Università degli Studi di Roma Tre
Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79 – 00146 Roma, Italy

Transfinite Blending Made Easy

Alberto Paoluzzi

RT-DIA-40-99 **Febbraio 1999**

Università "Roma Tre",
Via della Vasca Navale, 79
00146 Roma, Italy.

This work was partially supported by MURST.

Email: paoluzzi@dia.uniroma3.it

ABSTRACT

In this paper transfinite interpolation/approximation is discussed. This is a powerful approach to generation of curves, surfaces and solids (and even higher dimensional manifolds) by blending lower dimensional geometric objects. Transfinite blending, e.g. used in Gordon-Coons patches, is well known to mathematicians and CAD people. It is presented here in a very simple conceptual and computational framework, which leads such a powerful modeling to be easily handled by the non mathematically sophisticated user of graphics techniques. Transfinite blending is discussed in this paper by making use of a very powerful and simple functional language for geometric design.

1 Introduction

Parametric curves and surfaces, as well splines, are usually defined (see e.g. [3]) as vector-valued functions generated from some vector space of polynomials or rationals (i.e. ratio of polynomials) over the field of real numbers. In this paper it is conversely presented an unified view of curves, surfaces, and multivariate manifolds as vector-valued functions generated from the same vector spaces, but over the field of polynomial (or rational) functions itself. This choice implies that the *coefficients* of the linear combination which uniquely represents a curved mapping in a certain basis are not real numbers, as usually, but vector-valued functions.

This approach is a strong generalization, which contains the previous ones as very special cases. For example, the standard approach of Hermite interpolation for curves, where two extreme points and tangents are interpolated, can so be applied to surfaces, where two extreme curves of points are interpolated with assigned derivative curves, or even to volume interpolation of two assigned surfaces with assigned normals. Notice that such an approach is not new, and is quite frequently used in CAD applications, mainly to ship and airplane design, since from the times that Gordon-Coons patches were formulated [5, 7]. It is sometime called *function blending* [7, 9], or *transfinite interpolation* [8, 6].

Transfinite interpolation, that the author preferes to call *transfinite blending*, because it can also be used for approximation, is usually quite hard to handle by using standard imperative languages (usually the aged, lovely and fashioned Fortran). In particular, it is quite difficult to be abstracted, and too often *ad hoc* code must be developed to handle the specific application class or case. A strong mathematical background is also needed to both implement and handle such kind of software. This fact strongly discouraged the diffusion of such a powerful modeling technique outside the close neighbourhood of automobile, ship and airplane shell design.

The original contribution of this paper was both in using a general algebraic setting which simplifies the description of transfinite blending by the use of functions without variables, and in embedding such an approach into a modern functional computing environment [11], where functions can be easily multiplied and added exactly as numbers. This results in an amazing descriptive power when dealing with parametric geometry. Several examples of this power are given in the

paper. Consider, e.g., that multivariate transfinite Bezier blending of any degree with both domain and range spaces of any dimension is implemented (Section 3.2) with 11 lines of quite readable source code.

Last but not least, in the paper we limited our exposition to Bézier and Hermite cases for sake of space. Actually the same approach can be applied to any kind of parametric representation of geometry, including splines. Notice in particular that different kinds of curves surfaces and splines can be freely blended, so giving a kind of design freedom rarely seen before.

2 Background

2.1 Parametric geometry

Parametric representation of geometric objects is concerned with a mapping Φ between normed vector spaces X and Y, where

$$\Phi : U \to Y, \qquad U \subset X.$$

In most useful cases both X and Y are the support of Euclidean spaces, say E^p and E^q, with $p \leq q$, which can be identified with X and Y, respectively. Φ is often used as a simplicial mapping. This one is applied to some simplicial decomposition Σ of the domain U to generate a simplicial approximation $\Phi(\Sigma)$ of the image set $\Phi(U) \subset Y$. For example, an approximation with linear triangles of a parametric surface embedded in 3D is the common output of the modeling process of such a surface.

The generation of a smooth picture of the curved object $\Phi(U) \subset Y$ is usually left to the graphics hardware used for rendering. Illumination and shading techniques are there applied to either the colors or the normals of the vertices of $\Phi(\Sigma)$ to generate a picture of $\Phi(U)$ with the appropriate appearance of smoothness.

If the dimension of the domain space X is $p = 1$, then the geometric object $\Phi(U)$ is a curve embedded in a q-dimensional space. If $p = 2$, then $\Phi(U)$ is a surface embedded in a q-dimensional space. With $q = 2, 3$ the set $\Phi(U)$ is either a *plane* curve/surface or a *space* curve/surface, respectively.

2.2 Some `PLaSM` elements

`PLaSM` is a geometry-oriented extension of a subset of the functional language `FL`

developed by Backus, Williams and others at IBM Research [1, 2]. In the PLaSM language design the powerful algebraic approach to programming of FL was combined with a dimension-independent treatment of embedded data structures and geometric algorithms [10, 4, 12]. For sake of readibility, a table with the meaning of the few language constructs used in this paper is given in the Appendix.

A primitive MAP construct is given in PLaSM to generate a simplicial approximation $\Phi(\Sigma)$ of the point-set $\Phi(U)$, with $\Phi : U \to E^q$

$$\texttt{MAP:VectFun:Dom}$$

where VectFun is a CONSed component-wise expression of a parametric mapping, of the kind

$$\texttt{CONS} :< \Phi_1, \ldots, \Phi_q >$$

which evaluates to the vector-valued mapping function

$$\Phi_1 + \ldots + \Phi_q = \Phi \quad \text{such that} \quad \Phi : \Sigma(\texttt{Dom}) \to E^q,$$

and where Dom is a cell decomposition of any polyhedral subset of U. The semantics of a MAP expression is the following: (a) a *simplicial* complex Σ(Dom) which decomposes the Dom polyhedron is generated; (b) the function VectFun is applied to all vertices of such simplicial complex, so generating a simplicial approximation $\Phi(\Sigma(\texttt{Dom}))$ of the result. Clearly, VectFun must have so many component functions as the dimension of the target space E^q.

Some transfinite Bézier and Hermite maps are given as examples in the paper, by using only a minimal PLaSM subset. For a description of syntax and semantics of the language the reader is referred to [11]. Notice that ":" and "$\sim$" stand for function application and composition. Such PLaSM notations closely correspond to the standard functional notations, so that

$$\texttt{(g} \sim \texttt{f):x} \quad \text{stands for} \quad (g \circ f)(x).$$

Notice also that the operators "+", "$-$", "$*$" and "/" denote algebraic operations between either numbers or functions. As usual, with standard mathematical notation:

$$(f + g)(u) = f(u) + g(u), \quad (fg)(u) = f(u)g(u), \quad \text{and so on.}$$

The corresponding PLaSM notation would be:

$$\texttt{(f + g) : u = f : u + g : u}, \quad \texttt{(f * g) : u = f : u * g : u}, \quad \text{and so on.}$$

The *selector* functions `S1`, `S2`, ..., `Sn` are applied to sequences, and respectively select the first, second and n-th component of the sequence they are applied to:

$$\texttt{S2} :< \texttt{a}, \texttt{b}, \texttt{c}, \texttt{d} >= \texttt{b}.$$

Finally notice that the predefined `K` function is a very useful constructor of *constant* functions, and will be largely used in the sequel. With standard functional notation and with `PLaSM` notation, we have, respectively:

$$\begin{aligned} \kappa(a)(x) = (\kappa(a))(x) = a \qquad & \text{for any } x, \\ \texttt{K} : \texttt{a} : \texttt{x} = (\texttt{K} : \texttt{a}) : \texttt{x} = \texttt{a} \qquad & \text{for any } \texttt{x}. \end{aligned}$$

3 Generating geometry in function spaces

Parametric maps $\Phi : U \to Y$ used in Computer Graphics and CAD usually belong to the space of rational (i.e. ratio of polynomials) functions of bounded integer degree n. Since the space $\mathcal{Z}_n$ of such functions is a finite-dimensional vector space over the field $\mathcal{Z}_n$ itself, then each $\Phi \in \mathcal{Z}_n$ can be expressed uniquely as a linear combination of $n+1$ basis functions $\phi_i \in \mathcal{Z}_n$ with coordinate functions $\chi_i \in \mathcal{Z}_n$, so that

$$\Phi = \chi_0\phi_0 + \cdots + \chi_n\phi_n.$$

Hence a unique coordinate representation

$$\Phi = (\chi_0, \ldots, \chi_n)_{\mathcal{B}}$$

of the mapping is given, after a basis $\mathcal{B} = \{\phi_0, \ldots, \phi_n\} \subset \mathcal{Z}_n$ has been chosen. The *power* basis, the *cardinal* (or *Lagrange*) basis, the *Hermite* basis, the *Bernstein/Bézier* basis and the *B-spline* basis are the most common and useful choices for such a basis.

The coordinate functions χ_i may be easily generated, as will be explained in the following subsections, by using the "geometric handles" of the mapping, usually data points $p_i \in Y$, to be interpolated or approximated by the set $\Phi(U)$.

Only greek letters, either capitals or lower-case, will be used in the sequel to denote functions. Please notice that B and H are also greek upper-case letters for β and η, respectively.

3.1 Univariate case

Let consider the simple univariate case $\Phi : U \subset X \to Y$, where the dimension p of domain X is one. To generate the coordinate functions χ_i it is sufficient to transform each data point $p_i \in Y$ into a constant vector-valued function, so

$$\chi_i = \kappa(p_i), \quad \text{where} \quad \kappa(p_i) : U \to Y : u \mapsto p_i.$$

Using the functional notation with explicit variables, the constant function is such that

$$\kappa(p_i)(u) = p_i$$

for each parameter value $u \in U$. The (higher-level) constructor `K` of such constant functions plays a fundamental role in `PLaSM`, as well as in its parent language `FL`, as will be shown in Example 3.1.

Example 3.1 (Cubic Bézier curve in the plane) *The cubic Bézier plane curve depends on four points* `p0,p1,p2,p3` $\in E^2$, *which are given as formal parameters of the function* `Bezier3` *which generates the* Φ *mapping. The local functions* `b0,b1,b2,b3` *implement the Bernstein/Bézier basis functions* $\beta_k^3 = \binom{3}{k} u^k (1-u)^{3-k}$, $0 \leq k \leq 3$.

```
DEF Bezier3 (p0,p1,p2,p3::IsSeq) =
  [ (x:p0 * b0) + (x:p1 * b1) + (x:p2 * b2) + (x:p3 * b3),
    (y:p0 * b0) + (y:p1 * b1) + (y:p2 * b2) + (y:p3 * b3) ]
  WHERE
    b0 = u1 * u1 * u1,
    b1 = K:3 * u1 * u1 * u,
    b2 = K:3 * u1 * u * u,
    b3 = u * u * u,
    x  = K~S1, y = K~S2, u1 = K:1 - u, u = S1
  END;
```

The `x` *and* `y` *functions, defined as composition of a selector with the constant function constructor* `K`, *respectively select the first (second) component of their argument sequence and transform such a number in a constant function.*

A polygonal approximation of the Bézier curve with 20 *line segments is finally generated by evaluating the expression*

```
MAP: (Bezier3:<<0,2>,<1,2>,<1,0>,<2,0>>): (Domain:20);
```

where a 1D polyhedral complex which subdivides the $[0,1]$ *interval into* n *adjacent line segments is generated by the* PLaSM *function:*

```
DEF Domain (n::IsIntPos) = QUOTE:(#:n:(1/n));
```

3.2 Multivariate case

When the dimension p of the domain space X is greater than one, two main approaches can be used to construct a parametric mapping Φ. The first approach is the well-known "tensor-product" method; the second approach corresponds to "function blending", also called "transfinite blending".

Tensor-product method In tensor-product method, both coordinate functions (generated by data-points) and basis functions can be arranged into tensors $\chi^{n_1,\ldots,n_p}$ and $\phi^{n_1,\ldots,n_p}$ depending on p indices. The mapping function is so given by the scalar product of such function tensors:

$$\Phi = \chi^{n_1,\ldots,n_p} \cdot \phi^{n_1,\ldots,n_p} = \sum_{\substack{i_1=0,\ldots,n_1 \\ \cdots \\ i_p=0,\ldots,n_p}} \chi_{i_1,\ldots,i_p}\phi_{i_1,\ldots,i_p}.$$

More importantly, each multivariate basis function is generated by tensor product from the univariate basis of the same kind:

$$\phi_{i_1,\ldots,i_p}(u_1,\ldots,u_p) = \phi_{i_1}(u_1)\phi_{i_2}(u_2)\cdots\phi_{i_p}(u_p)$$

A well-known example is that of bicubic Bézier surface patches in three-space, where $U = [0,1]^2$, $p = 2$, $q = 3$ and $n_1 = n_2 = 3$.

Transfinite blending method Let consider a multivariate mapping $\Phi : U \to Y$, where $U \subset X$ and X is a p-dimensional space. Since Φ depends on p parameters, in the following will be denoted as Φ^p.

In *transfinite blending* Φ^p is computed by linear combination of $n_p + 1$ maps (depending on $p-1$ parameters) with the *univariate* basis of degree n_p. In other words:

$$\Phi^p = \Phi_0^{p-1}\phi_0 + \cdots \Phi_{n_p}^{p-1}\phi_{n_p}.$$

The coordinate representation of Φ with respect to the basis $\mathcal{B} = (\phi_0, \ldots, \phi_{n_p})$ is so given by $n_p + 1$ maps depending on $p - 1$ parameters:

$$\Phi^p = \left(\Phi_0^{p-1}, \ldots, \Phi_{n_p}^{p-1}\right)_{\mathcal{B}}$$

As an example of transfinite blending consider the generation of a bicubic Bézier surface mapping $B(u_1, u_2)$ as a combination of four Bézier cubic curve maps $B_k(u_1)$, with $0 \leq k \leq 3$, where some curve maps may possibly reduce to a constant point map:

$$B(u_1, u_2) = \sum_{k=0}^{3} B_k(u_1)\ \beta_k^3(u_2)$$

where

$$\beta_k^3(u) = \binom{3}{k} u^k\,(1-u)^{3-k}\,, \qquad 0 \leq k \leq 3,$$

is the Bernstein/Bézier cubic basis. Analogously, a three-variate Bézier solid body mapping $B(u_1, u_2, u_3)$, of degree n_3 on the last parameter, may be generated by univariate Bézier blending of surface maps $B_k(u_1, u_2)$, some of which possibly reduced to a curve map or even to a constant point map:

$$B(u_1, u_2, u_3) = \sum_{k=0}^{n_3} B_k(u_1, u_2)\ \beta_k^{n_3}(u_3)$$

The more interesting aspects of such approach are flexibility and simplicity. Conversely than in tensor-product method, there is no need that all component geometries have the same degree, and even neither that were all generated using the same function basis. For example, a quintic Bézier surface map may be generated by blending both Bézier curve maps of lower (even zero) degree together with Hermite and Lagrange curve maps. Furthermore, it is much simpler to combine lower dimensional geometries (i.e. maps) than to meaningfully assembly the multi-index tensor of control data (i.e. points and vectors) to generate multivariate manifolds with tensor-product method.

Transfinite Bézier blending The amazing descriptive power of the `PLaSM` functional approach to geometric design is made evident here, where transfinite Bézier blending of any degree is implemented in few lines of code, by easily combining coordinate maps which may depend on any number of parameters.

Notice in fact that the `Bézier` function given here can be used to blend points to give curve maps, to blend curve maps to give surface maps, to blend surface

maps to give solid maps, and so on. Notice also that the given implementation is independent on the dimensions p and q of domain and range spaces X and Y.

At this purpose, first a small toolbox of related functions is needed, to compute the factorial $n!$, the binomial coefficients $\binom{n}{i}$, the Bernstein/Bézier functions $\beta_i^n(u)$ and the $\mathcal{B}(u)(n)$ Bernstein basis of degree n, with

$$\mathcal{B}(u)(n) = \{\beta_0^n(u), \beta_1^n(u), \ldots, \beta_n^n(u)\}$$

```
DEF Fact (n::IsInt) = *:(CAT:<<1>, 2..n>);
DEF Choose (n,i::IsInt) = Fact:n / (Fact:i * Fact:(n-i));
DEF Bernstein (u::IsFun)(n::IsInt)(i::IsInt) =
  *~[K:(Choose:<n,i>),**~[ID,K:i], **~[-~[K:1,ID],K:(n-i)]]~u;
DEF BernsteinBase (u::IsFun)(n::IsInt) = AA:(Bernstein:u:n):(0..n);
```

Then the `Bezier:u` function is given, to be applied on the sequence of `ControlData`, which may contain either control points p_k or control maps Φ_k^{p-1}. In the former case each component of each control point is firstly transformed into a constant function. The body of the `Bezier:u` function just linearly combines component-wise the sequence (χ_k) of coordinate functions generated by the expression `(TRANS~fun):ControlData` with the basis sequence (β_k^n) generated by `BernsteinBase:u:degree`, where the `degree` n equates the number of geometric handles minus one.

```
DEF Bezier (u::IsFun) (ControlData::IsSeq) =
  (AA:(+~AA:*~TRANS) ~ DISTR):
    < (TRANS~fun):ControlData, BernsteinBase:u:degree >
WHERE
  degree = LEN:ControlData - 1,
  fun = (AA~AA):(IF:<IsFun,ID,K>)
END;
```

It is much harder to explain in few words what actual argument to pass (and why) for the formal parameter `u` of the `Bezier` function. As a rule of thumb let pass either the selector `S1` if the function must return a univariate (curve) map, or `S2` to return a bivariate (surface) map, or `S3` to return a threevariate (solid) map, and so on.

Example 3.2 (Bézier curves and surface) *Four Bézier curve maps* `C1`*,* `C2`*,* `C3`*, and* `C4` *of degrees* $1, 2, 3$ *and* 2 *are respectively defined as:*

```
DEF C0 = Bezier:S1:<<0,0,0>,<10,0,0>>;
DEF C1 = Bezier:S1:<<0,2,0>,<8,3,0>,<9,2,0>>;
DEF C2 = Bezier:S1:<<0,4,1>,<7,5,-1>,<8,5,1>,<12,4,0>>;
DEF C3 = Bezier:S1:<<0,6,0>,<9,6,3>,<10,6,-1>>;
```

It may be useful to notice that the control points have three coordinates, so that the generated maps C1, C2, C3, *and* C4 *will have three component functions. Such maps can be blended with the Bernstein/Bézier basis to produce a cubic bivariate (surface) mapping:*

$$B(u_1, u_2) = \texttt{C0}(u_1)\beta_0^3(u_2) + \texttt{C1}(u_1)\beta_1^3(u_2) + \texttt{C2}(u_1)\beta_2^3(u_2) + \texttt{C3}(u_1)\beta_3^3(u_2).$$

Such a linear combination of coordinate functions with the Bézier basis (this time working on the second coordinate of points in U domain) is performed by the PLaSM *function* Sup1, *defined in the following, by using again the* Bezier *function:*

```
DEF Surf1 = Bezier:S2:<C0,C1,C2,C3>;
```

A simplicial (in this case with triangles) approximation of the surface $B([0,1]^2) \subset E^3$ is finally generated by evaluating the PLaSM *expression*

```
MAP: (CONS:Surf1): (Domain:20 * Domain:20);
```

Notice that the primitive function CONS, *applied to the sequence of component functions generated by evaluating* Surf1, *produces a single vector-valued function. According to the semantics of the* MAP *operator,* CONS:sup1 *is applied to all vertices of the automatically generated simplicial decomposition Σ of the 2D product polyhedron* $(\texttt{Domain}:20 * \texttt{Domain}:20) \subset E^2$. *A simplicial approximation $B(\Sigma)$ of the surface $B([0,1]^2) \subset E^3$ is finally produced.*

The four generating curves and the generated cubic transfinitely blended surface are displayed in Figures 1a and 1b. It is possible to show that such surface interpolates the four boundary curves defined by the extreme control points, exactly as in the case of tensor-product method, but obviously with much greater generality, since any defining curve may be of any degree.

4 Transfinite Hermite interpolation

The cubic Hermite univariate map is the unique cubic polynomial $H : [0,1] \to E^q$ which matches two given points $p_0, p_1 \in E^q$ and derivative vectors $t_0, t_1 \in \Re^q$ for

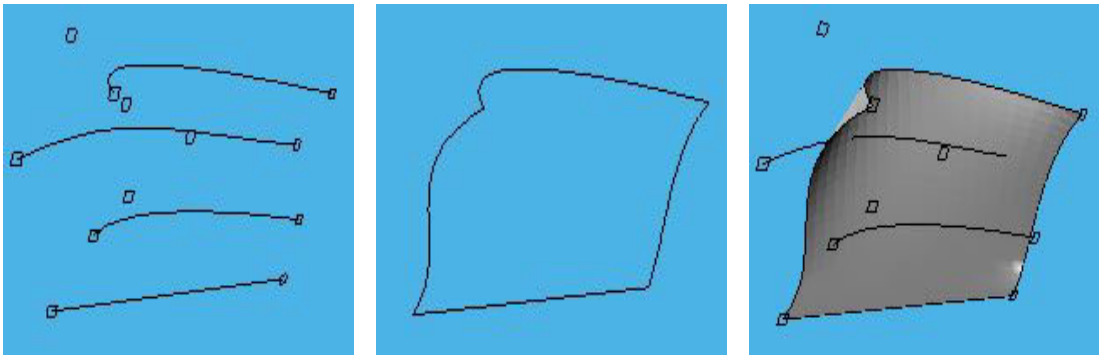

Figure 1: (a) Graphs of the four Bézier curve maps $\texttt{c0}(u_1)$, $\texttt{c1}(u_1)$, $\texttt{c2}(u_1)$ and $\texttt{c3}(u_1)$; (b) graphs of $\texttt{c0}(u_1)$ and $\texttt{c3}(u_1)$ together with the two Bézier maps $\texttt{b1}(u_2)$ and $\texttt{b2}(u_2)$ generated by the extreme control points; (c) graph of the surface $\texttt{Sur1}(u_1, u_2)$, joined to the graphs of the previous curves.

$u = 0, 1$ respectively. Let denote as $\mathcal{H}(3) = (\eta_0^3, \eta_1^3, \eta_2^3, \eta_3^3)$ the cubic Hermite function basis, with

$$\eta_i^3 : [0,1] \to \Re, \qquad 0 \leq i \leq 3,$$

and such that

$$\eta_0^3(u) = 2u^3 - 3u^2 + 1, \quad \eta_1^3(u) = 3u^2 - 2u^3, \quad \eta_2^3(u) = u^3 - 2u^2 + u, \quad \eta_3^3(u) = u^3 - u^2.$$

Then the mapping H can be written as

$$\begin{aligned} H &= \chi_0\ \eta_0^3 + \chi_1\ \eta_1^3 + \chi_2\ \eta_2^3 + \chi_3\ \eta_3^3 \\ &= \kappa(p_0)\ \eta_0^3 + \kappa(p_1)\ \eta_1^3 + \kappa(t_0)\ \eta_2^3 + \kappa(t_1)\ \eta_3^3. \end{aligned}$$

It is easy to verify, for the univariate case, that:

$$\begin{aligned} H(0) &= \kappa(p_0)(0) = p_0, \quad H(1) = \kappa(p_1)(1) = p_1, \\ H'(0) &= \kappa(t_0)(0) = t_0, \quad H'(1) = \kappa(t_0)(1) = t_1, \end{aligned}$$

and that the image set $H([0,1])$ is the desired curve in E^q.

The multivariate transfinite Hermite map H^q is easily defined by allowing the coordinate functions $(\chi_k)_{\mathcal{H}}$ to be any map depending on at most $p-1$ parameters.

Cubic transfinite Hermite implementation A cubic transfinite `Hermite3` mapping is implemented in `PLaSM`, where four data objects are given as formal

parameters. Such data objects may be either points/vectors, i.e. sequences of numbers, or $1/2/3/n$-variate maps, i.e. sequences of (curve/surface/solid/etc) component functions, or even both, as will be shown in the following examples.

```
DEF Hermite3 (u::IsFun) (p1,p2,t1,t2::IsSeq) =
  (AA:(+~AA:*~TRANS)~DISTL):
    <<h0,h1,h2,h3>, (TRANS~fun):<p1,p2,t1,t2>>
WHERE
  h0 = k:2 * u3 - k:3 * u2 + k:1,
  h1 = k:3 * u2 - k:2 * u3,
  h2 = u3 - k:2 * u2 + u,
  h3 = u3 - u2,  u3 = u*u*u,  u2 = u*u,
  fun = (AA~AA):(IF:<IsFun,ID,K>)
END;
```

4.1 Bivariate surface by cubic blending of curves

Example 4.1 (Grid generation) *Two Hermite curve maps* `c1` *and* `c2` *are defined, so that the 3D curves* `c1`$([0,1])$ *and* `c2`$([0,1])$ *are restrained to* $z = 0$ *plane.*

```
DEF c1 = Hermite3:S1:<<1,0,0>,<0,1,0>,<0,3,0>,<-3,0,0>>;
DEF c2 = Hermite3:S1:<<0.5,0,0>,<0,0.5,0>,<0,1,0>,<-1,0,0>>;
```

Some different grids are easily generated from the plane surface which interpolates the curves `c1` *and* `c2`*. At this purpose it is sufficient to apply the* `Hermite3:S2` *function to different tangent curves.*

```
DEF d = (AA:-~TRANS):<c2,c1>;
DEF grid1 = Hermite3:S2:<c1,c2,d,d>;
DEF grid2 = Hermite3:S2:<c1,c2,<-0.5,-0.5,0>,d>;
DEF grid3 = Hermite3:S2:<c1,c2,<S1:d,-0.5,0>,d>;
```

The grids generated by maps `grid1`*,* `grid2` *and* `grid3` *are shown in Figure 2. The tangent map* `d` *is simply obtained as vector difference of the curve maps* `c1` *and* `c2`*.*

It is interesting to notice that the map `grid1` *can be also generated as linear (transfinite) Bézier interpolation of the two curves. Clearly the solution as cubic Hermite is more flexible, as it is shown by Figures 2b and 2c.*

```
DEF grid1 = Bezier:S2:<c1,c2>;
MAP:(CONS:grid1):(Domain:8 * Domain:8);
```

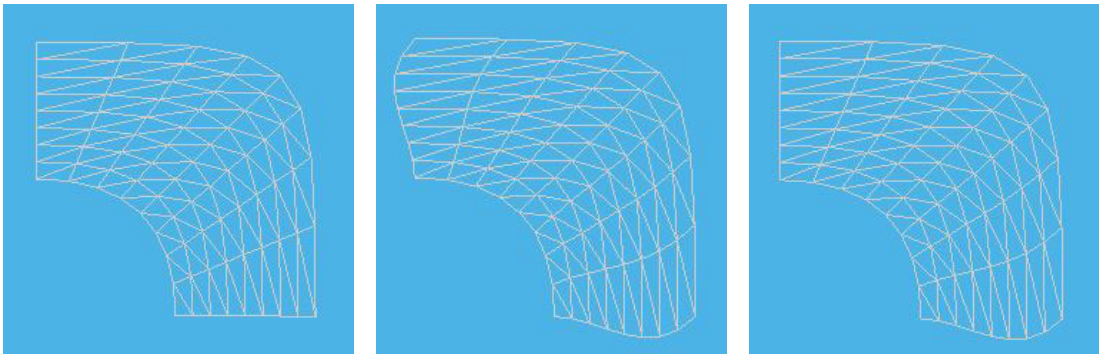

Figure 2: The simplicial complexes generated by the `MAP` operator on the `grid1`, `grid2` and `grid3` maps given in Example 4.1.

Example 4.2 (Surface interpolation of curves) *The curve maps* `c1` *and* `c2` *are here transfinitely interpolated by a* `Sur2` *mapping via cubic Hermite, with the further constraints that the tangent vectors along the first curve are constant and parallel to* $(0,0,1)$*, whereas along the second curve are also constant and parallel to* $(0,0,-1)$*. The resulting map*

$$\texttt{Sur2} : [0,1]^2 \to E^3$$

has unique representation as

$$\texttt{Sur2} = \texttt{c1}\ \eta_0^3 + \texttt{c2}\ \eta_1^3 + (\kappa(0),\kappa(0),\kappa(1))\ \eta_2^3 + (\kappa(0),\kappa(0),\kappa(-1))\ \eta_3^3.$$

Such a map is very easily implemented by the following `PLaSM` *definition. A simplicial approximation* $\texttt{Sur2}(\Sigma)$ *of the point-set* $\texttt{Sur2}([0,1]^2)$ *is generated by the* `MAP` *expression and is shown in Figure 3.*

```
DEF Sur2 = Hermite3:S2:< c1,c2,<0,0,1>,<0,0,-1> >;
MAP: (CONS:Sur2): (Domain:14 * Domain:14);
```

Example 4.3 (Surface interpolation of curves) *A different surface interpolation of the two plane curves* `c1` *and* `c2` *is given by the following script, where the boundary tangent vectors are constrained to be constant and parallel to* $(1,1,1)$ *and* $(-1,-1,-1)$*, respectively. Some pictures of the resulting surface are given in Figure 4.*

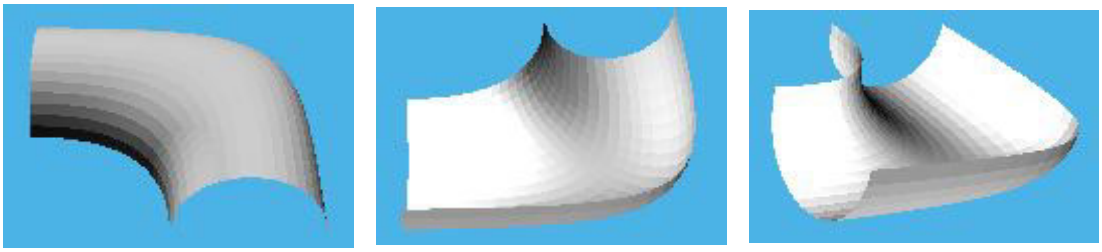

Figure 3: Some pictures of the surface interpolating two plane Hermite curves with constant vertical tangent vectors along the curves.

```
DEF Sur3 = Hermite3:S2:<c1,c2,<1,1,1>,<-1,-1,-1>>;
MAP: (CONS:Sur3): (Domain:14 * Domain:14);
```

Figure 4: Some pictures of a new surface interpolating the same Hermite curves with constant oblique tangent vectors.

4.2 Threevariate volume by cubic blending of surfaces

Generation of threevariate curved volume by cubic Hermite interpolation of extreme surfaces is explored in this section. At this purpose first two new curves, with quite small variations with respect to those of Section 4.1 are given, together with their interpolating surface `Sur4`.

```
DEF cc1a = Hermite3:S1:<<1,0,0.6>,<0,1,0.6>,<0,3,0>,<-3,0,0>>;
DEF cc2a = Hermite3:S1:<<0.5,0,1>,<0,0.5,1>,<0,1,1>,<-1,0,0>>;
DEF Sur4 = Hermite3:S2:<cc1a,cc2a,<1,1,1>,<-1,-1,-1>>;
```

Then a cell-decomposed polyhedral subset `SubDom1` $\subset U = [0,1]^3 \subset E^3$ is defined. A simplicial decomposition of it is shown in Figure 5a. Finally a cubic threevariate Hermite map `Vol1` : $U \to E^3$ which interpolates with constant

normal vectors the surfaces `sur2` and `sur4` is given. The graph of the simplicial mapping `Vol1`(Σ(`SubDom1`)), produced by the `MAP` expression, is shown in Figure 5b.

```
DEF SubDom1 = Domain:8 * Domain:8 * @0:(Domain:3);
DEF Vol1 = Hermite3:S3:<sur2,sur4,<1,1,1>,<-1,-1,-1>>;
MAP: (CONS:Vol1): SubDom1 ;
```

Another straightforward example of curved subset produced by this threevariate Hermite mapping is given by first producing a boundary subset `SubDom2` of the unit standard interval $U = [0,1]^3 \subset E^3$. In this case

$$\texttt{SubDom2} = [0,1] \times [0,1] \times \{0,1\} \cup [0,1] \times \{0,1\} \times [0,1]$$

is defined by four boundary faces. Then such domain subset is ...to the same $\texttt{Vol1} : U \to E^3$ mapping. The domain subset and the result of the mapping are shown in Figure 6c and Figure 6d, respectively. It is easy to notice that such a mapping produce a self-intersecting deformed volume. Notice also that `@0` is the `PLaSM` denotation for the extractor operator of the 0-skeleton of a polyhedral complex [12].

```
DEF SubDom2 = STRUCT:<
  Domain:8 * Domain:8 * @0:(Domain:1),
  Domain:8 * @0:(Domain:1) * Domain:8 >;
MAP: (CONS:Vol1): SubDom2 ;
```

Example 4.4 (Wing section grid)

```
DEF c1 = Hermite3:S1:<<0,0>,<10,0>,<0,3>,<6,-1>>
DEF c2 = Hermite3:S1:<<0,0>,<10,0>,<0,-3>,<6,-1>>;
DEF b1 = Hermite3:S1:<<-5,0>,<15,0>,<0,22>,<0,-22>>;
DEF b2 = Hermite3:S1:<<-5,0>,<15,0>,<0,-22>,<0,22>>;

DEF Norm (fun2::IsSeq) = <-~yu/den, xu/den>
WHERE
  xu = S1:fun2, yu = S2:fun2, sqr = ID * ID,
  den = MySQRT~+~AA:sqr~[-~yu, xu],
  MySQRT = IF:<EQ~[K:0,ID], K:0, SQRT>
END;
```

Figure 5: Arguments and corresponding graphs of a threevariate transfinite cubic Hermite map $[0,1]^3 \to E^3$ which interpolates two given bivariate (surface) maps.

Figure 6: Arguments and corresponding graphs of a threevariate transfinite cubic Hermite map $[0,1]^3 \to E^3$ which interpolates two given bivariate (surface) maps.

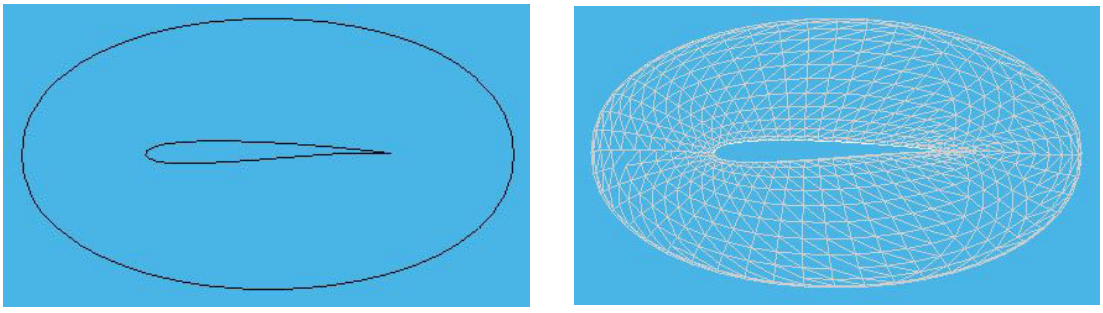

Figure 7: Figure of the grid decomposition of the field domain of the wing.

```
    DEF dom1 = T:1:1E-8:(Domain:24);
    DEF dom2 = T:1:1E-8:(Domain:12);
    DEF graph1 (f::IsSeq) = MAP:(CONS:f):dom1;
    DEF graph2 (f::IsSeq) = MAP:(CONS:f):(dom1 * dom2);
    DEF grid1 = Hermite3:S2:<c1,b1,Norm:c1,Norm:b1>;
    DEF grid2 = Hermite3:S2:<c2,b2,Norm:c2,Norm:b2>;

(STRUCT~CAT):<AA:graph2:<grid1,grid2>, AA:graph1:<c1,c2,b1,b2>>;
```

5 Conclusion

Transfinite blending of parametric geometry was presented here in a generalized but very simplified conceptual and computational framework. Such an approach lacks of efficient evaluation paradigms, like recursive subdivision. Also, our current implementation of the language is not so efficient as one would like. Anyway, the total balance of programming, modeling and evaluation times is highly positive, and great improvements can be forecast. Among the graphics applications of this approach to be fully explored we cite the (transfinite) morphing of parametrically generated geometric objects. Please try and enjoy!

Acknowledgment

I am indebted with Chris Hoffmann for asking some years ago if PLaSM, initially designed for polyhedral modeling in building design, could also manage curved geometries. This question started me thinking about the functional background of the language and discovering its amazing descriptive power when dealing with parametric geometry. I would also like to thank Luigi Morino and Umberto Iemma for posing a challenging problem of transfinite interpolation related to avionic design.

The *complete* implementation of the methods and examples here discussed is given in this paper by using PLaSM [11]. The interested reader is referred to the language pages at URL http://www.dia.uniroma3.it/plasm. All the figures were produced on a Macintosh PowerPC by grabbing the pictures generated by Intervista WorldView VRML plug-in for Netscape Navigator. VRML files are the common output from the PLaSM interpreter.

References

[1] Backus, J. "Can Programming Be Liberated from the Von Neumann's Style? A Functional Style and its Algebra of Programs". *Communications of the ACM*, **21**(8): 613–641, August 1978. (ACM Turing Award Lecture).

[2] Backus, J., Williams, J.H. and Wimmers, E.L. "An Introduction to the Programming Language FL". In *Research Topics in Functional Programming*, D.A. Turner (Ed.), Addison-Wesley, Reading, MA, 1990.

[3] Bartels, r.H., Beatty, J.C. and Barsky, B.A. "An Introduction to Splines for Use in Computer Graphics & Geometric Modeling". Morgan Kaufmann, Los Altos, CA, 1987.

[4] Bernardini, F., Ferrucci, V., Paoluzzi, A. and Pascucci, V. "A Product Operator on Cell Complexes". *Proc. of the ACM/IEEE 2nd Conf. on Solid Modeling and Appl.*, ACM Press, 43–52, February 1993.

[5] Coons, S.A. "Surfaces for Computer-Aided Design of Space Forms". *Tech. Rep. MAC-TR-41*, MIT, Cambridge, 1967.

[6] Goldman, R.N. "The Role of Surfaces in Solid Modeling". In *Geometric Modeling: Algorithms and New Trends*, G.E. Farin (Ed.), SIAM Publications, Philadelphia, Pennsylvania, 1987.

[7] Gordon, W.J., "Blending function Methods of bivariate and multivariate interpolation and approximation". *Res. Rep. GMR-834*, General Motors, Warren, Michigan, 1968.

[8] Gordon, W.J., "Spline-blended Surface interpolation through curve networks". *J. Math. Mech.*, **18**:931–952, 1969.

[9] Lancaster, P. and Salkauskas, K. "Curve and Surface Fitting. An Introduction". Academic Press, London, UK, 1986.

[10] Paoluzzi, A., Bernardini, F., Cattani, C. and Ferrucci, V. "Dimension-Independent Modeling with Simplicial Complexes". *ACM Trans. on Graphics*, **12**(1):56–102, January 1993.

[11] Paoluzzi, A., Pascucci, V., and Vicentino, M. "Geometric programming: a programming approach to geometric design". *ACM Trans. on Graphics*, **14**(4):266–306, July 1995.

[12] Pascucci, V., Ferrucci, V., and Paoluzzi, A. "Dimension-Indipendent Convex-Cell based HPC: Skeletons and Product". *International Journal of Shape Modeling*, **2**(1):37–67, January 1996.

List of PLaSM symbols

Function	Use	Value
MAP	MAP:VectFun:dom	graph of simplicial map
CONS	CONS:<f_1,...,f_n>	[f_1,...,f_n]
[f_1,...,f_n]	[f_1,...,f_n]:x	<f_1:x,...,f_n:x>
:	f:x	apply f to x
~	(g ~ f):x	g:(f:x)
+ - * /	(g + f):x	g:x + f:x
K	K:a:x	a (for any x)
S1	S1:<a,b,c,d>	a
QUOTE	QUOTE:<x_1,...,x_n>	1D polyhedron
#	#:n:x	<x,...,x> (n times)
CAT	CAT:<<a>,<b,c>,<>,<d>>	<a,b,c,d>
ID	ID:x	x (for any x)
AA	AA:f:<x_1,...,x_n>	<f:x_1,...,f:x_n>
TRANS	TRANS:<<1,2,3>,<a,b,c>>	<<1,a>,<2,b>,<3,c>>
DISTL	DISTL:<x,<a,b,c>>	<<x,a>,<x,b>,<x,c>>
LEN	LEN:<x_1,...,x_n>	n
IF	IF:<p,f,g>:x	if (p:x=true) f:x else g:x
IsSeq	IsSeq:<1,2,3>	True
IsFun	IsFun:K	True
IsInt	IsInt:10	True
STRUCT	STRUCT:<pol_1,...,pol_n>	polyhedral complex
@0	@0:pol	0-dim polyhedron

Keyword	Use	Meaning
DEF	DEF name (params::pred) = body	function definition
WHERE	WHERE local defs END	local functional envrmnt
=	name = body	local definition
body	expression	value
	expression	last_value