one-dimensional blossom. The left diagram illustrates a full evaluation; the middle diagram illustrates a partial evaluation of f resulting in f_2; and the right diagram illustrates a partial evaluation of f_2 resulting in f_3.

Swapping Knots

The knot swapping operations compute the coefficients of a blossom over an altered knot net. They are used to perform knot insertion and basis conversion. They employ an algorithm similar to partial evaluation, except that they they perform their calculations "in place," and do not need to allocate extra memory.

While these operations change both the coefficients and the knots of the blossom, they do not change the function that the blossom represents. That is, evaluating a blossom at the same arguments before and after swapping will yield the same results.

The operation `knotReplaceCoeffs(f,k,x)` computes the coefficients of the blossom `f` over a new knot net that has `x` at position `(k,n-1)`, where `n` is the degree of the blossom.

The operation `knotSwapCoeffs(f,k,from,to)` computes the coefficients of the blossom over the new knot net, where

1. the knot at `(k,from)` is moved to position

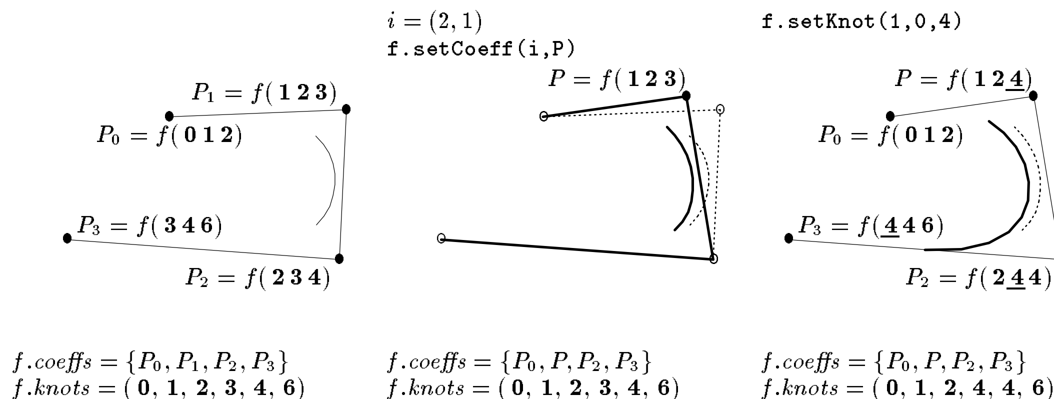

Figure 3: Operations for defining blossoms. The coefficients and knots resulting from each operation are shown below each figure.

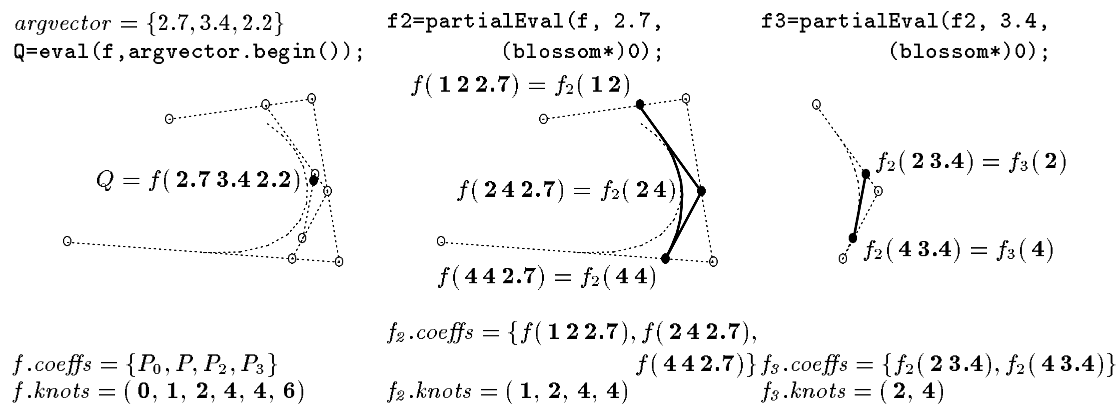

Figure 4: Operations for evaluating blossoms. The coefficients and knots resulting from each operation are shown below each figure.

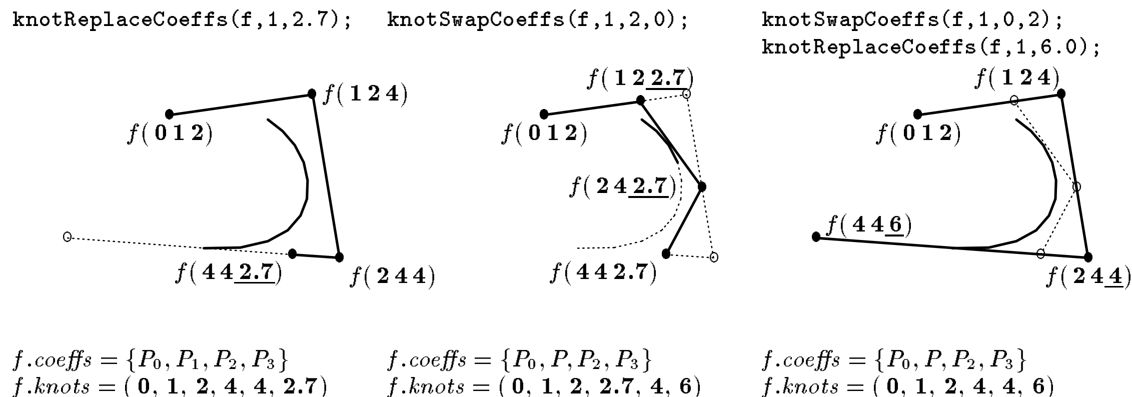

Figure 5: Operations for swapping knots of a blossom. The coefficients and knots resulting from each operation are shown below each figure.

(`k,to`), and

2. if `from<to`, the knots (`k,from+1`)..(`k,to`) are moved forward one position to (`k,from`)..(`k,to-1`), otherwise, they are moved backward one position.

The user must be careful in using these two operations as they may cause a knot net to become invalid (i.e. the B-basis may become degenerate).

Figure 5 shows the effect of these operations on a curve segment. In the left-hand picture, the knot net is changed from (0,1,2,4,4,6) to (0,1,2,4,4,2.7). This latter sequence does not correspond to a legal B-spline knot sequence, since the knots must be in increasing order in a B-spline. One result of this violation is that the curve segment over $[2,4)$ no longer fits inside the convex hull of the control points. The middle picture corresponds to the legal knot sequence (0,1,2,2.7,4,4). Note that the two operations of `knotReplaceCoeffs` and `knotSwapCoeffs` has effectively computed a knot insertion of the corresponding B-spline. The right-hand picture shows the reverse operation, knot deletion.

In the one-dimensional case, the two operations `knotReplaceCoeffs` and `knotSwapCoeffs` are usually used together because of the relationship to B-splines. However, these operations extend to higher dimensions, and in that case, there may be occasion to use them separately.

4 Evaluation of the System

This section aims to evaluate the Blossom Classes library's performance in practice by demonstrating its use in two different situations. The first example uses the Blossom Classes for manipulating monomials and B-patches to show that the library simplifies common CAGD computations. The second example implements several variations of the change of basis algorithms, demonstrating some of the more advanced features of the library.

4.1 Simple Demonstration

Monomials

This example shows how to use the library for manipulating polynomials in the familiar monomial basis. It prints out values of the polynomial x^3-2x^2+4x+3.

```
1 typedef Blossom<PtDomain,Pt> blossom;
  void main() {
    blossom f(getStdFrame(PtSpace(1)),3);
    Pt coeffs[] = { Pt(1,0), Pt(-2.0/3.0,0),
5                   Pt(4.0/3.0,0), Pt(3,1) };
    copy(coeffs,coeffs+4,f.getCoeffs().begin());
    for (double x = 0;x<=1;x+=.1)
      cout << diagonalEval(f,Pt(x,1)) << endl;
  }
```

Line 3 creates a cubic blossom whose knot net is $\{\vec{\delta}, \vec{\delta}, \vec{\delta}, \mathbf{0}, \mathbf{0}, \mathbf{0}\}$ (a Bézier basis over the standard 1-D frame). Lines 4–6 set the coefficients over this knot net. By the relation given in [8], the coefficients are $\{\vec{\delta}, -\frac{2}{3}\vec{\delta}, \frac{4}{3}\vec{\delta}, \mathbf{3}\}$. As discussed in that paper, every coefficient except the last is a vector. Lines 7 and 8 evaluate the polynomial in the interval [0,1].

Shaping and Tessellating a B-patch

The next example uses the library to make B-patches with different knots and coefficients. It demonstrates the three common operations of creating a blossom, setting knots and coefficients, and evaluating.

```
1 typedef Blossom<PtDomain,Pt> blossom;
  void tessellate(const blossom &f) {
    int n = f.getDegree();
    Pt u(1,0,0), v(0,1,0);
5   for (double x=0;x<=1;x+=.1) {
      for (double y=0;y<=1-x;y+=.1) {
        vector<Pt> argvector(n-1,Pt(x,y,1));
        blossom f2 = partialEval( f,
            argvector.begin(), argvector.end(),
10          (blossom*)0 );
        Pt norm = cross(diagonalEval(f2,u),
                        diagonalEval(f2,v));
        Pt pos = diagonalEval(f2,arg[0]);
        cout << pos << pos+norm << endl;
15    }
    }
  }

  void main() {
20  blossom f(getStdSimplex(PtSpace(2)),2);
    Pt coeffs[] = { Pt(0,0,0,1), Pt(1,0,0.5,1),
                    Pt(2,0,0,1), Pt(0,1,0.5,1),
                    Pt(1,1,1,1), Pt(0,2,0,1) };
    copy(coeffs,coeffs+6,f.getCoeffs().begin());
25  tessellate(f);
    f.setCoeff(E(0)+E(2),Pt(.5,1.5,.5,1));
    tessellate(f);
    f.setKnot(0,1,Pt(1,.5,1));
    tessellate(f);
30 }
```

On line 20, the main routine creates a quadratic blossom with knot net $\{(1,0,1),(1,0,1),(0,1,1),(0,1,1),(0,0,1),(0,0,1)\}$ (Bézier knot net over the standard simplex in 2-space). Lines 21–24 set up the initial coefficients, and line 25 calls the tessellate function. Lines 26–29 demonstrate moving a coefficient and a knot. In line 26, the expression `E(0)+E(2)` creates

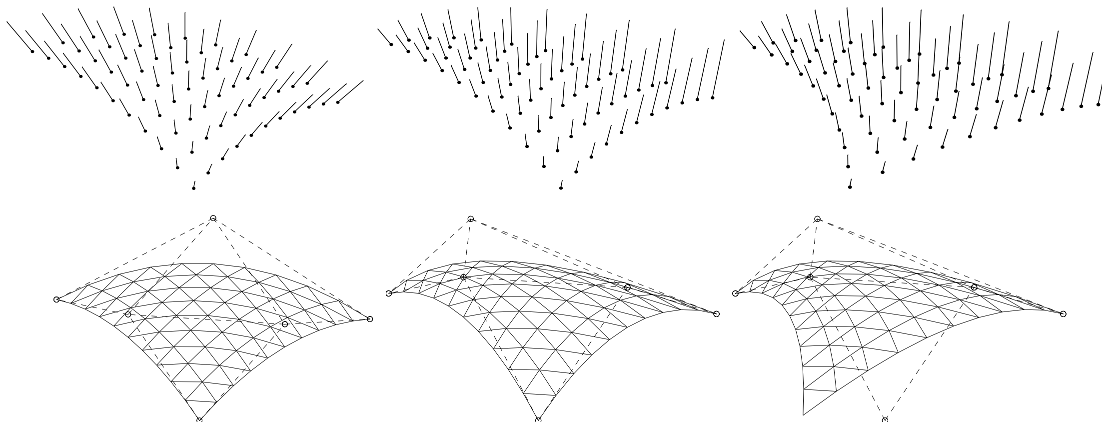

Figure 6: Results of simple example code.

the `MultiIndex` $(1,0,1)$. The effect is to move coefficient $P_{1,0}$ of `f`. The effect of these operations on the shape of the patch can be seen in Figure 6.

The `tessellate` function (lines 2–17) evaluates the points and normals on the surface over a tessellation. Lines 5 and 6 iterate over the tessellation of the standard simplex in increments of .1. The computation of the normal starts in lines 7 and 8 by partially evaluating `f` at the point `Pt(x,y,1)` $n-1$ times to get `f2`. Then, according to Equation 1, the directional derivatives are obtained by evaluating `f2` at `u` and `v` (lines 4,11,12). The normal is the cross product of the two directional derivatives. Finally, the point on the surface is obtained by reusing `f2` and evaluating at `Pt(x,y,1)` (line 13).

This program outputs a list of points (and normals) on the surface. The top row of Figure 6 displays these points and normals. To make the shape of the patches easier to see, on the bottom row, we drew only the points, with adjacent points connected; we also added the B-patch control nets for these surfaces.

Evaluation

As these simple examples demonstrate, common operations can be performed easily, in the obvious way. The operations have the same run-time and stability characteristics as the standard de Casteljau algorithm. However, since the algorithms are for B-patches, they are a little less efficient than the de Casteljau algorithm for Bézier patches.

4.2 Basis Conversion

As a more complex example of using the blossom package, we will develop an algorithm to convert polynomials from one B-basis representation to another. Basis conversion can be used to convert curve segments in Bézier or monomial form to B-spline form, and vice versa; or between B-patch and Bézier patch form. Basis conversion is also an important part of other algorithms, such as polynomial composition [11].

For simplicity, we solve the one-dimensional case. The following algorithms generalize to arbitrary dimensions. The problem, stated formally, is this: given the coefficients of f over the knot net $\{a_0,\ldots,a_{n-1},b_0,\ldots,b_{n-1}\}$, compute its coefficients over the knot net $\{a'_0,\ldots,a'_{n-1},b'_0,\ldots,b'_{n-1}\}$.

We will present three methods of basis conversion, and present the code to implement each method using our library. The purpose of these code blocks is a) to show some examples of the techniques required to produce efficient blossoming algorithms and b) to show that our library is capable of supporting these techniques.

For each approach, we give a diagram showing the number of affine combinations required by the algorithm. The number of affine combinations required for basis conversion depends on the degree of the polynomial and on the dimension of the various spaces involved. All three examples are for a degree 3 polynomial with a dimension 1 domain. In each computation diagram, the black circles represent the initial control points; the white circles are the control points for the new basis, and every vertex of a triangle (except the black points) requires one affine combination to compute.

One solution for basis conversion is to directly evaluate each of the new coefficients, $P_k = f(a'_{n-1-k}\cdots a'_0 b'_0 \cdots b'_k)$, over the new knots:

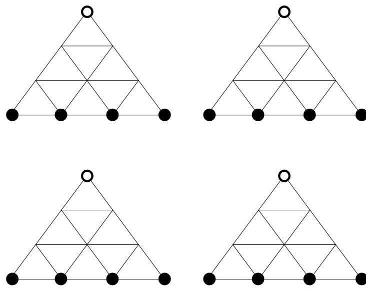

Figure 7: Simple basis conversion; performs 24 affine combinations. In this figure, the initial (black) points are $f(a_2a_1a_0)$, $f(a_1a_0b_0)$, $f(a_0b_0b_1)$, $f(b_0b_1b_2)$ for all four de Casteljau diagrams.

```
1  typedef Blossom<PtDomain,Pt> blossom;
   void basisConvert1(blossom &f,
                         Pt a[], Pt b[]) {
     int n = f.getDegree();
5    blossom g(n);
     for (int k=0;k<n;k++) {
       g.setKnot(0,k,a[k]);
       g.setKnot(1,k,b[k]);
     }
10   vector<Pt> argvector(n);
     for (i=0;i<n;i++) {
       Pt *p = copy(b,b+i,argvector.begin());
       copy(a,a+n-i,p);
       g.setCoeff( i*E(0)+(n-i)*E(1),
15                 eval(f,argvector.begin()) );
     }
     f = g;
   }
```

Line 5 creates a new blossom that will contain the new coefficients. Lines 6–9 set the knots of the new blossom. Line 11 iterates over each coefficient. Lines 12 and 13 set up the vector of arguments $(a'_{n-1-k}\cdots a'_0 b'_0 \cdots b'_k)$, and line 15 evaluates the arguments. Line 17 changes `f` to reflect the new knots and coefficients. The computation performed by this algorithm is shown in Figure 7.

This first solution is inefficient because the partial evaluations are recomputed each time. The next solution reuses the partial evaluations. This algorithm is the Sablonniere's basis conversion algorithm [13].

```
1  typedef Blossom<PtDomain,Pt> blossom;
   void basisConvert2(blossom &f,
                         Pt a[], Pt b[]) {
     int n = f.getDegree();
5    blossom g(n);
     for (int k=0;k<n;k++) {
       g.setKnot(0,k,a[k]);
```

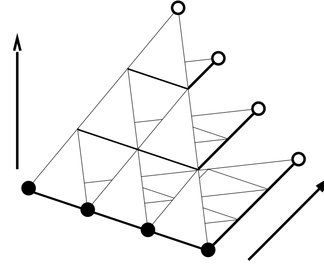

Figure 8: Sablonniere's basis conversion; performs 16 affine combinations.

```
       g.setKnot(1,k,b[k]);
     }
10   blossom f2 =f;
     for (int i=0;i<n;i++) {
       g.setCoeff(i*E(0)+(n-i)*E(1),
                  eval(f2,b,b+i,(blossom*)0));
       blossom f2 = partialEval(f2,a,a+n-i,
15                                 (blossom*)0);
     }
     g.setCoeff( n*E(0), f2.getCoeff(0*E(0)) );
     f = g;
   }
```

This routine is the same as `basisConvert1` up to line 9. For each $0 \leq i < n$, `f2` holds the partial evaluation of `f` on the arguments $a'_i \cdots a'_0$ (lines 10,14,15). Then, the coefficient of `g` is computed by finishing the evaluation on the arguments arguments $b'_{n-i} \cdots b'_0$ (lines 12,13,17). The computations performed by this algorithm is shown in Figure 8. This solution is still inefficient because the partial evaluations are thrown away each time.

The final solution uses knot swapping to replace each old knot with the corresponding new knot. This algorithm is Goldman's basis conversion algorithm for local B-spline bases [7]. If the knot nets are Bézier, the code is simply performing repeated subdivision.

```
1  typedef Blossom<PtDomain,Pt> blossom;
   void basisConvert3(blossom &f,
                         Pt a[], Pt b[]) {
     int n = f.getDegree();
5    for (int k=0;k<n;k++) {
       knotReplaceCoeffs(f,0,a[k]);
       knotSwapCoeffs(f,0,n-1,k);
     }
     for (k=0;k<n;k++)
10     knotReplaceCoeffs(f,1,b[k]);
       knotSwapCoeffs(f,1,n-1,k);
     }
   }
```

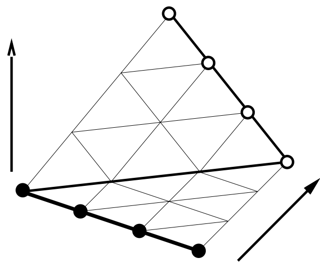

Figure 9: Goldman basis conversion; performs 12 affine combinations.

Lines 6 and 7 insert the knot a'_k into position $(0, k)$ of the knot net of `f`, displacing the old knot a_k. This is done for all k. Lines 10 and 11 does the same thing for b'_k. The computations performed by this algorithm is shown in Figure 9. Because of the way knot swapping works, this algorithm is much more efficient than the previous two.

Evaluation

This example shows the Blossom Classes is useful for experimenting with algorithms. There are many ways to achieve the same, mathematically equivalent result, yet each way leads to a different implementation. The Blossom Classes package makes it easy to obtain these implementations by translating the analysis into code in a direct way.

4.3 Other Algorithms

The blossoming package described in this paper is for polynomial functions. We also have developed a B-spline package (described in [8]), and used this B-spline package to develop a simple but efficient algorithm for degree raising b-splines [8, 9].

We have also used the blossom package for polynomial composition algorithms. In addition to providing straight-forward translation of mathematics to C++ code, the blossom package allowed us to easily test a new, optimal polynomial composition algorithm [8, 11].

5 Conclusions

Blossoming datatypes facilitate research by making modeling prototypes easier to write. The datatypes also make programs easier to read. They make it easy to see whether a program is correct by seeing whether the operations manipulate the concepts correctly. Thus, programs become easier to maintain and change.

We implemented the Blossom Classes library to provide these datatypes. Moreover, users can replace the Blossom Classes library's own datatypes with user-supplied datatypes. This ability to integrate user's datatypes into the library is important for two reasons: it means users can fit the library into existing applications, and users can specialize the datatypes for greater efficiency or extend them for greater functionality.

The blossoming package is available via anonymous ftp at `ftp.cgl.uwaterloo.ca` in `pub/software/blossom/`. See the `README` there for more details. This same package can be access on the World Wide Web at `ftp://ftp.cgl.uwaterloo.ca/software/blossom/`

6 Acknowledgments

We would like to thank the reviewers for their helpful comments that greatly improved this paper.

References

[1] Richard Bartels. Object oriented spline software. In Pierre-Jean Laurent, Alain Le Méhauté, and Larry L. Schumaker, editors, *Curves and Surfaces in gemetric design*, pages 27–34. A K Peters Ltd, 1994.

[2] Austin Dahl. Weyl: A language for computer graphics and computer aided geometric design. Technical Report TR 92-06-02, University of Washington, June 1992.

[3] Paul de Faget de Casteljau. *Formes à Pôles*, volume 2 of *Mathématiques et CAO*. Hermes, 51 rue Rennequin, 75017 Paris, 1985.

[4] Tony 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] Tony DeRose and Ronald Goldman. A tutorial introduction to blossoming. In H. Hagen and D. Roller, editors, *Geometric Modeling*. Springer, 1991.

[6] Gerald Farin. *Curves and Surfaces for Computer Aided Geometric Design*. Academic Press, third edition, 1990.

[7] Ronald Goldman and Phillip Barry. Wonderful triangle: A simple, unified algorithmic approach

to change of basis procedures in CAGD. In Tom Lyche and Larry L. Schumaker, editors, *Mathematical Methods in CAGD II*. Academic Press, Inc, 1992.

[8] Wayne Liu. Programming support for blossoming. Master's thesis, University of Waterloo, Waterloo, 1995. ftp://cs-archive.uwaterloo.ca/cs-archive/CS-95-40/CS-95-40.ps.Z.

[9] Wayne Liu. A simple, efficient algorithm for degree raising b-spline curves. Submitted for publication, 1996.

[10] Suresh Lodha and Ronald Goldman. A multivariate de Boor-fix formula. In Pierre-Jean Laurent, Alain Le Méhauté, and Larry L. Schumaker, editors, *Curves and Surfaces in Geometric Design*, pages 301–310. A K Peters Ltd, 1994.

[11] Stephen Mann and Wayne Liu. An analysis of polynomial composition algorithms. Technical Report CS-95-24, University of Waterloo, Waterloo, Ontario, N2L 3G1 CANADA, 1995.

[12] Lyle Ramshaw. Blossoming: A connect-the-dots approach to splines. Technical Report 19, Digital Equipment Corporation, Systems Research Centre, 21 June 1987.

[13] P. Sablonniere. Spline and Bézier polygons associated with a polynomial spline curve. *CAD*, 10(4):257–261, 1978.

[14] Philipp Slusallek, Reinhard Klein, Andreas Kolb, and Günther Greiner. Object oriented framework for curves and surfaces with applications. In Pierre-Jean Laurent, Alain Le Méhauté, and Larry L. Schumaker, editors, *Curves and Surfaces in gemetric design*, pages 457–466. A K Peters Ltd, 1994.

[15] Alexander Stepanov and Meng Lee. The standard template library. Technical Report HPL-94-34, Hewlett-Packard Labroatories, April 1994.