12

Parametric surfaces and solids

Parametric curves can be combined to give parametric surfaces or solids or higher dimensional manifolds. This chapter is mainly aimed at introducing *generative methods* for parametric surfaces as vector functions of two real parameters, but also presents extensions to three-variate solids and to n-variate manifolds. In particular, some useful classes of surfaces are discussed, i.e. the *profile product* surfaces, including rotational surfaces; the *ruled* surfaces, including generalized cylinders and cones; the surfaces generated by *tensor product* of curves or splines. *Transfinite blending* is a powerful approach to dimension-independent generation of manifold maps by blending lower dimensional geometric maps, as univariate interpolation or approximation in functional spaces. Transfinite blending includes the *skinning* of curve or spline grids with *Coons' patches*, transfinite Bézier and B-splines. The transfinite approach is presented here in a simple functional framework, with the result that this powerful generative tool may be easily handled by the common graphics user. Once more, the simple combinatorial semantics of the language may give the reader useful insights into tensor operations and transfinite combinations.

12.1 Introduction

A surface as a point set in Euclidean space $\mathbb{E}^3$ can be defined *implicitly* as a level set, usually the zero set, of a continuous scalar field $s : \mathbb{E}^3 \to \mathbb{R}$, i.e. as the set $s^{-1}(0)$.

Often more usefully, a surface in $\mathbb{E}^n$ may be defined *parametrically* as the image of a function $\boldsymbol{S} : U \to \mathbb{E}^n$ of two real parameters, i.e. with $U \subset \mathbb{R}^2$.

In this chapter we are mainly interested in the surface generating map $\boldsymbol{S}$, which we call *surface map*, or even simply *surface*, thus denoting the Euclidean point set as the *surface image*. When the parametric domain U is bounded, $\boldsymbol{S}(U)$ is called a surface *patch*. Surface points in $\boldsymbol{S}(U)$ are therefore associated with *coordinate pairs* $(u, v) \in U$. This association is mainly useful where it is one-to-one, i.e. where the surface is said to be *regular*.

Geometric Programming for Computer-Aided Design Alberto Paoluzzi
 ISBN 0-471-89942-9

12.1.1 Parametric representation

A surface as a point locus can be described, at least locally, as the image of a point-valued function

$$\boldsymbol{S} : U \to \mathbb{E}^n, \qquad n \geq 2$$

defined on an open set $U \subset \mathbb{R}^2$. Therefore we have

$$(u, v) \mapsto \begin{pmatrix} x_1(u,v) & x_2(u,v) & \cdots & x_n(u,v) \end{pmatrix}^T$$

and

$$\boldsymbol{S} = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \end{pmatrix}^T$$

where the $x_i : \mathbb{R}^2 \to \mathbb{R}$, $1 \leq i \leq n$, are called the *coordinate functions* of the surface.

Regularity The $\boldsymbol{S}$ surface is said to be *regular* in $(u, v) \in U$ if

1. the coordinate functions have continuous partial derivatives in (u, v);
2. the vectors

$$\begin{aligned} \boldsymbol{S}^u(u,v) &= \begin{pmatrix} \partial^u x_1(u,v) & \partial^u x_2(u,v) & \ldots & \partial^u x_n(u,v) \end{pmatrix} \\ \boldsymbol{S}^v(u,v) &= \begin{pmatrix} \partial^v x_1(u,v) & \partial^v x_2(u,v) & \ldots & \partial^v x_n(u,v) \end{pmatrix} \end{aligned}$$

are linearly independent. For a 3D surface this implies

$$\boldsymbol{S}^u(u,v) \times \boldsymbol{S}^v(u,v) \neq \boldsymbol{0}.$$

The $\boldsymbol{S}$ surface is said *regular on* U if it is regular for each $(u, v) \in U$.

Example 12.1.1 (3D sphere)
The parametric equation of the unit sphere in $\mathbb{E}^3$ can be derived starting from a half circumference in the $y = 0$ subspace, i.e. from the curve $\boldsymbol{c}(u) = \boldsymbol{R}_y(u)\, \boldsymbol{e}_1$. By using, from Section 6.3.1, the matrix of the $\boldsymbol{R}_y(u)$ rotation tensor about the y axis, we have:

$$\boldsymbol{c}(u) = \begin{pmatrix} \cos u & 0 & \sin u \\ 0 & 1 & 0 \\ -\sin u & 0 & \cos u \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos u \\ 0 \\ -\sin u \end{pmatrix}, \qquad -\frac{\pi}{2} \leq u \leq \frac{\pi}{2}.$$

So, the parametric equations of the sphere $S_2 = \{\boldsymbol{p} \in \mathbb{E}^3 : \|\boldsymbol{p}\| = 1\}$ can by generated as $\boldsymbol{R}_z(v)\, \boldsymbol{c}(u)$, i.e. by rotating $\boldsymbol{c}(u)$ about the z axis:

$$\boldsymbol{S}(u,v) = \begin{pmatrix} \cos v & -\sin v & 0 \\ \sin v & \cos v & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos u \\ 0 \\ -\sin u \end{pmatrix} = \begin{pmatrix} \cos u \cos v \\ \cos u \sin v \\ -\sin u \end{pmatrix}$$

with $-\frac{\pi}{2} \leq u \leq \frac{\pi}{2}$ and $-\pi \leq v \leq \pi$.

Let us consider where $\boldsymbol{S}(u, v)$ is regular. We have

$$\boldsymbol{S}^u(u,v) = \begin{pmatrix} -\sin u \cos v \\ -\sin u \sin v \\ -\cos u \end{pmatrix} \quad \text{and} \quad \boldsymbol{S}^v(u,v) = \begin{pmatrix} -\cos u \sin v \\ \cos u \cos v \\ 0 \end{pmatrix},$$

so that

$$\boldsymbol{S}^u(u,v) \times \boldsymbol{S}^v(u,v) = \det \begin{pmatrix} \boldsymbol{e}_1 & \boldsymbol{e}_2 & \boldsymbol{e}_3 \\ -\sin u \cos v & -\sin u \sin v & -\cos u \\ -\cos u \sin v & \cos u \cos v & 0 \end{pmatrix}$$

$$= (\cos^2 u \cos v)\, \boldsymbol{e}_1 + (\cos^2 u \sin v)\, \boldsymbol{e}_2 - (\sin u \cos u)\, \boldsymbol{e}_3.$$

Since $\|\boldsymbol{S}^u \times \boldsymbol{S}^v\| = |\cos u|$, then $\boldsymbol{S}$ is regular except where $\cos u = 0$, which holds only at the north and south poles.

12.2 Notable surface classes

A powerful approach to shape specification denoted as "generative modeling" was proposed by Snyder and Kajiya [Sny92, SK92]. As in other procedural methods of computer graphics,[1] shapes are described procedurally.

The power of the generative approach comes from the use of several operators to combine shapes, mainly parametric curves, to generate a great number of different kinds of surfaces and deformable solid models. Such a generative approach gives a natural, compact and symbolic representation to a large class of curved objects.

Some useful classes of surfaces are discussed in this section, and in particular *profile product* surfaces, *rotational* surfaces and *ruled* surfaces, including generalized *cylinders* and *cones*.

12.2.1 Profile product surfaces

The simplest generative method is perhaps the *profile product* [Bar81], where a surface $\boldsymbol{S}(u,v) = (S_1(u,v),\ S_2(u,v),\ S_3(u,v))$ is generated in $\mathbb{E}^3$ by affinely transforming a *cross-section* plane curve according to a plane *profile* curve.

In particular, a surface $\boldsymbol{S}$ is called the *profile product* of two plane curves $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, embedded into two coordinate subspaces and respectively called *profile* and *cross-section* curve

$$\boldsymbol{\alpha}(u) = \begin{pmatrix} \alpha_1(u) & 0 & \alpha_3(u) \end{pmatrix}^T$$

$$\boldsymbol{\beta}(v) = \begin{pmatrix} \beta_1(v) & \beta_2(v) & 0 \end{pmatrix}^T$$

when it is of the form

$$\boldsymbol{S}(u,v) = \begin{pmatrix} \alpha_1(u)\,\beta_1(v) & \alpha_1(u)\,\beta_2(v) & \alpha_3(u) \end{pmatrix}^T$$

Note We like to notice that the α_1 coordinate function is actually used as a scaling coefficient, whereas α_3 is used as a translation coefficient in the z direction. In fact, the above equation can be generated as:

$$\boldsymbol{S}(u,v) = \begin{pmatrix} \alpha_1(u) & 0 & 0 \\ 0 & \alpha_1(u) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \beta_1(v) \\ \beta_2(v) \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \alpha_3(u) \end{pmatrix} = \begin{pmatrix} \alpha_1(u)\,\beta_1(v) \\ \alpha_1(u)\,\beta_2(v \\ \alpha_3(u) \end{pmatrix}$$

[1] E.g., the POSTSCRIPT language [Ado85].

Example 12.2.1 (Surface by profile product)
In Script 12.2.1 we generate a surface as the profile product of two cubic Bézier curves, respectively defined as `alpha` and `beta`, that are both generated by the `Bezier` function, given in Script 11.2.5, as vector functions.

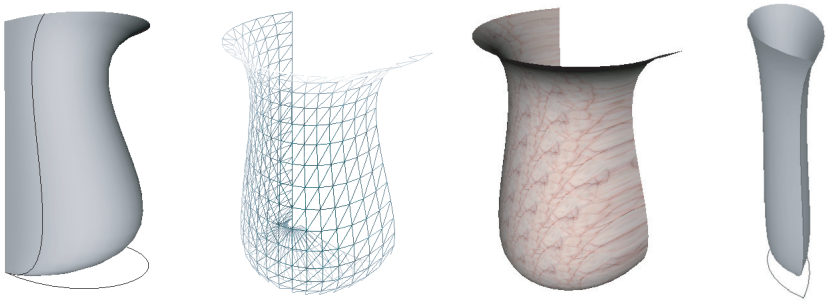

Figure 12.1 (a) Profile and section curve and generated surface (b) 1-skeleton (c) Texture-mapped surface (d) surface generated by same profile and different section curve

The reader should notice the composition of the components of the `section` function with the `[S2]` function, which allows application of the curve generating map `section` on the second coordinate of domain points.

Script 12.2.1 (Surface by profile product)

```
DEF ProfileProdSurface (profile, section::IsSeqOf:IsFun) = < fx, fy, fz >
WHERE
   fx = s1:profile * s1:section ~ [S2],
   fy = s1:profile * s2:section ~ [S2],
   fz = s3:profile
END;
```

Notice that the local definitions `fx`, `fy` and `fz` correspond to the three components of the parametric function $\boldsymbol{S}(u, v)$.

The following Script 12.2.2 produces the assembly of both generating curves and their profile product surface shown in Figure 12.1a. The `intervals` generator of 1D polyedral complexes is given in Script 11.2.1.

Example 12.2.2
Two plane curves $\mathbb{R}^1 \to \mathbb{E}^3$ playing the role of a *cross-section* and a *profile* in a surface product are shown in Figure 12.2a. Such curves are modeled as Bézier functions, using the code given in Section 12.4.1. The cross-section is obtained grouping four patches (rotated by multiples of 90°) of the `cross_function`. The profile curve is also a Bézier function.

The surfaces shown in Figure 12.2(b)and 12.2(c) are respectively generated by the following `PLASM` expressions:

Script 12.2.2 (Example of profile product surface)

```
DEF alpha = Bezier:<<0.1,0,0>,<2,0,0>,<0,0,4>,<1,0,5>>;
DEF beta = Bezier:<<0,0,0>,<3,-0.5,0>,<3,3.5,0>,<0,3,0>>;

STRUCT:<
   MAP:alpha:(Intervals:1:20),
   MAP:beta :(Intervals:1:20),
   MAP:(ProfileProdSurface:< alpha, beta >):
      (Intervals:1:20 * Intervals:1:20)
   >;
```

Script 12.2.3

```
DEF section = (Bezier ∼ TRANS):<<1,1,a,c,b,d,0>, <0,d,b,c,a,1,1>, #:13:0 >
WHERE
   a = 0.6, b = 0.6, c = 1.5, d = 0.2
END;

DEF profile = (Bezier ∼ TRANS):
   < <b,b,d,da,f,h,l,n,p,r,a,a,a>, #:13:0,
   <0,c,e,ea,g,i,m,o,q,s, hh-(2*delta),hh-delta,hh> >
WHERE
   a = 1, hh = 19, delta = 1.5, b = 8, c = 6, d = -3, e = 5,
   da = -3, ea = 9, f = 14, g = 8, h = 12, i = 18, l = -7, m = 13,
   n = -2.5, o = 14, p = 5, q = 12.5, r = 5, s = 18.5
END;
```

where `domain` is a uniform partition of the $[0,1]^2$ interval, and `lines_domain` is a set of segments there contained.

Rotational surfaces

When a smooth profile curve defined in xz plane, i.e.

$$\boldsymbol{\alpha}(u) = \begin{pmatrix} f(u) & 0 & g(u) \end{pmatrix}^T$$

is rotated about the z axis, a *rotational surface* is obtained:

$$\boldsymbol{S}(u,v) = \boldsymbol{R}_z(v)\,\boldsymbol{\alpha}(u) = \begin{pmatrix} \cos v & -\sin v & 0 \\ \sin v & \cos v & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} f(u) \\ 0 \\ g(u) \end{pmatrix} = \begin{pmatrix} f(u)\cos v \\ f(u)\sin v \\ g(u) \end{pmatrix}.$$

In this case it is easy to see that:

Script 12.2.4

```
MAP:(ProfileSurface:< section,profile >):lines;
ProfileSurface:<cross_function,profile_function,trimmed_domain>;
MAP:(ProfileSurface:< section,profile >):domain;
```

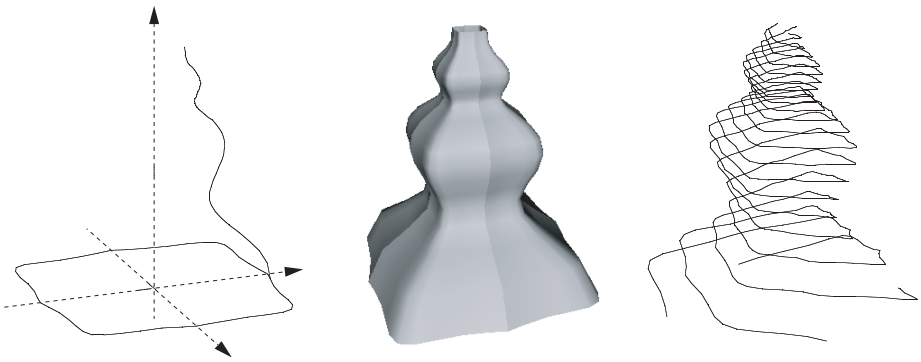

Figure 12.2 `ProfileSurface` product of the two plane curves: (a) `crossSection` and `profile` curves (b) surface (c) surface mapping applied to a set of domain lines

1. the surface $\boldsymbol{S}(u, v)$ is everywhere *regular* if $f(u) \neq 0$ and $\boldsymbol{\alpha}'(u) \neq \mathbf{0}$ for each u;
2. where the surface is regular, the curves corresponding to constant values of parameters u and v intersect orthogonally each other.

Note Notice that rotational surfaces are a special case of profile product surfaces, where the cross-section curve is a circle of unit radius, i.e. is defined as $\boldsymbol{\beta}(v) = (\cos(v),\ \sin(v),\ 0)^T$.

Implementation Rotational surfaces are generated by the `RotationalSurface` operator of Script 12.2.5, with formal parameter the `ProfileCurve`, given as a sequence of two coordinate functions.

Script 12.2.5 (Rotational surface)

```
DEF RotationalSurface (ProfileCurve::IsSeqOf:IsFun) =
   < f * cos ~ s2, f * sin ~ s2, g >
WHERE
   f = s1:ProfileCurve,
   g = s2:ProfileCurve
END;
```

Example 12.2.3 (Rotational surface)
A rotational surface is produced by Script 12.2.6 starting from a non-uniform B-spline profile curve. Both the profile curve and the generated surface are displayed in Figure 12.3.

Notice that the `ProfileCurve` symbol in Script 12.2.6 contains the pair of coordinate functions returned by `Blend`(ing) the `BsplineBasis` of order `4`, generated by the knot sequence `<0,0,0,0,1,2,3,4,4,4,4>`, with the sequence of 2D control points `<<0.1,0>,<2,0>,<6,1.5>,<6,4>,<2,5.5>,<2,6>,<3.5,6.5>>`. The cellular decomposition of surface `Domain` is generated by Cartesian product of the 1D polyhedral complexes produced by the `Intervals` function given in Script 11.2.1.

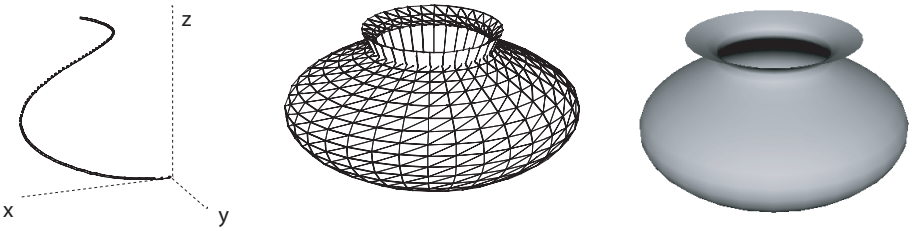

Figure 12.3 Revolution surface: (a) profile curve (b) surface tessellation (c) smooth surface shading

The faceted polyhedral approximation `out` of the rotational surface is finally exported to the VRML file `out.wrl` by specifying also a *crease angle* $\frac{\pi}{2}$, so that some browsers may render it with Gouraud shading, as shown in Figure 12.3c.

Script 12.2.6 (Example of rotational surface)

```
DEF ProfileCurve = Blend:(BsplineBasis:4:<0,0,0,0,1,2,3,4,4,4,4>):
   <<0.1,0>,<2,0>,<6,1.5>,<6,4>,<2,5.5>,<2,6>,<3.5,6.5>>;

DEF domain = Intervals:4:19 * Intervals:(2*PI):32;
DEF out = MAP:(RotationalSurface:ProfileCurve):domain;

VRML:(out CREASE (PI/2)):'out.wrl';
```

12.2.2 Ruled surfaces

A surface is said to be *ruled* when each surface point belongs to a straight line, that is entirely contained in the surface. Ruled surfaces may be thought of as generated by the motion of a straight line.

A simple and general vector expression can be given for the parametric form of ruled surfaces. If $\boldsymbol{\alpha}(u)$ is a curve that crosses all the surface lines, and $\boldsymbol{\beta}(u)$ is a vector oriented as the line crossing $\boldsymbol{\alpha}(u)$, then the ruled surface is

$$\boldsymbol{S}(u,v) = \boldsymbol{\alpha}(u) + v\boldsymbol{\beta}(u) \tag{12.1}$$

Implementation A `PLaSM` dimension-independent implementation of equation 12.1 is straightforward, and is given in Script 12.2.7, as a `RuledSurface` function of two parameter curves `alpha` and `beta`, passed as sequences of coordinate functions. Let us note that the vector operations defined in Chapter 3, i.e. `vectSum`, `vectDiff`, `scalarVectProd` and so on, were for this purpose defined as working between either vectors of numbers or vectors of functions.

Script 12.2.7 (Ruled surfaces)

```
DEF RuledSurface (alpha,beta::IsSeqOf:IsFun) =
   alpha vectSum (S2 scalarVectProd beta);
```

Example 12.2.4 (Hyperbolic paraboloid)
The surface called *hyperbolic paraboloid* has implicit equation $x^2 - y^2 + 2z = 0$, and regular parametrization given by

$$\boldsymbol{S}(u,v) = \begin{pmatrix} u-v & u+v & uv \end{pmatrix}^T,$$

so that we can write:

$$\begin{aligned} \boldsymbol{\alpha}(u) &= \begin{pmatrix} u & u & 0 \end{pmatrix}^T \\ \boldsymbol{\beta}(u) &= \begin{pmatrix} -1 & 1 & u \end{pmatrix}^T \\ \boldsymbol{S}(u,v) &= \boldsymbol{\alpha}(u) + v\boldsymbol{\beta}(u) = \begin{pmatrix} u-v & u+v & uv \end{pmatrix}^T \end{aligned}$$

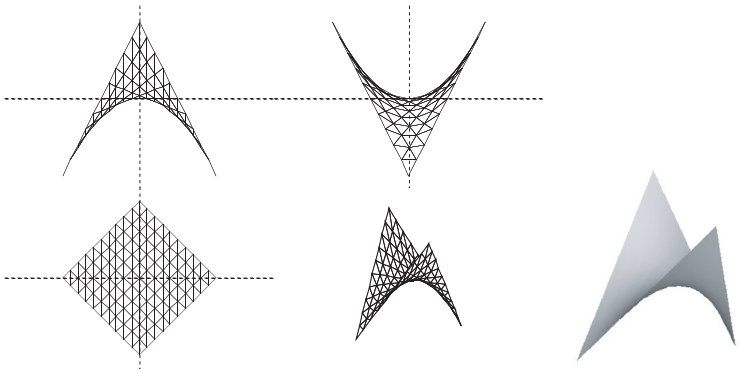

Figure 12.4 The patch $\boldsymbol{S}[-1,1]^2$ of hyperbolic paraboloid

The implementation of this ruled surface is given in Script 12.2.8. The surface patch resulting from the mapping $[-1,1]^2 \to \mathbb{E}^3$ is shown in Figure 12.4.

Script 12.2.8 (Hyperbolic paraboloid)

```
DEF domain = T:<1,2>:<-1,-1>:((Intervals:2:10) * (Intervals:2:10));
DEF Paraboloid =
   MAP:(RuledSurface:< <S1, S1, K:0>, <K:-1, K:1, S1> >):domain


VRML:(Paraboloid CREASE (PI/2)):'out.wrl'
```

Example 12.2.5 (Linear interpolation of curves)
In this example a ruled surface interpolating two arbitrary curves $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$ is generated. In this case the direction of the rules is given by $\boldsymbol{c}_2 - \boldsymbol{c}_1$, so that we have:

$$\boldsymbol{S}(u,v) = \boldsymbol{\alpha}(u) + v\,\boldsymbol{\beta}(u) = \boldsymbol{c}_1(u) + v\,(\boldsymbol{c}_2(u) - \boldsymbol{c}_1(u)).$$

where, clearly, $\boldsymbol{c}_2(u) - \boldsymbol{c}_1(u)$ is the direction of the line passing for $\boldsymbol{c}_1(u)$.

In particular, In Script 12.2.9 we generate the ruled surface interpolating a 3D Bézier cubic `c1` and a circle arc `c2` embedded in $\mathbb{E}^3$. The resulting surface is shown in Figure 12.5. Notice that the circle generating function is reparametrized by the mapping $[0,1] \rightarrow [0, \frac{3}{2}\pi]$ (see Section 5.1.2), and that the polyhedral `domain` $[0,1]^2$ is partitioned into 20×1 subintervals.

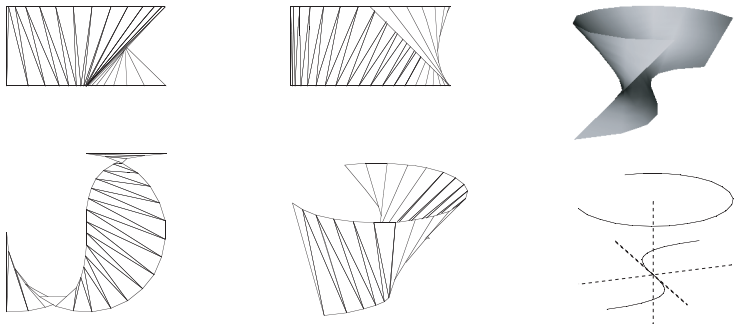

Figure 12.5 Linear interpolation of curves: surface connecting a portion of circle and a Bézier curve

A quite efficient and dimension-independent implementation of the ruled surface interpolating two arbitrary curves is given in Script 12.2.9, where `v1` and `v2` respectively denote the *sequences* of coordinate functions of the curves to be interpolated. The `RuledSurface` operator is given in Script 12.2.7.

Script 12.2.9 (Linear interpolation of two curves)

```
DEF v1 = Bezier:S1:<<1,1,0>,<-1,1,0>,<1,-1,0>,<-1,-1,0>>;
DEF v2 = < COS ~ (K:(3*PI/2) * S1), SIN ~ (K:(3*PI/2) * S1), K:1 >;

DEF domain = Intervals :1:10;
DEF out = MAP:(RuledSurface:< v1, v2 vectDiff v1 >):(domain * domain);

VRML:(out CREASE (PI/2)):'out.wrl';
```

12.2.3 Cylinders and cones

Cylinders and cones are subsets of the set of ruled surfaces. As such, they are characterized by vector equations that are special cases of equation 12.1. In particular, we denote as generalized cylinders and generalized cones the ruled surfaces whose lines either are all parallel or cross a single (proper) point, respectively.

Generalized cylinders

The generalized cylinders, simply called *cylinders* in the following sections, are ruled surfaces where the direction of the lines is given by a vector $\boldsymbol{\beta}$ with constant components. Therefore, the general vector equation of cylinders is:

$$\boldsymbol{S}(u,v) = \boldsymbol{\alpha}(u) + v\,\boldsymbol{\beta}.$$

It is easy to see that:

1. The special case of *straight circular cylinder* is obtained when $\boldsymbol{\alpha}(u)$ is a circle and $\boldsymbol{\beta}$ is normal to its plane.
2. $\boldsymbol{S}(u,v)$ is not regular where $\boldsymbol{\alpha}'(u) \times \boldsymbol{\beta} = \mathbf{0}$.
3. $\boldsymbol{\alpha}(\bar{u}) + v\boldsymbol{\beta}$ and $\boldsymbol{\alpha}(u) + \bar{v}\boldsymbol{\beta}$, where $\bar{u}, \bar{v} \in \mathbb{R}$ are fixed numbers, are the two families of *coordinate curves*, i.e. the curves with one constant parameter.

Example 12.2.6 (Cylindrical surface)
In Script 12.2.10, the `CylindricalSurface` generating function is mainly an alias for the `RuledSurface` function given in Script 12.2.7, with the important difference that the `beta` parameter is here a vector of reals, that is transformed into a sequence of constant coordinate functions by the expression `AA:K:beta`.

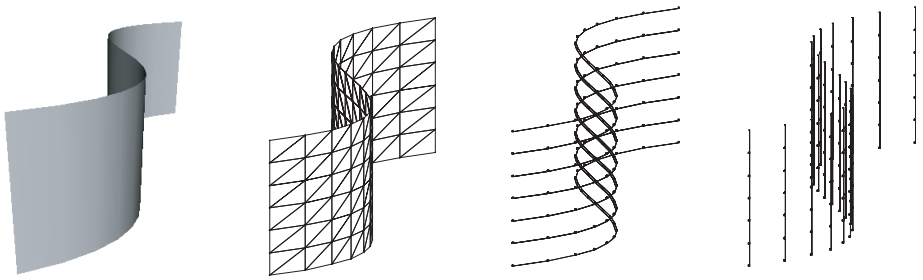

Figure 12.6 Polyhedral approximation of the cylindrical surface and some coordinate curves

Script 12.2.10 (Cylindrical surface)

```
DEF CylindricalSurface (alpha::IsSeqOf:IsFun; beta::IsSeqOf:IsReal) =
   RuledSurface:< alpha, AA:K:beta >;
```

The CylindricalSurface function given above clearly works in the hypotheses that:

1. the `alpha` section curve is three-dimensional;
2. the `beta` vector, parallel to the rules, is *non coplanar* with the section curve.

The example object generated by the Script 12.2.11 is displayed in Figure 12.6c. The geometric objects given in Figures 12.6a and 12.6b are produced by the other two definitions of the `domain` symbol.

At this point, the reader should write and test two different versions of a generating function for a circular and straight `cylinder` with arbitrary positive `radius` and `height`, by respectively using:

1. the `CylindricalSurface` given in Script 12.2.10, invoked with suitable values for actual parameters;

Script 12.2.11 (Cylindrical surface example)

```
DEF alpha = Bezier:S1:<<1,1,0>,<-1,1,0>,<1,-1,0>,<-1,-1,0>>;

DEF Udomain = Intervals:1:20;
DEF Vdomain = Intervals:1:6;

DEF domain = Udomain * Vdomain;
DEF domain = Udomain * @0:Vdomain;
DEF domain = @0:Udomain * Vdomain;

DEF out = MAP:(CylindricalSurface:< alpha, <0,0,1> >):domain;
VRML:out:'out.wrl';
```

2. a standard Cartesian product of the 2D `circle` with positive `radius`, times a suitable 1D segment of positive `lenght`.

Generalized cones

A *conical surface* is a ruled surface $\boldsymbol{S}(u, v)$ according with the general equation

$$\boldsymbol{S}(u, v) = \boldsymbol{\alpha} + v\boldsymbol{\beta}(u), \tag{12.2}$$

where the $\boldsymbol{\alpha}$ point, that crosses all the rays $v\boldsymbol{\beta}(u)$, is called the *apex* (or *vertex*) of the conical surface.

1. $\boldsymbol{S}(u, v)$ is said to be an *half-conical surface* if either $v \geq 0$ or $v \leq 0$, for each v.
2. A *circular cone* results if, for each u, the vectors $\boldsymbol{\beta}(u)$ give a constant angle θ with a fixed vector.
3. The surface $\boldsymbol{S}(u, v)$ is *regular* almost everywhere. It is non-regular for $v = 0$, i.e. on the apex, and where $\boldsymbol{\beta} \times \boldsymbol{\beta}' = \mathbf{0}$.

Implementation Once again, the `PLaSM` implementation of a *conical surface* is straightforward. It is given in Script 12.2.12, where the point `alpha` is the apex, and the curve `beta` describes the directions of the rays.

Script 12.2.12 (Conical surfaces)

```
DEF ConicalSurface (alpha::IsSeqOf:IsReal; beta::IsSeqOf:IsFun) =
   RuledSurface:< AA:K:alpha, beta vectDiff AA:K:alpha >;
```

Example 12.2.7 (Conical surface)
The conical surface patch generated by a cubic Bézier section on the plane $z = 0$ is given in Script 12.2.13 and displayed in Figure 12.7, where the $[0, 1]^2$ domain of the surface patch is decomposed into 20×6 convex cells (2D) and `MAP`ped into $20 \times 6 \times 2$ triangles (3D).

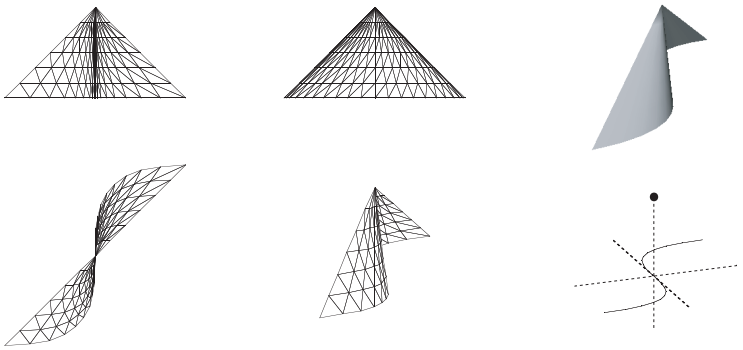

Figure 12.7 The conical surface generated by a Bézier curve in the $z = 0$ plane

Script 12.2.13 (Conical surface example)

```
DEF beta = Bezier:S1:<<1,1,0>,<-1,1,0>,<1,-1,0>,<-1,-1,0>>;
DEF domain = Intervals:1:20 * Intervals:1:6;

MAP:(ConicalSurface:< <0,0,1>, beta >):domain;
```

12.3 Tensor product surfaces

The parametric surfaces called *tensor product surfaces* are a bivariate generalization of parametric curves. As we know, polynomial curves can be seen as a linear combination of either points or vectors, called geometric handles, with a basis for polynomial functions of one variable, with appropriate degree. Analogously, polynomial surfaces are a linear combination of either points or vectors with a suitable basis of *bivariate* polynomial functions. Such a bivariate basis is obtained by *tensor product* of two univariate bases. The result of the linear combination of the bivariate basis with a set of geometric handles is a point-valued function of two real parameters, whose image set is the surface considered as a point locus. The properties of tensor product surfaces are a straightforward extension of properties of parametric curves. Rational surfaces in $\mathbb{E}^n$ are obtained from polynomial surfaces in $\mathbb{E}^{n+1}$.

Geometric form of a curve We know, from Chapter 11, that the geometric form of a polynomial curve of degree m can be written in matrix form as

$$\boldsymbol{c}(u) = \boldsymbol{U}_m \; \boldsymbol{B}_m^m \; \boldsymbol{G}^m = \boldsymbol{\mathcal{B}}_m(u) \; \boldsymbol{G}^m \tag{12.3}$$

where $\boldsymbol{U}_m$ is the row vector containing the power basis of degree m, $\boldsymbol{B}_m^m \in \mathbb{R}_{m+1}^{m+1}$ is the matrix of the geometric form, $\boldsymbol{G}^m$ is a one-index column tensor of geometric handles, and $\boldsymbol{\mathcal{B}}_m(u)$ is the polynomial basis, with degree m, of the geometric form.

Geometric form of a surface Let us suppose that the geometric handles $\boldsymbol{G}^m = (\boldsymbol{p}^0, \ldots, \boldsymbol{p}^m)^T$ are not constant points, but point-valued functions of the parameter

$v \in [0,1]$, i.e. curves, all with the same geometric form and with the same degree n:

$$\boldsymbol{G}^m = \boldsymbol{G}^m(v) = \begin{pmatrix} \boldsymbol{c}^0(v) \\ \cdots \\ \boldsymbol{c}^m(v) \end{pmatrix},$$

so that we can rewrite equation (12.3) as

$$\boldsymbol{S}(u,v) = \boldsymbol{\mathcal{B}}_m(u)\ \boldsymbol{G}^m(v) = \boldsymbol{\mathcal{B}}_m(u) \begin{pmatrix} \boldsymbol{c}^0(v) \\ \cdots \\ \boldsymbol{c}^m(v) \end{pmatrix} \tag{12.4}$$

where each $\boldsymbol{c}^i(v)$ can be written as

$$\boldsymbol{c}^i(v) = \boldsymbol{\mathcal{B}}_n(u)\ \boldsymbol{G}_i^n, \qquad 0 \leq i \leq m.$$

so that we have:

$$\begin{pmatrix} \boldsymbol{c}^0(v) & \cdots & \boldsymbol{c}^m(v) \end{pmatrix} = \boldsymbol{\mathcal{B}}_n(u) \begin{pmatrix} \boldsymbol{G}_0^n & \cdots & \boldsymbol{G}_m^n \end{pmatrix}.$$

In order to get a column tensor of geometric handles, we can transpose the above expression:

$$\begin{pmatrix} \boldsymbol{c}^0(v) \\ \cdots \\ \boldsymbol{c}^m(v) \end{pmatrix} = (\boldsymbol{\mathcal{B}}_n(v) \begin{pmatrix} \boldsymbol{G}_0^n & \cdots & \boldsymbol{G}_m^n \end{pmatrix})^T = \begin{pmatrix} \boldsymbol{G}_n^0 \\ \cdots \\ \boldsymbol{G}_n^m \end{pmatrix} \boldsymbol{\mathcal{B}}^n(v)$$

And then it is possible to substitute in equation (12.4), getting finally

$$\boldsymbol{S}(u,v) = \boldsymbol{\mathcal{B}}_m(u) \begin{pmatrix} \boldsymbol{G}_n^0 \\ \cdots \\ \boldsymbol{G}_n^m \end{pmatrix} \boldsymbol{\mathcal{B}}^n(v) = \boldsymbol{\mathcal{B}}_m(u)\ \boldsymbol{\mathcal{G}}_n^m\ \boldsymbol{\mathcal{B}}^n(v) \tag{12.5}$$

where $\boldsymbol{\mathcal{B}}_m(u)$ and $\boldsymbol{\mathcal{B}}^n(v)$ are two univariate basis of polynomial (rational) functions of degrees m and n, respectively, and $\boldsymbol{\mathcal{G}}_n^m = (\boldsymbol{G}_j^i)_{j=0,\ldots,n}^{i=0,\ldots,m}$ is a two-index tensor of control handles.

Tensor product form Making explicit the dependence of the surface on the degrees n and m of the polynomial (rational) bases in the u and v parameters, we can write in components:

$$\boldsymbol{S}(m,n)(u,v) = \boldsymbol{\mathcal{B}}_m(u)\ \boldsymbol{\mathcal{G}}_n^m\ \boldsymbol{\mathcal{B}}^n(v) = \sum_{i,j=0}^{m,n} B_i(u)\ B_j(v)\ \boldsymbol{p}_{ij} = \sum_{i,j=0}^{m,n} B_{ij}(u,v)\ \boldsymbol{p}_{ij}$$

The last expression is equal to the inner product of the bivariate basis times the control handles of the surface. In turn, the bivariate basis may be computed as tensor product of the univariate bases:

$$\boldsymbol{S}(m,n)(u,v) = \boldsymbol{\mathcal{B}}_m^n(u,v) \cdot \boldsymbol{\mathcal{G}}_n^m = \boldsymbol{\mathcal{B}}_m(u) \otimes \boldsymbol{\mathcal{B}}^n(v) \cdot \boldsymbol{\mathcal{G}}_n^m.$$

Note The equivalent matrix form

$$\boldsymbol{S}(m,n)(u,v) = \boldsymbol{U}_m \; \boldsymbol{M}_m^m \; \boldsymbol{\mathcal{G}}_n^m \; \boldsymbol{M}_n^n \; \boldsymbol{V}^n$$

is often improperly called the "tensor product form". That denotation would be more appropriately used for the form given above.

Implementation The *tensor product form* of surfaces will be primarily used, in the remainder of this chapter, to support the `PLaSM` implementation of polynomial (rational) surfaces. For this purpose, we start by defining some basic operators on function tensors.

In particular, a toolbox of basic tensor operations is given in Script 12.3.1. The `ConstFunTensor` operator produces a tensor of constant functions starting from a tensor of numbers; the recursive `FlatTensor` may be used to "flatten" a tensor with any number of indices by producing a corresponding one index tensor; the `InnerProd` and `TensorProd` are used to compute the inner product and the tensor product of conforming tensors of functions, respectively.

Script 12.3.1 (Toolbox of tensor operations)

```
DEF ConstFunTensor = IF:<IsSeq, AA:ConstFunTensor, K>;
DEF TensorInnerProd = InnerProd ~ AA:FlatTensor ;
DEF FlatTensor = IF:<IsSeqOf:IsFun, ID,CAT ~ AA:FlatTensor> ;
DEF InnerProd = + ~ AA:* ~ TRANS;
DEF TensorProd = AA:DistlTerm ~ DISTR;
DEF DistlTerm = IF:<IsFun ~ S2, FunProd, AA:DistlTerm ~ DISTL>;
DEF FunProd (f,g::IsFun) = f ~ [S1] * g ~ TAIL;
```

In Script 12.3.2 a `TensorProdSurface` dimension-independent generator is given, with input parameters the sequences `Basis1` and `Basis2` of univariate polynomial bases, and the conforming array of `ControlPoints`.

Algorithm The algorithm implemented in Script 12.3.2 works as follows:

1. By suitably transposing the `controlPoints` parameter, a sequence $(\boldsymbol{X_i})$ of coordinate tensors, with $\boldsymbol{X_i} \in \mathbb{R}_n^m$ and $1 \leq i \leq d$, is returned into `CoordTensors`. Notice that d is the dimension of the target space of the $\boldsymbol{S}(m,n) : \mathbb{R}^2 \to \mathbb{E}^d$ mapping.
2. Each such two-index array of numbers $\boldsymbol{X_i}$ is transformed into a two-index array of constant functions by the expression

   ```
   AA:ConstFunTensor:CoordTensors
   ```

3. The bivariate basis $\boldsymbol{\mathcal{B}}_m^n(u,v)$ is returned into `BivariateBasis` by applying the `TensorProd` operator on the univariate `uBasis` and `vBasis`.
4. The sequence of the d coordinate functions of the $\boldsymbol{S}(m,n) : \mathbb{R}^2 \to \mathbb{E}^d$ mapping is finally produced, by distributing the `BivariateBasis` over the sequence $(\boldsymbol{X_i})$, $1 \leq i \leq d$, and by applying to each pair the `TensorScalarProd` operator.

Script 12.3.2 (Tensor product surface)

```
DEF IsMatOf (IsType::IsBool) =
   AND ~ [AND ~ CAT ~ (AA ~ AA):IsType, EQ ~ AA:LEN, IsSeq];

DEF TensorProdSurface (uBasis,vBasis::IsSeq)(ControlPoints::IsMatOf:IsPoint) =
   (AA:TensorInnerProd ~ DISTL):
      < BivariateBasis, AA:ConstFunTensor:CoordTensors >
WHERE
   CoordTensors = (TRANS~AA:TRANS):controlPoints,
   BivariateBasis = TensorProd:< uBasis, vBasis >
END;
```

12.3.1 Bilinear and biquadratic surfaces

In this section we discuss some useful low-degree geometric forms of tensor product surfaces. A bilinear surface can be seen as both the Lagrange and the Bézier forms, interpolating four arbitrary control points, arranged as a 2×2 array. Biquadratic Lagrange forms interpolate a 3×3 array of control points.

Bilinear surfaces

We know that the parametric equation of the line segment though two points $\boldsymbol{p}_1, \boldsymbol{p}_2 \in \mathbb{E}^d$ can be written as

$$\boldsymbol{\alpha}(u) = \boldsymbol{p}_1 + (\boldsymbol{p}_2 - \boldsymbol{p}_1)\, u \qquad u \in [0,1],$$

or, in matrix form, as

$$\boldsymbol{\alpha}(u) = \begin{pmatrix} u & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \end{pmatrix}.$$

If $\boldsymbol{p}_1, \boldsymbol{p}_2$ are linear functions of a v parameter, or, in other words, are themselves *curves*, then the equation

$$\boldsymbol{S}(u,v) = \begin{pmatrix} u & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{p}_1(v) \\ \boldsymbol{p}_2(v) \end{pmatrix} \tag{12.6}$$

will describe a *bilinear surface*. In fact, we have

$$\boldsymbol{p}_1(v) = \begin{pmatrix} v & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{p}_{11} \\ \boldsymbol{p}_{12} \end{pmatrix}$$

$$\boldsymbol{p}_2(v) = \begin{pmatrix} v & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{p}_{21} \\ \boldsymbol{p}_{22} \end{pmatrix}$$

so that

$$\begin{pmatrix} \boldsymbol{p}_1(v) & \boldsymbol{p}_2(v) \end{pmatrix} = \begin{pmatrix} v & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{p}_{11} & \boldsymbol{p}_{21} \\ \boldsymbol{p}_{12} & \boldsymbol{p}_{22} \end{pmatrix}.$$

By transposing both sides of the previous equation we have

$$\begin{pmatrix} \boldsymbol{p}_1(v) \\ \boldsymbol{p}_2(v) \end{pmatrix} = \begin{pmatrix} \boldsymbol{p}_{11} & \boldsymbol{p}_{12} \\ \boldsymbol{p}_{21} & \boldsymbol{p}_{22} \end{pmatrix} \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix}^T \begin{pmatrix} v \\ 1 \end{pmatrix}$$

whose right-hand side can be substituted for the left-side in equation (12.6). So, by setting

$$\boldsymbol{U}_1 = \begin{pmatrix} u & 1 \end{pmatrix}$$

$$\boldsymbol{M}_1 = \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix},$$

we can finally write the matrix form

$$\boldsymbol{S}(u,v) = \boldsymbol{U}_1 \; \boldsymbol{M}_1 \; \boldsymbol{G}_1^1 \; \boldsymbol{M}^1 \; \boldsymbol{V}^1 = \boldsymbol{B}_1(u) \otimes \boldsymbol{B}^1(v) \cdot \boldsymbol{G}_1^1 \tag{12.7}$$

of the bilinear surface, where

$$\boldsymbol{G}_1^1 = \begin{pmatrix} \boldsymbol{p}_{11} & \boldsymbol{p}_{12} \\ \boldsymbol{p}_{21} & \boldsymbol{p}_{22} \end{pmatrix}.$$

This kind of surface is called *bilinear* because it is linear both in the u and in the v parameter. Due to its construction, the surface patch $\boldsymbol{S}[0,1]^2 \subset \mathbb{E}^d$ clearly interpolates the four control points. Notice that such points belong to $\mathbb{E}^d$, with arbitrary positive integer d. Usually it is either $d = 3$ or $d = 2$.

Implementation A *dimension-independent* bilinear patch defined by four extreme points can be generated by the `BilinearSurface` operator given in Script 12.3.3.

The reader may easily check from equation (12.7) that $\boldsymbol{B}_1(u) = (1-u, u)$ and that $\boldsymbol{B}^1(v) = (1-v, v)^T$. Such a basis is both Lagrange and Bernstein/Bézier, so that our implementation follows. The `BernsteinBasis` generator is given in Script 12.5.2.

Script 12.3.3 (Bilinear surface)

```
DEF BilinearSurface (ControlPoints::IsMatOf:IsPoint) =
   TensorProdSurface:< BernsteinBasis:S1:1, BernsteinBasis:S1:1 >:ControlPoints
```

Example 12.3.1 (Bilinear surface patch)
The bilinear patch produced by four 3D points given in Script 12.3.4 is shown in Figure 12.8. The `Intervals` constructor of 1D polyhedral complexes is given in Script 12.2.2.

Biquadratic Lagrange form

In this case the constraint is to require the surface passage for 9 arbitrary points, arranged as a 3×3 array.

As always, the tensor product form of the parametric vector equation is easy to obtain. Let us remember[2] that a quadratic polynomial curve through 3 points has the

---

[2] See Section 11.2.2.

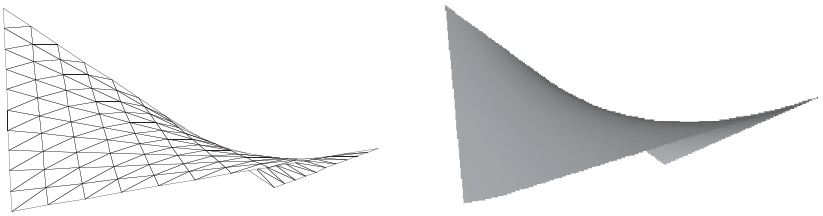

Figure 12.8 Patch of bilinear surface defined by four extreme points

Script 12.3.4 (Bilinear surface example)

```
DEF Mapping = BilinearSurface:<<<0,0,0>,<2,-4,2>>,<<0,3,1>,<4,0,0>>>;

MAP:Mapping:(Intervals:1:10 * Intervals:1:10)
```

equation:

$$\boldsymbol{\alpha}(u) = \begin{pmatrix} u^2 & u & 1 \end{pmatrix} \begin{pmatrix} 2 & -4 & 2 \\ -3 & 4 & -1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \end{pmatrix} = \boldsymbol{U}_2 \; \boldsymbol{M}_2 \begin{pmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \boldsymbol{p}_3 \end{pmatrix},$$

so that

$$\boldsymbol{S}(u,v) = \boldsymbol{U}_2 \; \boldsymbol{M}_2 \begin{pmatrix} \boldsymbol{p}_{11} & \boldsymbol{p}_{12} & \boldsymbol{p}_{13} \\ \boldsymbol{p}_{21} & \boldsymbol{p}_{22} & \boldsymbol{p}_{23} \\ \boldsymbol{p}_{31} & \boldsymbol{p}_{32} & \boldsymbol{p}_{33} \end{pmatrix} \boldsymbol{M}^2 \; \boldsymbol{V}^2 = \boldsymbol{B}_2(u) \otimes \boldsymbol{B}^2(v) \cdot \boldsymbol{G}_2^2.$$

Therefore, the Lagrange's basis of degree 2 is

$$\boldsymbol{B}_2(u) = U_2 \; \boldsymbol{M}_2 = \begin{pmatrix} u^2 & u & 1 \end{pmatrix} \begin{pmatrix} 2 & -4 & 2 \\ -3 & 4 & -1 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} b_2(u) & b_1(u) & b_0(u) \end{pmatrix},$$

with

$$\begin{aligned} b_2(u) &= 2u^2 - 3u + 1, \\ b_1(u) &= -4u^2 + 4u, \\ b_0(u) &= 2u^2 - u. \end{aligned}$$

Implementation A `BiquadraticSurface` generator function is given in Script 12.3.5 by using the the tensor product operations encoded within the `TensorProdSurface` operator defined in Script 12.3.2.

Example 12.3.2 (Biquadratic patch)
The generation of a biquadratic surface patch using the Hermite's interpolating geometric form is exemplified in Script 12.3.6. The generated patch is shown in Figure 12.9. Once more, the `Mapping` sequence of coordinate functions is `MAP`ped over a suitable partitioning of the $[0,1]^2$ domain, that is decomposed into 10×10 squared cells.

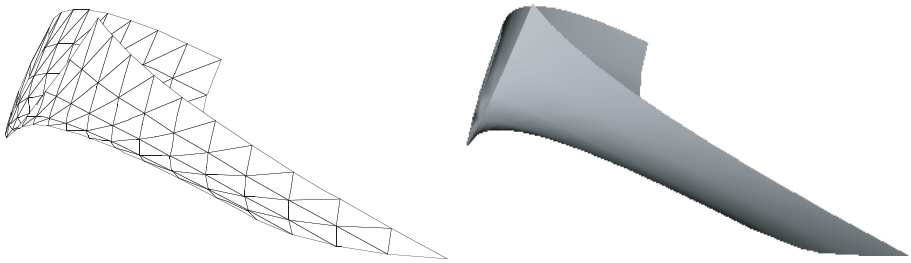

Figure 12.9 Biquadratic surface patch interpolating 3×3 assigned points

Script 12.3.5 (Biquadratic surfaces)

```
DEF BiquadraticSurface (ControlPoints::IsMatOf:IsPoint) =
   TensorProdSurface:< Basis, Basis >:ControlPoints
WHERE
   Basis = < u2, u1, u0 >,
   u2 = (k:2 * uu) - (K:3 * u) + K:1,
   u1 = (k:4 * u) - (k:4 * uu),
   u0 = (k:2 * uu) - u, uu = u * u, u = s1
END;
```

12.3.2 Bicubic surfaces

As already discussed in the previous chapter, cubic curves and bicubic surfaces are largely used by geometric modeling systems, since they provide a sufficient flexibility to define complex shapes by suitably connecting segments of curves or surfaces with C^1 continuity.

Algebraic form

The *algebraic form* of a bicubic polynomial surface is defined as

$$\boldsymbol{S}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} \boldsymbol{a}_{ij} u^i v^j, \qquad i,j \in [0,1]$$

where the 16 vector coefficients $\boldsymbol{a}_{ij}$ are called *algebraic coefficients* of the surface, that results to have $16 \times 3 = 48$ *degrees of freedom* in $\mathbb{E}^3$. In matrix notation, the algebraic form of a bicubic surface is

$$\boldsymbol{S}(u,v) = \boldsymbol{U}_3 \; \boldsymbol{A} \; \boldsymbol{V}_3^T,$$

Script 12.3.6 (example of biquadratic patch)

```
DEF Mapping = BiquadraticSurface:<
   <<0,0,0>, <2,0,1>,<3,1,1>>,
   <<1,3,-1>,<3,2,0>,<4,2,0>>,
   <<0,9,0>, <2,5,1>,<3,3,2>> >;

MAP:Mapping:(Intervals:1:10 * Intervals:1:10)
```

where

$$\begin{aligned} \boldsymbol{U}_3 &= \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \\ \boldsymbol{V}_3 &= \begin{pmatrix} v^3 & v^2 & v & 1 \end{pmatrix}. \end{aligned}$$

A more explicit vector form is

$$\begin{aligned} \boldsymbol{S}(u,v) = \; & \boldsymbol{a}_{33}\, u^3v^3 + \boldsymbol{a}_{32}\, u^3v^2 + \boldsymbol{a}_{31}\, u^3v + \boldsymbol{a}_{30}\, u^3 \\ & \boldsymbol{a}_{23}\, u^2v^3 + \boldsymbol{a}_{22}\, u^2v^2 + \boldsymbol{a}_{21}\, u^2v + \boldsymbol{a}_{20}\, u^2 \\ & \boldsymbol{a}_{13}\, uv^3 + \boldsymbol{a}_{12}\, uv^2 + \boldsymbol{a}_{11}\, uv + \boldsymbol{a}_{10}\, u \\ & \boldsymbol{a}_{03}\, v^3 + \boldsymbol{a}_{02}\, v^2 + \boldsymbol{a}_{01}\, v + \boldsymbol{a}_{00}. \end{aligned}$$

As we already know from the study of parametric curves, the so-called *geometric forms* of the surface, defined by suitable polynomial bases, different from the standard power basis, and where the degrees of freedom have some explicit geometric meaning, are much more useful to the designer. Similarly to what we have seen for the curves, we can discover such useful bases for the bicubic surfaces by setting $4 \times 4 = 16$ vector constraints, and by solving their linear system.

Lagrange's form

As for curves, the passage of the surface is imposed for a set of 4×4 points $\boldsymbol{p}_j^i \in \mathbb{E}^d$, arranged within a

$$\boldsymbol{G}_{LL} = (\boldsymbol{p}_j^i)_{j=0,\ldots,3}^{i=0,\ldots,3}$$

tensor, so that the surface mapping $\boldsymbol{S} : [0,1]^2 \to \mathbb{E}^d$ is generated as

$$\boldsymbol{S}(u,v) = \boldsymbol{U}_3\ \boldsymbol{M}_L\ \boldsymbol{G}_{LL}\ \boldsymbol{M}_L^T\ \boldsymbol{V}_3^T.$$

where the geometric Lagrange matrix $\boldsymbol{M}_L$ is the same already computed in Section 11.2.3 for the cubic polynomial curves. The implementation of the bicubic Lagrange's mapping is left to the reader.

Bicubic Hermite geometric form

Let us consider that

$$\boldsymbol{S}(u,v) = \boldsymbol{U}_3\ \boldsymbol{M}_h \begin{pmatrix} \boldsymbol{p}_1(v) \\ \boldsymbol{p}_2(v) \\ \boldsymbol{s}_1(v) \\ \boldsymbol{s}_2(v) \end{pmatrix}$$

where $\boldsymbol{p}_1(v), \boldsymbol{p}_2(v), \boldsymbol{s}_1(v)$ and $\boldsymbol{s}_2(v)$ are cubic Hermite curves. So, we may write

$$\begin{pmatrix} \boldsymbol{p}_1(v) & \boldsymbol{p}_2(v) & \boldsymbol{s}_1(v) & \boldsymbol{s}_2(v) \end{pmatrix} =$$
$$\begin{pmatrix} v^3 & v^2 & v & 1 \end{pmatrix} \boldsymbol{M}_h \begin{pmatrix} \boldsymbol{q}_{11} & \boldsymbol{q}_{21} & \boldsymbol{q}_{31} & \boldsymbol{q}_{41} \\ \boldsymbol{q}_{12} & \boldsymbol{q}_{22} & \boldsymbol{q}_{32} & \boldsymbol{q}_{42} \\ \boldsymbol{q}_{13} & \boldsymbol{q}_{23} & \boldsymbol{q}_{33} & \boldsymbol{q}_{43} \\ \boldsymbol{q}_{14} & \boldsymbol{q}_{24} & \boldsymbol{q}_{34} & \boldsymbol{q}_{44} \end{pmatrix}.$$

The resulting matrix and tensor forms for the bicubic Hermite surface mapping are:

$$\boldsymbol{S}(u,v) = \boldsymbol{U}_3 \ \boldsymbol{M}_h \ \boldsymbol{G}_{hh} \ \boldsymbol{M}_h^T \ \boldsymbol{V}_3^T,$$

$$\boldsymbol{S}(u,v) = \boldsymbol{H}_3(u) \otimes \boldsymbol{H}_3(v) \cdot \boldsymbol{G}_{hh},$$

where

$$\boldsymbol{H}_3(u) = \begin{pmatrix} h_3(u) & h_2(u) & h_1(u) & h_0(u) \end{pmatrix}$$

is the *cubic Hermite basis*, i.e.:

$$\begin{aligned} h_3(u) &= 2u^3 - 3u^2 + 1, \\ h_2(u) &= -2u^3 + 3u^2, \\ h_1(u) &= u^3 - 2u^2 + u, \\ h_0(u) &= u^3 - u^2. \end{aligned}$$

Implementation Our standard format for the implementation of bicubic Hermite surfaces as tensor product surfaces is used in Script 12.3.7, where `HermiteBasis` just contains the `PLaSM` implementation with function algebra of the basis polynomials given above.

Script 12.3.7 (Bicubic Hermite surfaces)

```
DEF HermiteSurface (ControlPoints::IsMatOf:IsPoint) =
   TensorProdSurface:< HermiteBasis, HermiteBasis >:ControlPoints
WHERE
   HermiteBasis = < h3, h2, h1, h0 >,
   h0 = (uuu) - (uu),
   h1 = (uuu) - (k:2 * uu) + u,
   h2 = (k:3 * uu) - (k:2 * uuu),
   h3 = (k:2 * uuu) - (k:3 * uu) + (k:1),
   uuu = uu * u, uu = u * u,
   u = s1
END;
```

Example 12.3.3 (Bicubic Hermite patch)
A bicubic Hermite patch is generated by Script 12.3.8 and displayed in Figure 12.10, both as polyhedral approximation with $10\times10\times2$ triangles and with smooth rendering using a Gouraud shader. Notice that the only input is the 4×4 tensor of geometric handles, organized as a 4×4 matrix of 3D points, represented in `PLaSM` as sequences of coordinates.

Coordinate functions It may be interesting to generate the graphs of the three coordinate functions of the mapping $\boldsymbol{S} : [0,1]^2 \to \mathbb{E}^3$ produced in Example 12.3.3. The triplet of coordinate function graphs is generated by the out object of Script 12.3.9, and is shown in Figure 12.11.

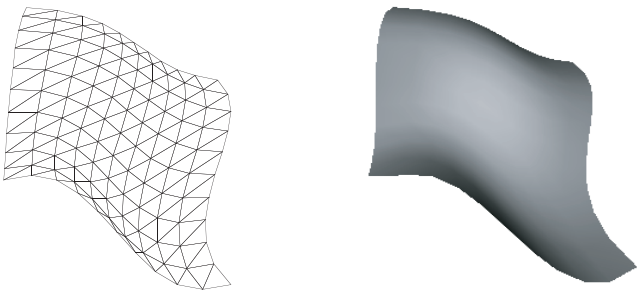

Figure 12.10 Bicubic Hermite's patch: (a) faceted approximation (b) smooth rendering

Script 12.3.8

```
DEF Mapping = HermiteSurface: <
   <<0,0,0>, <2,0,1>, <3,1,1>, <4,1,1>>,
   <<1,3,-1>,<3,2,0>, <4,2,0>, <4,2,0>>,
   <<0,4,0>, <2,4,1>, <3,3,2>, <5,3,2>>,
   <<0,6,0>, <2,5,1>, <3,4,1>, <4,4,0>> >;

MAP:Mapping:(Intervals:1:10 * Intervals:1:10);
```

Meaning of Hermite's geometry tensor It is interesting to discuss the semantics of a decomposition into four submatrices of the bicubic Hermite's geometry tensor $\boldsymbol{\mathcal{G}}_3^3$. Let us first remember that

$$\boldsymbol{S}(u,v) = \boldsymbol{U}_3 \boldsymbol{M}_h \ \boldsymbol{\mathcal{G}}_{hh} \ \boldsymbol{M}_h^T \boldsymbol{V}^3$$

where

$$\boldsymbol{\mathcal{G}}_{hh} = \begin{pmatrix} \boldsymbol{q}_1^1 & \boldsymbol{q}_2^1 & \boldsymbol{q}_3^1 & \boldsymbol{q}_4^1 \\ \boldsymbol{q}_1^2 & \boldsymbol{q}_2^2 & \boldsymbol{q}_3^2 & \boldsymbol{q}_4^2 \\ \boldsymbol{q}_1^3 & \boldsymbol{q}_2^3 & \boldsymbol{q}_3^3 & \boldsymbol{q}_4^3 \\ \boldsymbol{q}_1^4 & \boldsymbol{q}_2^4 & \boldsymbol{q}_3^4 & \boldsymbol{q}_4^4 \end{pmatrix}$$

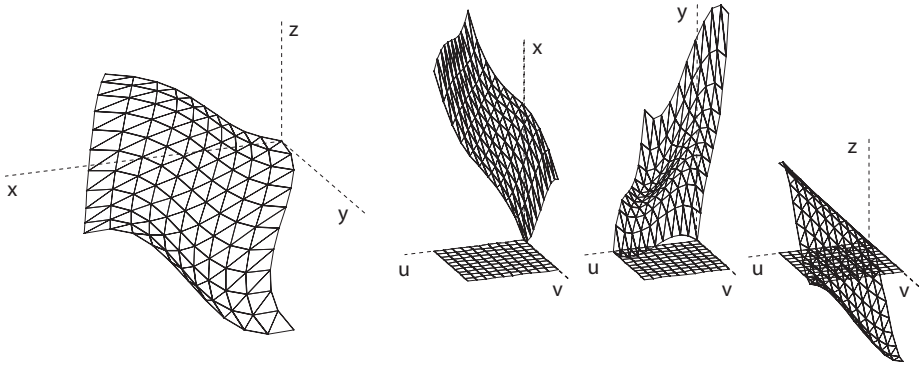

Figure 12.11 (a) Bicubic Hermite's patch in $\mathbb{E}^3$; (b) graphs of coordinate functions $[0,1]^2 \to \mathbb{R}$ from (u,v) space

Script 12.3.9

```
DEF u = s1; DEF v = s2;
DEF x = s1; DEF y = s2; DEF z = s3;

DEF out = STRUCT:<
   (STRUCT ~ [MAP:[u, v, x:Mapping], EMBED:1]):domain, T:<1,2>:<-1.5,1>,
   (STRUCT ~ [MAP:[u, v, y:Mapping], EMBED:1]):domain, T:<1,2>:<-1.5,1>,
   (STRUCT ~ [MAP:[u, v, z:Mapping], EMBED:1]):domain
>;
```

If we look into the development process of the previous equation, it is easy to see that $\mathcal{G}_{hh}$ can be decomposed into four submatrices corresponding to the surface $\boldsymbol{S}$, its first partial derivatives $\boldsymbol{S}^u$, $\boldsymbol{S}^v$ and its mixed second partial derivative $\boldsymbol{S}^{uv}$, respectively evaluated in the four corners of $[0,1]^2$ domain:

$$\mathcal{G}_{hh} = \begin{pmatrix} \boldsymbol{S}(0,0) & \boldsymbol{S}(0,1) & \boldsymbol{S}^u(0,0) & \boldsymbol{S}^u(0,1) \\ \boldsymbol{S}(1,0) & \boldsymbol{S}(1,1) & \boldsymbol{S}^u(1,0) & \boldsymbol{S}^u(1,1) \\ \\ \boldsymbol{S}^v(0,0) & \boldsymbol{S}^v(0,1) & \boldsymbol{S}^{uv}(0,0) & \boldsymbol{S}^{uv}(0,1) \\ \boldsymbol{S}^v(1,0) & \boldsymbol{S}^v(1,1) & \boldsymbol{S}^{uv}(1,0) & \boldsymbol{S}^{uv}(1,1) \end{pmatrix}$$

Example 12.3.4 (Effect of tangent vectors)
By reversing the direction of the four vectors $\boldsymbol{S}^u(0,0), \boldsymbol{S}^u(0,1), \boldsymbol{S}^u(1,0)$ and $\boldsymbol{S}^u(1,1)$, as done in Script 12.3.10, we get the effect shown in Figure 12.12a.

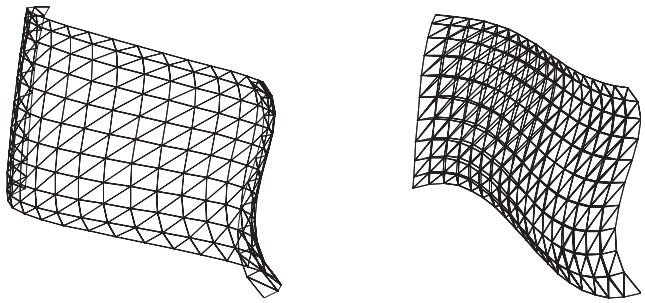

Figure 12.12 (a) Hermite's patch with reversed $\boldsymbol{S}^u$ directions (b) Display of the differences between Hermite's and Ferguson's patches

Script 12.3.10 (Reversing four partial derivatives)

```
DEF ReversedExample = HermiteSurface:<
   <<0,0,0>, <2,0,1>, AA:-:<3,1,1>, AA:-:<4,1,1>>,
   <<1,3,-1>,<3,2,0>, AA:-:<4,2,0>, AA:-:<4,2,0>>,
   <<0,4,0>, <2,4,1>, <3,3,2>, <5,3,2>>,
   <<0,6,0>, <2,5,1>, <3,4,1>, <4,4,0>> >

MAP:ReversedExample:(Intervals:1:10 * Intervals:1:10)
```

Example 12.3.5 (Ferguson's surfaces)
The geometric meaning of the second mixed partial derivatives $\boldsymbol{S}^{uv}$, also called *twist vectors*, is not easy to understand. The easiest way to assign such derivatives at the patch corner points is by giving them a null value. In such a choice, the Hermite's surfaces are called Ferguson's surfaces, or else *F-patches*. An example of Ferguson's surface is produced by Script 12.3.11. The difference between a Ferguson's patch and an Hermite's patch with the same control tensor is displayed in Figure 12.12b.

Script 12.3.11

```
DEF FergusonExample = HermiteSurface:<
   <<0,0,0>, <2,0,1>, <3,1,1>, <4,1,1>>,
   <<1,3,-1>,<3,2,0>, <4,2,0>, <4,2,0>>,
   <<0,4,0>, <2,4,1>, AA:(k:0):<3,3,2>, AA:(k:0):<5,3,2>>,
   <<0,6,0>, <2,5,1>, AA:(k:0):<3,4,1>, AA:(k:0):<4,4,0>> >

MAP:FergusonExample:(Intervals:1:10 * Intervals:1:10)
```

Bicubic Bézier geometric form

A derivation process, similar to the one leading to the bicubic Hermite's form, can be done for the bicubic Bézier surface, defined as the linear combination of the Bézier polynomial basis of degree three with four Bézier curves of the same degree:

$$\boldsymbol{S}(u,v) = \boldsymbol{B}_3(u) \begin{pmatrix} \boldsymbol{q}_1(v) \\ \boldsymbol{q}_2(v) \\ \boldsymbol{q}_3(v) \\ \boldsymbol{q}_4(v) \end{pmatrix},$$

where

$$\boldsymbol{B}_3(u) = \begin{pmatrix} (1-u)^3 & 3u(1-u)^2 & 3u^2(1-u) & u^3 \end{pmatrix}$$

$$\boldsymbol{q}_i(v) = \boldsymbol{B}_3(v) \begin{pmatrix} \boldsymbol{q}_{1i} \\ \boldsymbol{q}_{2i} \\ \boldsymbol{q}_{3i} \\ \boldsymbol{q}_{4i} \end{pmatrix}, \qquad 1 \le i \le 4.$$

Hence it is possible to write

$$\boldsymbol{S}(u,v) = \boldsymbol{B}_3(u)\ \boldsymbol{G}_{bb}\ \boldsymbol{B}_3^T(v),$$

with

$$\boldsymbol{G}_{bb} = \begin{pmatrix} \boldsymbol{q}_{11} & \boldsymbol{q}_{12} & \boldsymbol{q}_{13} & \boldsymbol{q}_{14} \\ \boldsymbol{q}_{21} & \boldsymbol{q}_{22} & \boldsymbol{q}_{23} & \boldsymbol{q}_{24} \\ \boldsymbol{q}_{31} & \boldsymbol{q}_{32} & \boldsymbol{q}_{33} & \boldsymbol{q}_{34} \\ \boldsymbol{q}_{41} & \boldsymbol{q}_{42} & \boldsymbol{q}_{43} & \boldsymbol{q}_{44} \end{pmatrix}.$$

Clearly, other possible forms for the Bézier bilinear surface are:

$$\begin{aligned} \boldsymbol{S}(u,v) &= \boldsymbol{U}_3\ \boldsymbol{M}_b\ \boldsymbol{G}_{bb}\ \boldsymbol{M}_b^T\ \boldsymbol{V}_3^T, \\ &= \sum_{i=1}^{4}\sum_{j=1}^{4} B_i(u)B_j(v)\boldsymbol{q}_{ij}, \\ &= \boldsymbol{B}_3(u) \otimes \boldsymbol{B}_3(v) \cdot \boldsymbol{G}_{bb}. \end{aligned}$$

Implementation The implementation of a dimension-independent `BezierSurface` generator of surface maps of arbitrary degrees (m, n) is given in Script 12.3.12. It is simply defined by applying the `TensorProdSurface` operator to a pair of Bernstein/Bézier bases of degrees `m` and `n`, where `m` and `n` are related to the numbers of rows and columns of the `ControlPoints` matrix, respectively. The `BernsteinBasis` generator is given in Script 11.2.3.

The reader should notice that the `BezierSurface` operator is able to generate a Bézier surface of *arbitrary degrees*, embedded in a $\mathbb{E}^d$ space, with an *arbitrary dimension* d. For example, a bilinear surface though 4 points could be generated as a `BezierSurface` with a 2×2 `ControlPoints` array.

Script 12.3.12 (Bézier surfaces)

```
DEF BezierSurface (ControlPoints::IsMatOf:IsPoint) =
   TensorProdSurface:< BernsteinBasis:m, BernsteinBasis:n >:ControlPoints
WHERE
   m = LEN:ControlPoints - 1,
   n = (LEN ~ S1):ControlPoints - 1
END;
```

Example 12.3.6 (Bicubic Bézier surface generation)
Four quadruples of four control points needed to generate a bicubic surface, and called $\texttt{points}_i, 1 \leq i \leq 4$, are given in Script 12.3.14. The surface generated by last expression is shown in Figure 12.13b. The BezierSurface generator is given in Script 12.3.12.

Script 12.3.13 (Bicubic Bézier example)

```
DEF points1 = <<0,0,0>,<0,3,4>,<0,6,3>,<0,10,0>>;
DEF points2 = <<3,0,2>,<2,2.5,5>,<3,6,5>,<4,8,2>>;
DEF points3 = <<6,0,2>,<8,3,5>,<7,6,4.5>,<6,10,2.5>>;
DEF points4 = <<10,0,0>,<11,3,4>,<11,6,3>,<10,9,0>>;

DEF pointArray = < points1, points2, points3, points4 >;

MAP:(BezierSurface:pointArray):(Intervals:1:10 * Intervals:1:10);
```

Example 12.3.7 (Same surface from Bézier curves)
We develop here an instructive variation of Example 12.3.6, where the same bicubic surface is generated starting from the 4 curves produced by the four rows of the 4×4

`pointArray` tensor given in Script12.3.13. The geometric construction we refer to is shown in Figure 12.13. We like to note that:

1. the `biCubicBezier` surface image interpolates only the first and last Bézier curves in `primaryGrid`;
2. both the second and third curve in `primaryGrid` are only approximated by the surface;
3. the `secondaryGrid` is constituted by v-curves (and their control polygons) corresponding to 0, 0.5 and 1 values of the u parameter;
4. all the curves in `secondaryGrid` are interpolated by the surface, whereas their generating polygons have control points belonging to the curves in `primaryGrid`.

Script 12.3.14 (Grid of control polygons)

```
DEF curves = (CONS ~ AA:Bezier):pointArray;

DEF BezierAndPolygon (points::IsSeq) = STRUCT:<
   MAP:(Bezier:points):(Intervals:1:10),
   polyline:points
>;

DEF primaryGrid = (STRUCT ~ AA:BezierAndPolygon):pointArray;
DEF secondaryGrid = (STRUCT ~ AA:(BezierAndPolygon ~ curves ~ [ID])):
   < 0 , 0.5, 1 >;

DEF surfaceMap = BezierSurface:pointArray;
DEF biCubicBezier = MAP:surfaceMap:(Intervals:1:10 * Intervals:1:10)

STRUCT:< primaryGrid, secondaryGrid, biCubicBezier >;
```

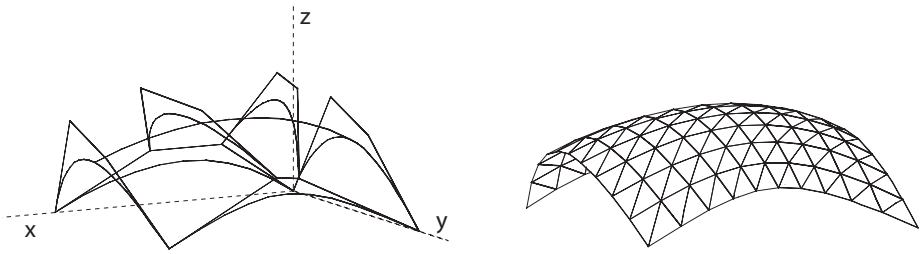

Figure 12.13 The geometric construction generated by `STRUCT:< firstGrid, secondaryGrid, biCubicBezier >;` Notice that the surface does not interpolate the two intermediate curves

12.3.3 Fixed-degrees tensor product surfaces

A quite common case of tensor product surface with different degrees in the u and v parametric directions is the linear/quadratic one. We show its equations as a further example. Hence, in this case we have:

$$\boldsymbol{S}(u,v) = \boldsymbol{U}_1 \ \boldsymbol{M}_1^1 \ \boldsymbol{\mathcal{G}}_2^1 \ \boldsymbol{M}_2^{2^T} \ \boldsymbol{V}^2$$

that can be rewritten as

$$\boldsymbol{S}(u,v) = \boldsymbol{\mathcal{B}}_2^1(u,v) \cdot \boldsymbol{\mathcal{G}}_2^1 = \sum_{i=1}^{2} \sum_{j=1}^{3} B_j^i(u,v) \ \boldsymbol{p}_j^i,$$

where

$$\boldsymbol{\mathcal{B}}_2^1(u,v) = \boldsymbol{\mathcal{B}}_1(u) \otimes \boldsymbol{\mathcal{B}}_2(v)$$

$$\boldsymbol{B}_1(u) = U_1 \ \boldsymbol{M}_1 = \begin{pmatrix} B_1(u) & B_0(u) \end{pmatrix}$$
$$\boldsymbol{B}_2(v) = V_2 \ \boldsymbol{M}_2 = \begin{pmatrix} B_2(v) & B_1(v) & B_0(v) \end{pmatrix}$$

and

$$\boldsymbol{G}_{12} = \begin{pmatrix} \boldsymbol{p}_{11} & \boldsymbol{p}_{12} & \boldsymbol{p}_{13} \\ \boldsymbol{p}_{21} & \boldsymbol{p}_{22} & \boldsymbol{p}_{23} \end{pmatrix}.$$

Example 12.3.8 (Linear/quadratic Bézier surface)
A simple example of implementation of a Bézier surface linear in u and quadratic in v is given in Script 12.3.15. The resulting ruled surface image is presented in Figure 12.14b. In this case the control tensor $\boldsymbol{\mathcal{G}}_2^1$ is a 2×3 `PLaSM` matrix of 3D points.

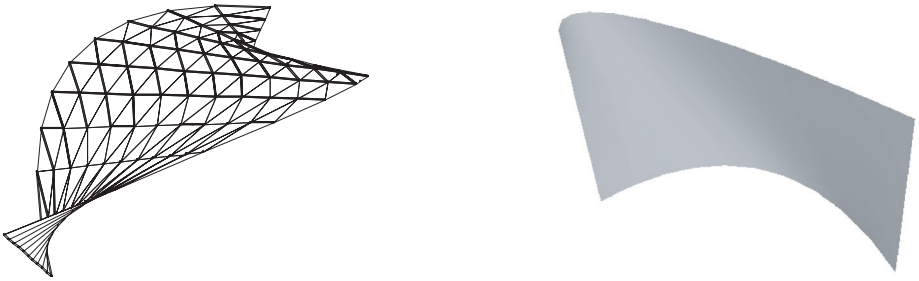

Figure 12.14 Two polynomial surfaces of degrees $(1,2)$ generated by different bases in the v direction

12.3.4 NURB surfaces

NURB surfaces are the tensor product bivariate extension of non-uniform rational B-spline curves. Due to their local control, the exact representation of quadrics, the containment in the local and global convex hulls of control points, and the other properties inherited from NURB curves, such surfaces are nowadays the industry standard representation for geometric modeling and boundary representation (see Section 13.5) of solids.

Script 12.3.15

```
DEF Mapping = BezierSurface:<
   <<0,0,0>, <1,1,-1>, <3,1,0>>,
   <<0,3,0>, <4,3,-2>, <5,4,-1>> >

MAP:Mapping:(Intervals:1:10 * Intervals:1:10)
```

Tensor product NURBS

As we know from Section 11.4.2, a NURB spline curve can be written as

$$\boldsymbol{R}(t) = \bigcup_{i=k}^{m} \boldsymbol{R}_i(t) = \sum_{i=0}^{m} \boldsymbol{p}_i \, N_{i,h}(t), \qquad t \in [t_k, t_{m+1}).$$

where $N_{i,h}(t)$ is the non-uniform rational B-basis function of initial value t_i, order h and degree k, that is zero outside the knot interval $[t_i, t_{i+h})$.

The tensor product NURB surface of orders (h_1, h_2)

$$\boldsymbol{S}(u,v) = \bigcup_{i=k_1}^{m_1} \bigcup_{j=k_2}^{m_2} \boldsymbol{S}_{ij}(u,v) = \sum_{i=0}^{m_1} \sum_{j=0}^{m_2} \boldsymbol{p}_{ij} \, N_{i,j,h_1,h_2}(u,v),$$

with $u \in [u_{k_1}, u_{m_1+1})$, $v \in [v_{k_2}, v_{m_2+1})$, is clearly generated by a tensor product bivatiate basis of (h_1, h_2) orders

$$N_{i,j,h_1,h_2}(u,v) = N_{i,h_1}(u) \, N_{j,h_2}(v), \qquad 0 \le i \le m_1, \ 0 \le j \le m_2,$$

combined with a two-index array $(\boldsymbol{p}_{ij})$ of control points. The surface may be closed using repeated patterns of knots and control points at the extremes of the two sequences. Notice that, whereas control points are just repeated, extreme knots must be translated.

Trimmed NURBS When NURBS surface patches are joined together as the result of Boolean operations between boundary representations[3] of solid models, only portions of a tensor product surface are often used, that may be specified by giving a set of trimming curves in parameter (u, v) space of the patch. Such curves, usually given as 2D NURB splines, are converted into a set of closed loops by a suitable combination with the boundary of parameter space. The used portions of the surface are those internal to the loops, according to some loop orientation. Notice that in order to have a meaningfull description, the loops must not intersect themselves or each other. The geometry representation of the boundary using trimmed NURB surfaces can represent objects of considerable complexity, as well as to exactly represent spheres, cylinders, cones and toruses, and their Boolean combinations. Trimmed NURBS are currently the industry stardard representation of surfaces within the kernel geometric libraries adopted by most CAD systems.

[3] See Section 13.5.

12.4 Higher-order tensor products

The bases of polynomials used to produce the generating maps of three-variate parametric solids can be generated by tensor product of 3 univariate bases. More generally, the same approach can be used to generate d-variate parametric manifolds.

12.4.1 Multivariate Bézier manifolds

In this section the standard tensor product approach for Bézier surfaces is extended to Bézier solids and to d-variate Bézier manifolds.

Parametric curves and surfaces Let us remember that Bézier maps for curves and surfaces are respectively described, in form of combinations of control points with blending polynomials, as:

$$\begin{aligned} C(u) &= \sum_{i=0}^{n} B_i^n(u) p_i \\ S(u,v) &= \sum_{j=0}^{m} B_j^m(v) \sum_{i=0}^{n} B_i^n(u) p_{i,j} \\ &= \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u) B_j^m(v) p_{i,j} = \sum_{i,j=0}^{n,m} B_{i,j}^{n,m}(u,v) p_{i,j} \end{aligned}$$

Parametric solids Analogously, for three-variate bodies we have:

$$B(u,v,w) = \sum_{k=0}^{l} B_k^l(w) \sum_{j=0}^{m} B_j^m(v) \sum_{i=0}^{n} B_i^n(u) p_{i,j,k}$$

$$= \sum_{i=0}^{n}\sum_{j=0}^{m}\sum_{k=0}^{l} B_i^n(u) B_j^m(v) B_k^l(w) p_{i,j,k} = \sum_{i,j,k=0}^{n,m,l} B_{i,j,k}^{n,m,l}(u,v,w) p_{i,j,k},$$

where the blending functions are the Bernstein's polynomials and $p_i, p_{i,j}, p_{i,j,k} \in \mathbb{E}^d$ can be seen as the elements of a tensor of points with 1,2 or 3 indices, respectively.

Multivariate manifolds A *d-variate Bézier Manifold map* of integer degrees $n_1, n_2, ..., n_d$ is a map $M : \mathbb{R}^d \to \mathbb{E}^n$ such that

$$\begin{aligned} M(u_1, u_2, \ldots, u_d) &= \sum_{i_1=0}^{n_1} B_{i_1}^{n_1}(u_1) \sum_{i_2=0}^{n_2} B_{i_2}^{n_2}(u_2) \cdots \sum_{i_d=0}^{n_d} B_{i_1}^{n_d}(u_d) \cdot p_{i_1,i_2,\ldots,i_d} \\ &= \sum_{i_1,i_2,\ldots,i_d=0}^{n_1,n_2,\ldots,n_d} B_{i_1,i_2,\ldots,i_d}^{n_1,n_2,\ldots,n_d}(u_1, u_2, \ldots, u_d) \cdot p_{i_1,i_2,\ldots,i_d}, \quad (12.8) \end{aligned}$$

where $p_{i_1,i_2,\ldots,i_d} \in \mathbb{E}^n$, and where

$$B_{i_1,i_2,\ldots,i_d}^{n_1,n_2,\ldots,n_d}(u_1, u_2, \ldots, u_d) = B_{i_1}^{n_1}(u_1) B_{i_2}^{n_2}(u_2) \cdots B_{i_1}^{n_d}(u_d). \quad (12.9)$$

A *d-variate Bézier manifold* can be seen as the image set $M(U) \subset \mathbb{R}^n$ of a d-variate Bézier map on a compact domain $U \subset \mathbb{R}^d$. Notice that M depends only on the sequence of degrees $n_1, n_2, \ldots, n_d$ and on the tensor of control points $p^{i_1,i_2,\cdots,i_d}$.

Tensor product A tensorial formulation of this mapping can be given as follows. Let us denote with

$$\mathcal{B}^n = (B_i^n)_{i=0,\cdots,n}$$

the Bézier basis of order n, i.e. a one index tensor of functions $\mathbb{R} \to \mathbb{R}$. A left associative tensor product of such function tensors is

$$\mathcal{B}^{n_1\cdots n_h} \otimes \mathcal{B}^m = \mathcal{B}^{n_1\cdots n_h m} = \left(B_{i_1\cdots i_h i_{h+1}}^{n_1\cdots n_h m}\right)_{\substack{i_1=0,\cdots,n_1 \\ \cdots \quad \cdots \\ i_{h+1}=0,\cdots,m}}$$

where

$$B_{i_1 i_2\cdots i_d}^{n_1 n_2\cdots n_d} = B_{i_1}^{n_1} B_{i_2}^{n_2} \cdots B_{i_d}^{n_d} : \mathbb{R}^d \to \mathbb{R}$$

such that

$$(u_1, u_2, \ldots, u_d) \mapsto B_{i_1}^{n_1}(u_1)\, B_{i_2}^{n_2}(u_2) \cdots B_{i_1}^{n_d}(u_d).$$

It follows that the Bézier tensor $\mathcal{B}^{n_1 n_2\cdots n_d}$ can be computed as

$$\mathcal{B}^{n_1 n_2\cdots n_d} = \mathcal{B}^{n_1} \otimes \mathcal{B}^{n_2} \otimes \cdots \otimes \mathcal{B}^{n_d}.$$

Tensor of control points A similar notation is used for a tensor (depending on d indices) of points of an Euclidean space $\mathbb{E}^n$

$$P^{n_1\cdots n_d} = \left(p^{i_1\cdots i_d}\right)_{\substack{i_1=0,\cdots,n_1 \\ \cdots \quad \cdots \\ i_d=0,\cdots,d}}, \qquad p^{i_1\cdots i_d} \in \mathbb{E}^n$$

and for the corresponding tensors of coordinates, with $p = (x_1, \ldots, x_d)$

$$X_k^{n_1\cdots n_d} = \left(x_k^{i_1\cdots i_d}\right)_{\substack{i_1=0,\cdots,n_1 \\ \cdots \quad \cdots \\ i_d=0,\cdots,d}}, \qquad x_k^{i_1\cdots i_d} \in \mathbb{R}$$

Two other concepts remain to be defined. The constant functional, which transforms a tensor of numbers in a tensor of constant functions, and the inner product of tensors. The constant functional K from tensors of numbers to tensors of constant functions is defined as:

$$K(X_k^{n_1\cdots n_d}) \equiv \mathcal{X}_k^{n_1\cdots n_d} = (\chi_k^{i_1\cdots i_d})$$

where

$$\chi_k^{i_1\cdots i_d} : \mathbb{R}^d \to \mathbb{R} : (u_1, \ldots, u_d) \mapsto x_k^{i_1\cdots i_d}$$

for any $(u_1, \ldots, u_d)$.

Inner product of tensors Finally, the inner product of function tensors of the same order is defined as

$$\mathcal{B}^{n_1 \cdots n_d} \cdot \mathcal{X}_k^{n_1 \cdots n_d} = \sum_{i_1, \cdots, i_d = 0}^{n_1, \cdots, n_d} B_{i_1 \cdots i_d}^{n_1 \cdots n_d} \; \chi_k^{i_1 \cdots i_d}$$

where summation and product are between $\mathbb{R} \to \mathbb{R}$ functions.

Implementation A bottom-up approach is assumed in implementing the `BezierManifold` generator function. First, a set of basic functions is defined, in order to compute factorials, binomial coefficients, the Bernstein function $B_i^n : \mathbb{R} \to \mathbb{R}$ and the Bernstein base $\mathcal{B}^n = (B_i^n)$. The `Bernstein` operator seems a little tricky, but it implements a functional with signature `Int` $\times$ `Int` $\to$ (`Real` $\to$ `Real`).

Hence the most natural functional interface for the definition of a Bézier d-manifold is given in Script 12.4.1.

Script 12.4.1 (Bézier manifolds by tensor product)

```
DEF BezierManifold (degrees::IsSeq) (controlPoints::IsSeq) =
   (AA:TensorInnerProd ~ DISTL):<BezierTensor,CoordTensors>
WHERE
   BezierTensor = (INSR:TensorProd ~ AA:BernsteinBase):degrees,
   CoordTensors = AA:ConstFunTensor:controlPoints
END;
```

Note It is very important to note that such an approach works correctly for an arbitrary polyhedral domain $U \subset \mathbb{R}^d$. In particular, `domain` may coincide both with a d-dimensional interval and with any m-dimensional polyhedral subset of it, $0 \leq m \leq d$. For instance, a 3-variate Bézier map can be applied either to a 3D interval, or to a 3D polyhedron, or to a (set of) polyhedral surface(s), or to a (set of) polyhedral curve(s) or even to a (set of) point(s). The only constraint is that both `controlPoints` and the Bézier manifold, i.e. $M(U)$, live in the same space, i.e. have the same number n of coordinates. It may be useful to note that, in order to always correctly obtain a piece-wise linear result of any mapping, a simplicial decomposition of the `domain` is always performed by the `MAP` function.

Example 12.4.1 (Bézier 3-manifold)
The definition of a `BezierExample` can simply be the application of `Manifold` function on actual parameters `BezierMap` and `domain`. The specialized `BezierMap` is obtained by applying the `BezierManifold` operator of Script 12.4.2 to the degrees `<2,2,2>`, and to the tensor $\mathbb{P}^{3,3,3}$ of its control points. The result is a three-variate `BezierMap` : $\mathbb{R}^3 \to \mathbb{R}^3$ whose three parameters have all degree two.

In Figure 12.15 the results generated by the `Bezier_example` for different values of `domain` are reported. The four definitions of the polyhedral `domain` are referred to a 1-dimensional grid. Four polyhedral values generated by the function `Bezier_example` varying the definition of the `domain` are generated by Script 12.4.3.

Script 12.4.2 (Bézier 3-manifold example (1))

```
DEF BezierExample = MAP:BezierMap:domain

DEF BezierMap = BezierManifold:degrees:<Xtensor,Ytensor,Ztensor>
WHERE
   degrees = <2,2,2>,
   Xtensor = <<<0,1,2>,<-1,0,1>,<0,1,2>>,
              <<0,1,2>,<-1,0,1>,<0,1,2>>,
              <<0,1,2>,<-1,0,1>,<0,1,2>>>,
   Ytensor = <<<0,0,0.8>,<1,1,1>,<2,3,2>>,
              <<0,0,0.8>,<1,1,1>,<2,3,2>>,
              <<0,0,0.8>,<1,1,1>,<2,3,2>>>,
   Ztensor = <<<0,0,0>,<0,0,0>,<0,0,0>>,
              <<1,1,1>,<1,1,1>,<1,1,1>>,
              <<2,2,1>,<2,2,1>,<2,2,1>>>
END;
```

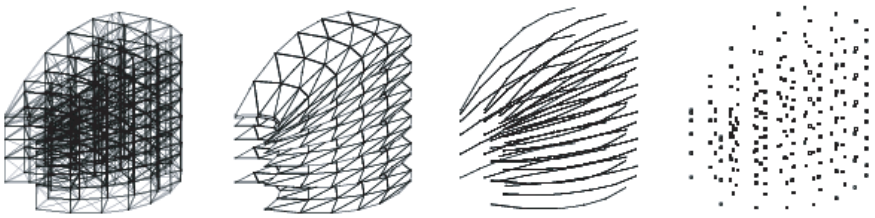

Figure 12.15 (a) Three-quadratic Bézier solid (b), (c) and (d) Submanifolds of dimensions 2, 1 and 0, respectively

12.5 Transfinite methods

Transfinite blending stands for interpolation or approximation in *functional spaces*. In this case a bivariate mapping is generated by blending some univariate maps with a suitable basis of polynomials. Analogously, a three-variate mapping is generated by blending some bivariate maps with a polynomial basis, and so on. Transfinite methods are quite frequently used in CAD applications, mainly for automobile, ship and airplane shell design. It is sometimes called *function blending* [Gor68, LS86], or *transfinite interpolation* [Gor69, Gol87]. Gordon-Coons's patches were discovered [Coo67, Gor68] in the MIT project MAC at the end of the 1960s. The

Script 12.4.3 (Bézier 3-manifold example (2))

```
DEF grid1D = (QUOTE∼#:step):(1/step) WHERE step = 5 END;

DEF domain = grid1D * grid1D * grid1D;
DEF domain = grid1D * grid1D * @0:grid1D;
DEF domain = grid1D * @0:grid1D * @0:grid1D;
DEF domain = @0:grid1D * @0:grid1D * @0:grid1D

MAP:(BezierManifold:degrees:controlPoints):domain
```

skinning of grids of curves used by high-end animation systems is a transfinite interpolation. Transfinite methods have recently been applied to interpolate implicitly defined sets of different cardinality [RSST01].

The implementation of transfinite blending with `PLaSM` consists in using functions without variables, that can be combined, e.g. multiplied and added, exactly as numbers. This approach seems both very powerful and simple. Several examples of such power are given in the following sections. Consider, e.g., that multivariate transfinite Bézier blending of *arbitrary degree* with both domain and range spaces of *arbitrary dimension* is implemented in Section 12.5.4 with only 11 lines (!) of quite readable source code.

12.5.1 Rationale of the approach

Transfinite blending is a strong generalization of standard parametric methods. For example, the standard Hermite generation of cubic curves, where two extreme points and tangents are interpolated, can be readily applied to surfaces, where two extreme curves are interpolated with assigned derivative curves, as well as to volume interpolation of two assigned surfaces with assigned derivatives fields.

Definition A *d-variate parametric mapping* is a point-valued polynomial function $\mathbf{\Phi} : U \subset X \to Y$ with degree k, domain U, support $X = \mathbb{R}^d$ and range $Y = \mathbb{E}^n$.

As commonly used in Computer Graphics and CAD, such point-valued polynomials $\mathbf{\Phi} = (\Phi_j)_{j=1,\dots,n}$ belong component-wise to the vector space $\mathbb{P}_k[\mathbb{R}]$ of polynomial functions of bounded integer degree k over the $\mathbb{R}$ field.

Polynomial coordinates Since the set $\mathbb{P}_k$ of polynomials is also a vector space $\mathbb{P}_k[\mathbb{P}_k]$ over $\mathbb{P}_k$ itself *as a field*, then each mapping component Φ_j, $1 \leq j \leq n$, can be expressed uniquely as a linear combination of $k+1$ basis elements $\phi_i \in \mathbb{P}_k$ with *coordinates* $\xi_j^i \in \mathbb{P}_k$, so that

$$\Phi_j = \xi_j^0 \phi_0 + \cdots + \xi_j^k \phi_k, \qquad 1 \leq j \leq n.$$

Hence a unique coordinate representation

$$\Phi_j = \left(\xi_j^0, \dots, \xi_j^k\right)^T, \qquad 1 \leq j \leq n$$

of the mapping is given, after a basis $\{\phi_0, \dots, \phi_k\} \subset \mathbb{P}_k$ has been chosen.

Choice of a basis If the basis elements are collected into a vector $\boldsymbol{\phi} = (\phi_i)$, then it is possible to write:

$$\mathbf{\Phi} = \mathbf{\Xi}\ \boldsymbol{\phi}, \qquad \boldsymbol{\phi} \in \mathbb{P}_k^{k+1},\ \mathbf{\Phi} \in \mathbb{P}_k^n.$$

where

$$\mathbf{\Xi} = \left(\xi_j^i\right), \qquad 1 \leq i \leq n,\ 0 \leq j \leq k.$$

is the coordinate representation of a linear operator in $Lin[n \times (k+1), \mathbb{P}_k]$ that maps the $k+1$ basis polynomials of k degree, into the n polynomials which transform

the vectors in the domain $U \subset \mathbb{R}^d$ into the $\mathbb{E}^n$ points of the manifold. A quite standard choice in computer-aided geometric modeling is $U = [0,1]^d$. As seen in previous sections, 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 the $\boldsymbol{\phi} = (\phi_i)$ basis. This approach can be readily extended to the space $\mathcal{Z}_k$ of rational functions of degree k.

Blending operator The *blending operator* $\boldsymbol{\Xi}$ specializes the maps generated by a certain basis, to fit and/or to approximate a given set of specific data (points, tangent vectors, boundary curves or surfaces, boundary derivatives, and so on). Its coordinate functions ξ_j^i may be easily generated, as will be shown in the following subsections, by either

1. transforming the "geometric handles" of the mapping into vectors of constant functions, in the standard (non-transfinite) case. These are usually points or vectors $\boldsymbol{x}_j = (x_j^i) \in \mathbb{E}^n$ to be interpolated or approximated by the set $\boldsymbol{\Phi}(U)$;
2. assuming directly as ξ_j^i the components of the curve (surface, etc.) maps to be fitted or approximated by $\boldsymbol{\Phi}$, in the transfinite case.

Notation For the sake of readability, only Greek letters, either capitals or lower-case, are used in this section to denote functions. Latin letters are used for numbers and vectors of numbers. As usual in this book, bold letters denote vectors, points or tensors. Please remember that B and H are also Greek capitals for β and η, respectively.

12.5.2 Univariate case

Let us consider the simple univariate case $\boldsymbol{\Phi} : U \subset X \to Y$, where the dimension d of support space X is one. To generate the coordinate functions ξ_j^i it is sufficient to transform each data point $\boldsymbol{x}_i = (x_j^i) \in Y$ into a vector of constant functions, so that

$$\xi_j^i = \kappa(x_j^i), \quad \text{where} \quad \kappa(x_j^i) : U \to Y : u \mapsto x_j^i.$$

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

$$\kappa(x_j^i)(u) = x_j^i$$

for each parameter value $u \in U$. The `PLaSM` implementation clearly uses the constant functional `K` at this purpose.

Example 12.5.1 (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 in Script 12.5.1 as formal parameters of the function `Bezier3`, that generates the Φ mapping by linearly combining the basis functions with the coordinate functions. The local functions `b0,b1,b2,b3` implement the cubic Bernstein/Bézier basis functions $\beta_i^3 : \mathbb{R} \to \mathbb{R}$ such that $u \mapsto \binom{3}{i} u^i (1-u)^{3-i}$, $0 \le i \le 3$.

The `x` and `y` functions, defined as composition of a selector with the constant functional `K`, respectively select the first (second) component of their argument

Script 12.5.1

```
DEF Bezier3 (p0,p1,p2,p3::IsSeqOf:isReal) =
   [ (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;
```

sequence and transform such a number in a constant function. The reader should notice that `+` and `*` are used as operators *between functions* in the script above.

12.5.3 Multivariate case

When the dimension d of the support 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 that we already know (see Sections 12.3 and 12.4). The second method is called *function blending*, also known as *transfinite blending*.

Transfinite blending Let us consider a polynomial mapping $\boldsymbol{\Phi} : U \to Y$ of degree k_d, where U is d-dimensional and Y is n-dimensional. Since $\boldsymbol{\Phi}$ depends on d parameters, in the following it will be denoted as ${}^d\boldsymbol{\Phi} = ({}^d\Phi_j)$, $1 \le j \le n$. In this case ${}^d\boldsymbol{\Phi}$ is computed component-wise by linear combination of coordinate maps, depending on $d-1$ parameters, with the *univariate* polynomial basis $\boldsymbol{\phi} = (\phi_i)$ of degree k_d. Formally we can write:

$$ {}^d\Phi_j = {}^{d-1}\Phi_j^0\, \phi_0 + \cdots {}^{d-1}\Phi_j^{k_d}\, \phi_{k_d}, \qquad 1 \le j \le n. $$

The coordinate representation of ${}^d\Phi_j$ with respect to the basis $(\phi_0, \ldots, \phi_{k_d})$ is thus given by $k_d + 1$ maps depending on $d-1$ parameters:

$$ {}^d\Phi_j = \left({}^{d-1}\Phi_j^0, \ldots, {}^{d-1}\Phi_j^{k_d} \right). $$

In matrix notation, after a polynomial basis $\boldsymbol{\phi}$ has been chosen, it is

$$ {}^d\Phi = \boldsymbol{\Xi}\, \boldsymbol{\phi}, \quad \text{where} \quad \boldsymbol{\Xi} = ({}^{d-1}\Phi_j^i), \quad \boldsymbol{\phi} = (\phi_i), \quad 0 \le i \le k_d,\ 1 \le j \le n. $$

Example 12.5.2 (Bicubic Bézier)
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 $B_i(u_1)$, with $0 \le i \le 3$, where some curve maps may possibly reduce to a constant point map:

$$ B(u_1, u_2) = \sum_{i=0}^{3} B_i(u_1)\ \beta_i^3(u_2) $$

where

$$\beta_i^3(u) = \binom{3}{i} u^i \, (1-u)^{3-i}, \qquad 0 \le i \le 3,$$

is the Bernstein/Bézier cubic basis. Analogously, a three-variate Bézier solid body mapping $B(u_1, u_2, u_3)$, of degree k_3 on the last parameter, may be generated by univariate Bézier blending of surface maps $B_i(u_1, u_2)$, some of which may possibly be reduced to a curve map or even to a constant point map:

$$B(u_1, u_2, u_3) = \sum_{i=0}^{k_3} B_i(u_1, u_2) \; \beta_i^{k_3}(u_3)$$

Note The more interesting aspects of transfinite blending are *flexibility* and *simplicity*. Unlike the tensor-product method, there is no need for all component geometries to have the same degree, nor be 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.

12.5.4 Transfinite Bézier

The full power of the `PLaSM` functional approach to geometric programming is used in this section, where dimension-independent transfinite Bézier blending of any degree is implemented in few lines of code, by easily combining coordinate maps which may depend on an arbitrary number of parameters.

We note that the `Bezier` : $[0,1]^d \to \mathbb{E}^n$ mapping given here can be used:

1. to blend points to give curve maps;
2. to blend curve maps to give surface maps;
3. to blend surface maps to give solid maps;
4. and so on ...

Notice also that the given implementation is independent on the dimensions d and n of support and range spaces.

Implementation For this purpose, first a small toolbox of related functions is needed, to compute the factorial function, the binomial coefficients, the $\boldsymbol{\beta}^k = (\beta_i^k)$ Bernstein basis of degree k, and the β_i^k Bernstein/Bézier polynomials.

Then the `Bezier:u` function is given in Script 12.5.3, to be applied on the sequence of `ControlData`, which may contain either control points $\boldsymbol{x}_i = (x_j^i)$ or control maps ${}^{d-1}\Phi_i = ({}^{d-1}\Phi_j^i)$, with $0 \le i \le k$, $1 \le j \le n$. In the former case each component x_j^i of each control point is first transformed into a constant function.

The body of the `Bezier:u` function just linearly combines component-wise the sequence (ξ_j^i) of coordinate maps generated by the expression

Script 12.5.2 (Transfinite Bézier toolbox)

```
DEF Fact (n::IsInt) = IF:< C:EQ:0, K:1, * ~ INTSTO >;
DEF Choose = IF:< OR ~ [C:EQ:0 ~ S2, EQ], K:1, Choose ~ AA:(C:+:-1) * / >;
DEF Bernstein (u::IsFun)(n::IsInt)(i::IsInt) =
   * ~ [K:(Choose:<n,i>),** ~ [ID,K:i], ** ~ [- ~ [K:1,ID],K:(n-i)]] ~ u;
DEF BernsteinBasis (u::IsFun)(n::IsInt) = AA:(Bernstein:u:n):(0..n);
```

```
(TRANS~fun):ControlData
```

with the basis sequence (β_i^k) generated by `BernsteinBasis:u:degree`, where `degree` equates the number of geometric handles minus one.

Script 12.5.3 (Dimension-independent transfinite Bézier mapping)

```
DEF Bezier (u::IsFun) (ControlData::IsSeq) = (AA:InnerProd ~ DISTR):
   < (fun ~ TRANS):ControlData, BernsteinBasis: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 the actual argument is 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 12.5.3 (Bézier curves and surface)
Four Bézier $[0,1] \to \mathbb{E}^3$ maps `C1`, `C2`, `C3`, and `C4`, respectively of degrees $1, 2, 3$ and 2 are defined in Script 12.5.4.

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 transfinite bivariate (i.e. 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 (here working on the second coordinate of points in $[0,1]^2$) is performed by the `PLaSM` function `Surf1`, defined by using again the `Bezier` function.

A simplicial approximation (with triangles) of the surface $B[0,1]^2 \subset \mathbb{E}^3$ is finally generated by evaluating the last expression of Script 12.5.4.

According to the semantics of the `MAP` operator, `Surf1` is applied to all vertices of the automatically generated simplicial decomposition Σ of the 2D product $(\texttt{Intervals}:1:20 * \texttt{Intervals}:1:20) \subset \mathbb{R}^2$. A simplicial approximation $B(\Sigma)$ of the surface $B([0,1]^2) \subset \mathbb{E}^3$ is finally produced and displayed in Figure 12.16c.

The four generating curves and the generated cubic blended surface are displayed in Figure 12.16. It is possible to see (Figure 12.16) that such surface interpolates the four boundary curves defined by the extreme control points, exactly as in the case of

Script 12.5.4

```
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>>;

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

MAP: Surf1: (Intervals:1:20 * Intervals:1:20);
```

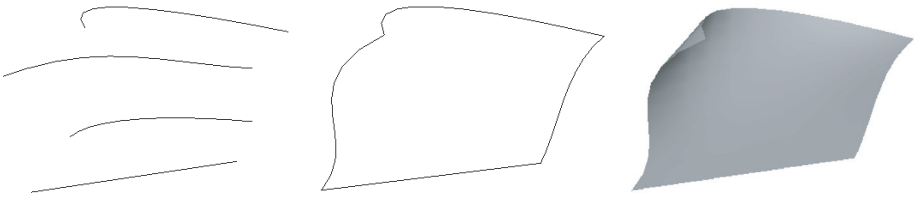

Figure 12.16 (a) Graphs of four Bézier curve maps `c0`, `c1`, `c2` and `c3` (b) Graphs of `c0` and `c3` together with graphs of bicubic maps `b0` and `b1` generated by extreme control points (c) Graph of `Surf1` surface

tensor-product method, but obviously with much greater generality, since any defining curve may be of any degree.

12.5.5 Transfinite Hermite

The cubic Hermite univariate map is the unique cubic polynomial $\boldsymbol{H} : [0,1] \to \mathbb{E}^n$ which matches two given points $p_0, p_1 \in \mathbb{E}^n$ and derivative vectors $t_0, t_1 \in \mathbb{R}^n$ for $u = 0, 1$ respectively. Let us denote as $\eta^3 = \left(\eta_0^3, \eta_1^3, \eta_2^3, \eta_3^3\right)$ the cubic Hermite function basis, with

$$\eta_i^3 : [0,1] \to \mathbb{R}, \qquad 0 \le i \le 3,$$

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 $\boldsymbol{H}$ can be written, in vector notation, as

$$\begin{aligned} \boldsymbol{H} &= \boldsymbol{\xi}_0\, \eta_0^3 + \boldsymbol{\xi}_1\, \eta_1^3 + \boldsymbol{\xi}_2\, \eta_2^3 + \boldsymbol{\xi}_3\, \eta_3^3 \\ &= \kappa(\boldsymbol{p}_0)\, \eta_0^3 + \kappa(\boldsymbol{p}_1)\, \eta_1^3 + \kappa(\boldsymbol{t}_0)\, \eta_2^3 + \kappa(\boldsymbol{t}_1)\, \eta_3^3. \end{aligned}$$

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

$$\begin{aligned} \boldsymbol{H}(0) &= \kappa(\boldsymbol{p}_0)(0) = \boldsymbol{p}_0, \quad \boldsymbol{H}(1) = \kappa(\boldsymbol{p}_1)(1) = \boldsymbol{p}_1, \\ \boldsymbol{H}'(0) &= \kappa(\boldsymbol{t}_0)(0) = \boldsymbol{t}_0, \quad \boldsymbol{H}'(1) = \kappa(\boldsymbol{t}_1)(1) = \boldsymbol{t}_1, \end{aligned}$$

and that the image set $\boldsymbol{H}[0,1]$ is the desired curve in $\mathbb{E}^n$.

A multivariate transfinite Hermite map $\boldsymbol{H}^n$ can easily be defined by allowing the blending operator $\boldsymbol{\Xi} = \left(\boldsymbol{\xi}_j\right) = \left(\xi_j^i\right)$ to contain maps depending at most on $d-1$ parameters.

Implementation A transfinite `CubicHermite` mapping is implemented here, with four data objects given as formal parameters. Such data objects may be either points/vectors, i.e. sequences of numbers, or $1/2/3/d$-variate maps, i.e. sequences of (curve/surface/solid/etc.) component maps, or even mixed sequences, as will be shown in the following examples.

Script 12.5.5 (Dimension-independent transfinite cubic Hermite)

```
DEF HermiteBasis (u::IsFun) = < h0,h1,h2,h3 >
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;

DEF CubicHermite (u::IsFun) (p1,p2,t1,t2::IsSeq) =
   (AA:InnerProd ~ DISTL): < HermiteBasis:u, (TRANS ~ fun):<p1,p2,t1,t2> >;
```

12.5.6 Connection surfaces

The creation of surfaces smoothly connecting two given curves with assigned derivative fields by cubic transfinite blending is discussed in this section. The first applications concern the generation of surfaces in 3D space, the last concern the generation of planar grids as 1-skeletons of 2D surfaces, according to the dimension-independent character of the given `PLaSM` implementation of transfinite methods.

The curve maps $\texttt{c1}(u)$ and $\texttt{c2}(u)$ of Example 12.5.6 are here interpolated in 3D by a `Surf2` mapping using the cubic Hermite basis $\eta^3 = (\eta^3_j)$, $0 \leq j \leq 3$, with the further constraints that the tangent vector field $\texttt{Surf2}^v(u, 0)$ along the first curve are constant and parallel to $(0, 0, 1)$, whereas $\texttt{Surf2}^v(u, v)$ along the second curve is also constant and parallel to $(0, 0, -1)$. The resulting map

$$\texttt{Surf2} : [0, 1]^2 \to \mathbb{E}^3$$

has unique vector representation in $\mathbb{P}^3_3[\mathbb{P}_3]$ as

$$\texttt{Surf2} = \texttt{c1}\ \eta^3_0 + \texttt{c2}\ \eta^3_1 + (\kappa(0), \kappa(0), \kappa(1))\ \eta^3_2 + (\kappa(0), \kappa(0), \kappa(-1))\ \eta^3_3.$$

Example 12.5.4 (Surface interpolation of curves)
Such a map is very easily implemented by the following `PLaSM` definition. A simplicial approximation $\texttt{Surf2}(\Sigma)$ of the point set $\texttt{Surf2}([0, 1]^2)$ is generated by the `MAP` expression in Script 12.5.6, and is shown in Figure 12.17.

Example 12.5.5 (Surface interpolation of curves)
A different surface interpolation of the two plane curves `c1` and `c2` is given in Script 12.5.7, where the boundary tangent vectors are constrained to be constant

Script 12.5.6 (Surface by Hermite's interpolation (1))

```
DEF Surf2 = CubicHermite:S2:< c1,c2,<0,0,1>,<0,0,-1> >;
MAP: Surf2: (Domain:14 * Domain:14);
```

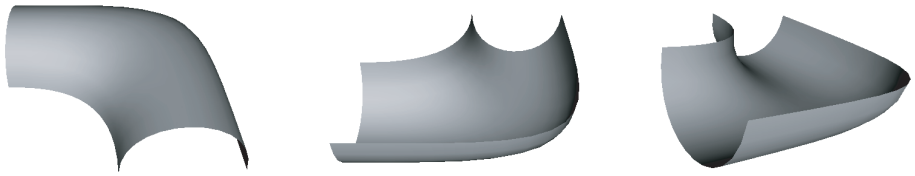

Figure 12.17 Some pictures of the surface interpolating two plane Hermite curves with constant vertical tangent vectors along the curves.

and parallel to $(1, 1, 1)$ and $(-1, -1, -1)$, respectively. The resulting surface is given in Figure 12.18. The same surface is used in Section 5.8.1 of the chapter on Differential Geometry, to compute and display the Gauss' curvature field over a surface.

Script 12.5.7 (Surface by Hermite's interpolation (2))

```
DEF Surf3 = CubicHermite:S2:<c1,c2,<1,1,1>,<-1,-1,-1>>;
MAP: Surf3: (Domain:14 * Domain:14);
```

Example 12.5.6 (Grid generation)
Two planar Hermite curve maps `c1` and `c2` are defined, so that the curve images `c1`$[0, 1]$ and `c2`$[0, 1]$.

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

The grids generated by maps `grid1`, `grid2` and `grid3` are shown in Figure 12.19. The tangent map `dd` is simply obtained as component-wise difference of the curve maps `c2` and `c1`.

It is interesting to notice that the map `grid1` can be also generated as a linear (transfinite) Bézier interpolation of the two curves, as given below. Clearly the solution as cubic Hermite is more flexible, as it is shown by Figures 12.19b and 12.19c.

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

Example 12.5.7 (Wing section grid)
A 2D grid for a computational problem is generated in a domain of E^2 bounded by the four Hermite curves given in Script 12.5.9 and shown in Figure 12.20a.

First, a function `Norm` is defined in Script 12.5.10 to return the 2D unit normal vector in $\mathbb{P}^2_k$ generated by a formal parameter `fun2` $\in \mathbb{P}^2_k$. Remember that if $v = (x, y)$, then $v^\perp = (-y, x)$. The Euclidean vector norm as squared root of the sum of squared coordinates is used.

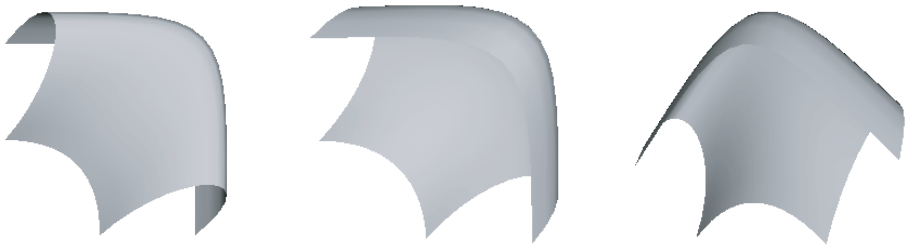

Figure 12.18 Some pictures of a new surface interpolating the same Hermite curves with constant oblique tangent vectors.

Script 12.5.8 (Examples of planar grids)

```
DEF c1 = CubicHermite:S1:<<1,0>,<0,1>,<0,3>,<-3,0>>;
DEF c2 = CubicHermite:S1:<<0.5,0>,<0,0.5>,<0,1>,<-1,0>>;
DEF dd = (AA:- ∼ TRANS):<c2,c1>;

DEF grid1 = CubicHermite:S2:<c1,c2,dd,dd>;
DEF grid2 = CubicHermite:S2:<c1,c2,<-0.5,-0.5>,dd>;
DEF grid3 = CubicHermite:S2:<c1,c2,<S1:dd,-0.5>,dd>;
```

Finally two different discretization of the interval $(0, 1]$ are prepared in Script 12.5.11, together with two functions to be used for abstracting the 1-variate and the 2-variate simplicial mapping:

Then two bivariate (transfinite) cubic Hermite maps are generated between the wing profile curves and the boundary curves, using normal univariate maps.

The geometry discretization shown in Figures 12.20a and 12.20b are respectively generated by the last two `PLaSM` expressions. The approach shown here in 2D may be readily applied to the volume discretization of much more complicated 3D domains.

12.5.7 Coons' surfaces

We know that a parametric surface is a function $\boldsymbol{S} : U \subset \mathbb{R}^2 \to \mathbb{E}^n$. Let assign four boundary curves, corresponding to the edges of the U domain:

$$\boldsymbol{S}(u, 0), \quad \boldsymbol{S}(u, 1), \quad \boldsymbol{S}(0, v), \quad \boldsymbol{S}(1, v).$$

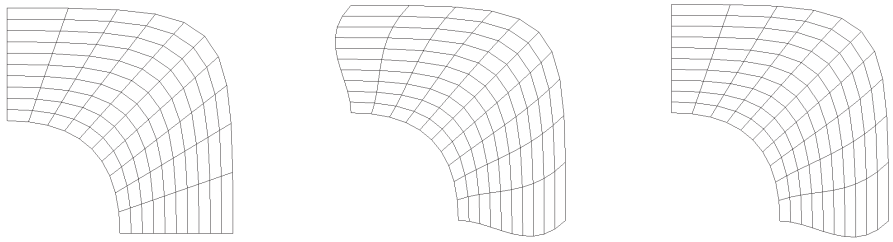

Figure 12.19 The simplicial complexes generated by the `MAP` operator on the `grid1`, `grid2` and `grid3` maps given in Example 12.5.6.

Script 12.5.9 (Wing profile)

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

Script 12.5.10 (Perpendicular to a 2D function vector)

```
DEF Norm (fun2::IsSeqOf:IsFun) = <- ~ 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;
```

A surface [Coo67] which interpolates the given curves is known as *Coons' patch*, that usually interpolates the four boundary curves by using linear blending functions as

$$\alpha_0(t) = 1 - t, \qquad \alpha_1(t) = t.$$

The vector expression of the Coons' surface patch is:

$$\begin{aligned}
\boldsymbol{S}(u,v) &= \boldsymbol{S}_1(u,v) + \boldsymbol{S}_2(u,v) - \boldsymbol{S}_3(u,v) \\
&= \begin{pmatrix} \alpha_0(u) & \alpha_1(u) \end{pmatrix} \begin{pmatrix} \boldsymbol{S}(0,v) \\ \boldsymbol{S}(1,v) \end{pmatrix} + \\
&\quad \begin{pmatrix} \boldsymbol{S}(u,0) & \boldsymbol{S}(u,1) \end{pmatrix} \begin{pmatrix} \alpha_0(v) \\ \alpha_1(v) \end{pmatrix} - \\
&\quad \begin{pmatrix} \alpha_0(u) & \alpha_1(u) \end{pmatrix} \begin{pmatrix} \boldsymbol{S}(0,0) & \boldsymbol{S}(0,1) \\ \boldsymbol{S}(1,0) & \boldsymbol{S}(1,1) \end{pmatrix} \begin{pmatrix} \alpha_0(v) \\ \alpha_1(v) \end{pmatrix}
\end{aligned}$$

The above equation can be interpreted as the sum of three signed bivariate vector functions, where

1. $\boldsymbol{S}_1(u,v)$ is the ruled surface interpolating $\boldsymbol{S}(0,v)$ and $\boldsymbol{S}(1,v)$;
2. $\boldsymbol{S}_2(u,v)$ is the ruled surface interpolating $\boldsymbol{S}(u,0)$ and $\boldsymbol{S}(u,1)$;
3. $\boldsymbol{S}_3(u,v)$ is the bilinear surface interpolating the four corner points $\boldsymbol{S}(0,0)$, $\boldsymbol{S}(0,1)$, $\boldsymbol{S}(1,0)$ and $\boldsymbol{S}(1,1)$.

Implementation The implementation is very easy, and directly corresponds to the mathematical definition of equation (12.10). For this purpose two vector operations `scalarVectProd` and `vectSum` given in Script 2.1.19, for the product of a vector times

Script 12.5.11

```
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);
```

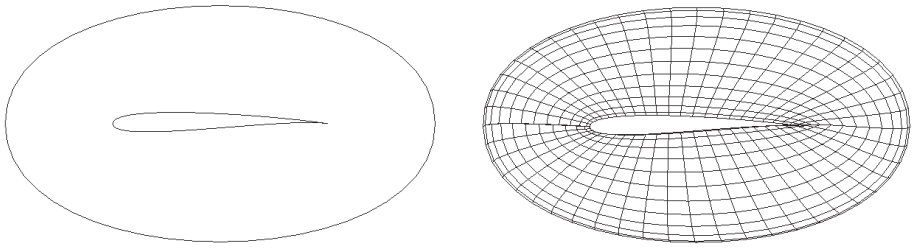

Figure 12.20 (a) The plane domain bounded by the four curve maps `c1, c2` and `b1, b2` (b) Wire-frame picture of domain `grid1`$(\Sigma[0,1]^2) \cup$ `grid2`$(\Sigma[0,1]^2)$.

Script 12.5.12 (Domain discretization)

```
DEF grid1 = CubicHermite:S2:<c1,b1,Norm:c1,Norm:b1>;
DEF grid2 = CubicHermite:S2:<c2,b2,Norm:c2,Norm:b2>;
(STRUCT ~ AA:graph1):<c1,c2,b1,b2>;
(STRUCT ~ AA:graph2):<grid1,grid2>;
```

a scalar and for the addition of vectors of any length are used. Notice that such operations are performed between function vectors and not between number vectors.

The 2D domain of the mapping is obtained as the polyhedral product of two partitions of the 1D segment.

Example 12.5.8 (Coons' patch)
Let us suppose that the boundary curves are Bézier curves of degrees 1, 4, 3 and 2, respectively. The composition with the function `[S2]` is needed to reuse the `Bezier PLaSM` script when mapping from a 2D domain. The analogous composition with `[S1]` is given just for symmetry.

A Coons' patch which smoothly interpolates the four given curves is generated by the last expression of Script 12.5.14, and is displayed in Figure 12.21.

Example 12.5.9 (Curve on surface mapping)
No constraints exist in `PLaSM` on the dimension and the topology of the domain of

Script 12.5.13 (Coons' patches)

```
DEF CoonsPatch (Su0,Su1,S0v,S1v::IsSeqOf:IsFun) =
   (vectSum ~ AA:scalarVectProd):
   <<a0u,S0v>,<a1u,S1v>,<a0v,Su0>,<a1v,Su1>,<-~a0v,b0u>,<-~a1v,b1u>>
WHERE
   b0u = (vectSum ~ AA:scalarVectProd):<<a0u,S_00>,<a1u,S_10>>,
   b1u = (vectSum ~ AA:scalarVectProd):<<a0u,S_01>,<a1u,S_11>>,
   S_00 = (AA:K ~ CONS:Su0):<0,0>, S_01 = (AA:K ~ CONS:S0v):<0,1>,
   S_10 = (AA:K ~ CONS:Su0):<1,0>, S_11 = (AA:K ~ CONS:S1v):<1,1>,
   a0u = K:1 - u, a1u = u,
   a0v = K:1 - v, a1v = v,
   u = S1, v = S2
END;

DEF Domain2D (n,m::IsIntPos) = Intervals:1:n * Intervals:1:n;
```

Script 12.5.14

```
DEF Su0 = Bezier:<<0,0,0>,<10,0,0>> ~ [S1];
DEF Su1 = Bezier:<<0,10,0>,<2.5,10,3>,<5,10,-3>,<7.5,10,3>,<10,10,0>> ~ [S1];
DEF S0v = Bezier:<<0,0,0>,<0,0,3>,<0,10,3>,<0,10,0>> ~ [S2];
DEF S1v = Bezier:<<10,0,0>,<10,5,3>,<10,10,0>> ~ [S2];

MAP:(CoonsPatch:<Su0,Su1,S0v,S1v>):(Domain2D:<12,12>)
```

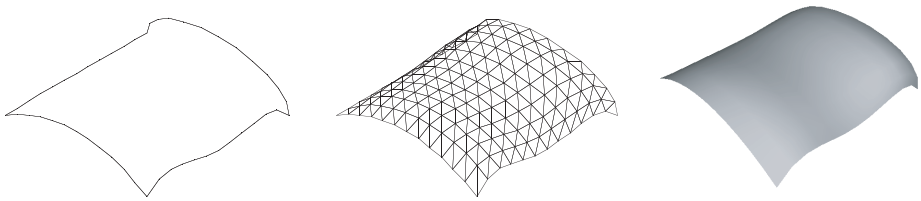

Figure 12.21 (a) Four assigned boundary curves (b) Polyhedral approximation of the Coons' surface; (c) Smooth rendering

a parametric mapping, which can be any d-dimensional polyhedral complex, with $d \leq n$ and n the dimension of the mapping range. Thus, an example is given where a spiral subset of 2D parameter space is mapped using the vector function `CoonsPatch` previously defined.

In particular, let us consider the function `SpiralMap` : $[0, 6\pi] \subset \mathbb{R} \rightarrow \mathbb{E}^2$, such that $t \mapsto (u, v)$, with

$$\begin{aligned} u &= \frac{1}{2} + \frac{1}{2}\left(1 - 0.2\frac{t}{6\pi}\right)\sin t, \\ v &= \frac{1}{2} + \frac{1}{2}\left(1 - 0.2\frac{t}{6\pi}\right)\cos t. \end{aligned}$$

The 2D `Spiral` polyhedron in Figure 12.22a is mapped in $\mathbb{E}^3$ according to the last expression of Script 12.5.15. The resulting curve image in 3D space is shown in Figure 12.22b.

Script 12.5.15

```
DEF SpiralMap = [radius * COS, radius * SIN] ~ S1
WHERE radius = K:1 - (K:(0.2 / (6 * PI)) * ID) END;

DEF Spiral = (T:<1,2>:<0.5,0.5>~S:<1,2>:<0.5,0.5>):
   (MAP:SpiralMap:(Intervals:(6 * PI):100));

MAP:(CoonsPatch:<Su0,Su1,S0v,S1v>):Spiral;
```

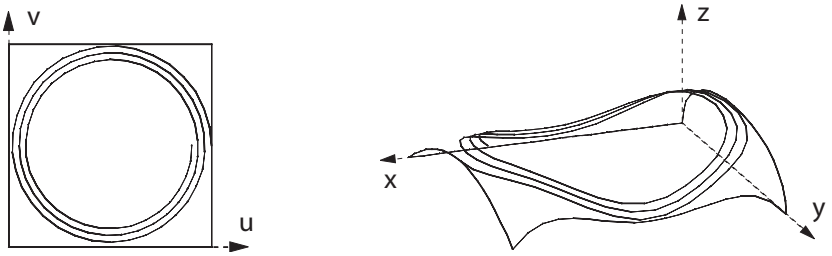

Figure 12.22 (a) Spiral in parametric domain $[0,1]^2$ (b) Its image under the Coons' mapping defined by four 3D curves.

12.5.8 Thin solids generated by surfaces

A thin parametric solid described by a three-variate vector map $\boldsymbol{V}(u,v,w)$ generated by a surface $\boldsymbol{S}(u,v)$ and with constant (small) thickness ℓ may be defined as

$$\boldsymbol{V}(\boldsymbol{S})(u,v,w) = \boldsymbol{S}(u,v) + w\ \boldsymbol{n}(u,v), \qquad u,v,w \in [0,1]^2 \times [0,\ell] \tag{12.10}$$

where

$$\boldsymbol{n}(\boldsymbol{S})(u,v) = \frac{\boldsymbol{S}^u(u,v) \times \boldsymbol{S}^v(u,v)}{\|\boldsymbol{S}^u(u,v) \times \boldsymbol{S}^v(u,v)\|}$$

is the unit vector function normal to the surface. Notice that, using the variable-free notation (see Section 5.1.1) for functions, we have:

$$\boldsymbol{V} : (\mathbb{R}^2 \to \mathbb{E}^3) \to (\mathbb{R}^3 \to \mathbb{E}^3).$$

where

$$\boldsymbol{V}(\boldsymbol{S}) = \boldsymbol{S} + \sigma(3)\ \boldsymbol{n}(\boldsymbol{S}).$$

In other words, the $\boldsymbol{V}$ operator produces a *volume map* starting from a *surface map*.

Implementation Luckily enough, it is very easy to write a `PLaSM` function `ThinSolid` which implements in 3D the $\boldsymbol{V}$ operator described above. First, we remember that the unit *normal* field $\boldsymbol{n}$ to a surface $\boldsymbol{S} : \mathbb{R}^2 \to \mathbb{E}^3$ is given by the `N` operator defined in Script 5.7.1. So, by using our standard operators for vector operations between function vectors, we can give the `ThinSolid` definition of Script 12.5.16. As usual, `S3` is the predefined selector of the w coordinate of domain points, and the formal parameter `surface` is a sequence of 3 bivariate coordinate functions $\mathbb{R}^2 \to \mathbb{R}$, written using the selector functions `S1` and `S2`, for u and v, respectively.

Script 12.5.16 (ThinSolid operator)

```
DEF ThinSolid (surface::IsSeqOf:IsFun) =
   surface vectSum (S3 scalarVectProd N:surface)
```

Note We like to note the generality of the `ThinSolid` operator given above. It can be used to *solidify* every kind of parametric 3D surface discussed in this chapter, provided that the input to `ThinSolid` is a *triplet* of coordinate functions $\mathbb{R}^2 \to \mathbb{R}$. Even a 2D surface, namely `surf2D`, can be solidified into a triplet of coordinate functions of a solid map, by using a `PLaSM` expression like

```
ThinSolid:(surf2D AR K:0);
```

The reader should notice that the `ThinSolid` operator cannot be used to transform a 2-manifold in $\mathbb{E}^n$ into a 3-manifold in $\mathbb{E}^n$, because the normal field is defined only in 3D as the vector product of partial derivatives. A more general definition would require exterior algebra, which is beyond the scope of this book, but is left as an interesting project for the mathematically skilled reader.

Example 12.5.10 (Thin Coons' solid)
We conclude here our Coons' section by generating a thin solid slab from a 3D Coons' surface. Therefore, in Script 12.5.17, starting again from the 3D curves `Su0`, `Su1`, `S0v` and `S1v` defined in Script 12.5.14 and from the `CoonsPatch` and `Domain2D` generators given in Script 12.5.13, we first define a `solidMapping` sequence of coordinate functions of a volume map $\mathbb{R}^3 \to \mathbb{E}^3$. The thin solid *slab* of 0.5 thickness shown in Figure 12.23 is finally generated by the last script expression in the standard `PLaSM` way.

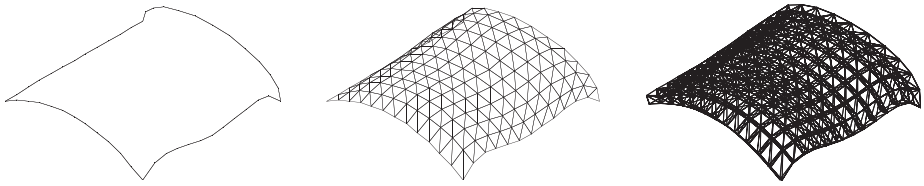

Figure 12.23 (a) Boundary curves (b) Interpolating surface (c) Thin Coons' solid generated by boundary curves

Script 12.5.17 (Coons' slab)

```
DEF solidMapping = (ThinSolid ~ CoonsPatch):<Su0,Su1,S0v,S1v>;
DEF Domain3D = Domain2D:<12,12> * Intervals:0.5:1;

MAP:solidMapping:Domain3D ;
```

12.6 Examples

Some non-trivial programming examples with parametric surfaces and solids are given in this section. In particular, we move one step further in the umbrella example of previous chapter, by generating the cloth canvas as surfaces depending on the opening angle; then we generate an helicoidal spiral volume resembling the Guggenheim Museum in New York City and a preliminary volume design of a sport building, and, finally, we model a free-form duct with cross-sections of constant area interpolating some given curves.

12.6.1 Umbrella modeling (3): tissue canvas

We continue here the step-wise refinement of the geometric modeling of the *parametric umbrella* introduced in Section 8.5.3 as a hierarchical assembly, already extended with curved rods in Section 11.5.2. A pair of cloth canvases is thus defined for the umbrella portion associated with the single instance of the `RodPair` object. The two canvases, shown in Figure 12.25a and Figure 12.25b, are defined as Coons' surfaces delimited by four Bézier curves of various degrees.

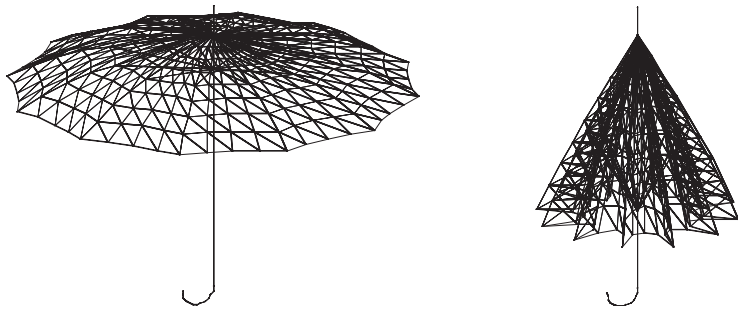

Figure 12.24 The new values of `Umbrella:<10,80>` and `Umbrella:<10,30>` after the redefinition of `RodPair`.

Before going on, our diligent reader should take another look at the previous implementation of the problem.

For `Canvas1`, two quadratic Bézier limiting curves (`ru0` and `ru1`) are generated by the same control points of `Rod11` and by their rotated instance (around the z axis); the other two curves are obtained by taking respectively the first and the last points (`p1` and `p3`) of such curves, and by computing two more points (`p12` and `p22`) on the horizontal plane. The position of such control points is a proper function of the opening angle of the umbrella.

A function `MoveToWC` is needed to reproduce the pipeline of 3D transformations (discussed in Section 6.4.2) which must be applied to the extreme points of `rod11`. Notice that within the `Umbrella` generating function the same transformations are applied to polyhedra by using primitive `PLaSM` operators.

The second canvas just differs from the first one in the choice of the limiting curves. In particular, the functions `ru0` and `ru1` are now given as linear Bézier with two control points, and `r0v` is given as a Bézier function of degree 0 depending on a single control point.

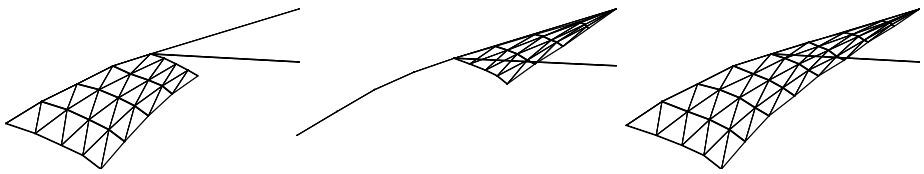

Figure 12.25 (a) First canvas (b) Second canvas (c) Complex generated by `RodPair:<10,80>`

Script 12.6.1

```
DEF Canvas1 (h, alpha::Isreal) =
   MAP:(CoonsPatch:<ru0,ru1,r0v,r1v>):(Domain2D:<4,4>)
WHERE
   handles1 = MoveToWC:<<0,0,0>,<0,0,AB/2>,<AB/2 * sin:beta,0,AB>>,
   MoveToWC = AA:(Tz:h ~ Ry:(-:alphaRad) ~ Tz:(-:AB) ~ Sz:-1),
   handles2 = AA:(Rz:(PI/6)):handles1,
   ru0 = Bezier:handles1,
   ru1 = Bezier:handles2,
   r0v = Bezier:<p1, p12, Rz:(PI/6):p1> ~ [S2],
   r1v = Bezier:<p2, p22, Rz:(PI/6):p2> ~ [S2],
   p1 = FIRST:handles1,
   p2 = LAST:handles1,
   p12 = Rz:(PI/12):<S1:p1 - AB/2*COS:alphaRad, S2:p1, S3:p1>,
   p22 = Rz:(PI/12):<S1:p2 - AB*COS:alphaRad, S2:p2, S3:p2>,
   beta = -:alphaRad/4,
   alphaRad = alpha*PI/180,
   AB = h*4/10
END;
```

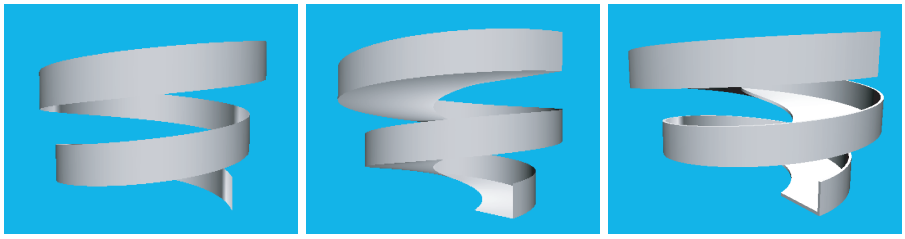

Figure 12.26 (a) Helicoidal cylinder with vertical rules (b) Double helicoidal surface defined by two helix curves (c) Thin solid slabs generated by two surfaces

The function `RodPair` is thus defined again, by inserting in it a proper pair of canvases, shown in Figure 12.25. The new configurations of the whole umbrella for different opening angles are shown in Figure 12.24.

12.6.2 Helicoidal spiral volume

In this example we generate some curves and surfaces by using parametric affine transformations, and solids by linear transfinite interpolation of surfaces. Our goal is to make the reader acquainted with this kind of generative methods, which may offer useful shape modeling tools. In Figure 12.26 we show both intermediate and final results of our modeling session, i.e. a pair of spiral helix surfaces and solids with both horizontal and vertical rule directions.

In Figure 12.27 the design approach we are going to discuss is stepwise illustrated. The equations used and their `PLaSM` implementation are given below.

Script 12.6.2

```
DEF Canvas2 (h, alpha::Isreal) =
   MAP:(CoonsPatch:<ru0,ru1,r0v,r1v>):(Domain2D:<4,4>)
WHERE
   handles1 = MoveToWC:<<0,0,0>,<0,0,AB>>,
   MoveToWC = AA:(Tz:h ~ Ry:(-:alphaRad) ~ Sz:-1),
   handles2 = AA:(Rz:(PI/6)):handles1,
   ru0 = Bezier:handles1,
   ru1 = Bezier:handles2,
   r0v = Bezier:<p1> ~ [S2],
   r1v = Bezier:<p2, p22, Rz:(PI/6):p2> ~ [S2],
   p1 = FIRST:handles1,
   p2 = LAST:handles1,
   p22 = Rz:(PI/12):<S1:p2 - AB/2*COS:alphaRad, S2:p2, S3:p2>,
   beta = -:alphaRad/4,
   alphaRad = alpha*PI/180,
   AB = h*4/10
END;
```

Script 12.6.3

```
DEF RodPair (h, alpha::Isreal) = STRUCT:<
   Canvas1:<h, alpha>,
   Canvas2:<h, alpha>,
   T:3:h, R:<3,1>:(-:alphaRad), Rod10,
   STRUCT:<T:3:(-:AB), Rod11 >,
   T:3:(-:AB), R:<3,1>:(2*alphaRad), Rod2 >
WHERE
   alphaRad = alpha*PI/180,
   Rod10 = S:3:-1:(Rod:(AB)),
   Rod11 = S:3:-1:(RodCurve:<AB,-:alphaRad/4,4>),
   Rod2 = S:3:-1:(Rod:AB),
   AB = h*4/10
END;
```

Planar spiral curve

This *curve* may be obtained by rotating the unit vector $\boldsymbol{e}_1$ with parametric angle u, and then by uniform scaling the $\mathbb{E}^2$ plane with $R+\frac{d}{2\pi}u$ scaling parameter, where R is the starting *radius* and d is the *distance* between the two points reached by the curve before and after a 2π turn. Hence, we can write:

$$\begin{aligned}\boldsymbol{c}(u) &= \begin{pmatrix} R+\frac{d}{2\pi}u & 0 \\ 0 & R+\frac{d}{2\pi}u \end{pmatrix}\begin{pmatrix} \cos u & -\sin u \\ \sin u & \cos u \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad 0 \le u \le 2\pi n \\ &= \begin{pmatrix} \left(R+\frac{d}{2\pi}u\right)\cos u & \left(R+\frac{d}{2\pi}u\right)\sin u \end{pmatrix}\end{aligned}$$

and, using the variable-free functional notation:

$$\boldsymbol{c} = \begin{pmatrix} (\underline{R}+\underline{\left(\frac{d}{2\pi}\right)}\mathrm{id})\cos & (\underline{R}+\underline{\left(\frac{d}{2\pi}\right)}\mathrm{id})\sin \end{pmatrix}$$

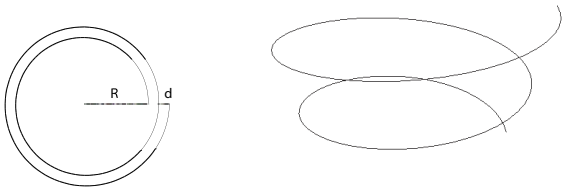

Figure 12.27 (a) Planar spiral (b) Helicoidal spiral

Implementation We can directly rewrite in Script 12.6.4 the previous notation, using the `radius` symbol for R. The curve shown in Figure 12.27a is produced with *Adobe Illustrator*© by importing the SVG file produced by the script. As always, the `S1` selector is used rather than the `ID` function, because the coordinate functions must be applied to the domain decomposition vertices, that are represented as sequences.

Script 12.6.4 (Planar spiral)

```
DEF spiral (radius,d::isRealPos) = < rad * COS ~ S1, rad * SIN ~ S1 >
WHERE
   rad = K:radius + K:(d/(2*PI)) * S1
END;

DEF out = MAP:(spiral:<3,0.5>):(intervals:(2*PI*2):200);
svg:out:400:'out.svg'
```

Helicoidal spiral

By adding a parametric translation in the z direction, where h is the *pitch*, i.e. the z difference after one helix turn, we get the *helicoidal spiral curve* in $\mathbb{E}^3$, whose parametric equation as a function of the turning angle u is:

$$\boldsymbol{c}(u) = \begin{pmatrix} \left(R + \frac{d}{2\pi}u\right)\cos u \\ \left(R + \frac{d}{2\pi}u\right)\sin u \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ \frac{h}{2\pi}u \end{pmatrix}$$

And, in variable-free notation

$$\boldsymbol{c} = \begin{pmatrix} (\underline{R} + \underline{(\tfrac{d}{2\pi})}\mathrm{id})\cos & (\underline{R} + \underline{(\tfrac{d}{2\pi})}\mathrm{id})\sin & \underline{(\tfrac{h}{2\pi})}\mathrm{id} \end{pmatrix}^T$$

Implementation In Script 12.6.5 the `spiralhelix` generator is given, as a function depending on `radius`, `d` and `h`, and producing a sequence of three coordinate functions. The produced curve is shown in Figure 12.27b. Notice that the `*` operator has a higher precedence than `+`.

Script 12.6.5 (Helicoidal spiral (1))

```
DEF spiralhelix (radius,d,h::isReal) = < rad * COS~S1, rad * SIN~S1, z >
WHERE
   rad = K:radius + K:(d/(2*PI)) * S1,
   z = K:(h/(2*PI)) * S1
END;

DEF out = MAP:(spiralhelix:<3,1,2>):(intervals:(4*PI):200);
VRML:out:'out.wrl'
```

Helicoidal cylinder

The *helicoidal spiral cylinder* surface shown in Figure 12.26a is obtained by bivariate transfinite interpolation of two curves. Such a Bézier surface of the *first degree* in the v parameter is generated by two control (curve) maps. In particular, we execute the following steps:

1. the sequence of coordinate functions of a $\boldsymbol{c}_1$ curve is generated;
2. a second curve $\boldsymbol{c}_2 = \boldsymbol{c}_1 + (\underline{0}, \underline{0}, \underline{h})^T$ is defined;
3. the coordinate functions of the surface are produced as linear transfinite Bézier combination of the coordinate functions of $\boldsymbol{c}_1$ and $\boldsymbol{c}_2$;
4. finally, the surface coordinate functions are mapped over some suitable domain decomposition.

Implementation Our helicoidal spiral cylinder is implemented in Script 12.6.6 by following the procedure discussed above. The `c1` curve is a `spiralhelix`; the `c2` curve is a translated copy of `c1`. Notice that the translation is given as a sum by a vector (sequence) of constant functions. The resulting `sup` surface is generated by the transfinite operator `Bezier:S2` given in Script 12.5.3. The reader should try some other generation method directly applicable to this case.

Script 12.6.6 (Helicoidal spiral cylinder)

```
DEF c1 = spiralhelix:<3,1,2>;
DEF c2 = c1 vectSum <K:0, K:0, K:1>;

DEF sup = Bezier:S2:< c1, c2 >;
DEF dom2D = intervals:(4*PI):100 * intervals:1:1;

DEF out = MAP:sup:dom2D;
VRML:(out CREASE (PI/2)):'out.wrl';
```

Pair of connected surfaces

A pair of helicoidal spiral surfaces, connected on a common curve, with horizontal and vertical rules, respectively, is produced by Script 12.6.7 using three curves `c0`, `c1` and `c2`. For this purpose the `c0` curve is generated with smaller radius and `d` parameter equal to zero. The last expression of the script produces the model shown

in Figure 12.26b.

Script 12.6.7 (Pair of spiral surfaces)

```
DEF c0 = spiralhelix:<1,0,2>;
DEF c1 = spiralhelix:<3,1,2>;
DEF c2 = c1 vectSum <K:0, K:0, K:1>;

DEF surf0 = Bezier:S2:< c0, c1 >;
DEF surf1 = Bezier:S2:< c1, c2 >;
(STRUCT ~ [MAP:surf0, MAP:surf1]):dom2D ;
```

Double solid cylinder slab

A solid model of the two spiral slabs shown in Figure 12.26c is finally generated by assembling two helicoidal 3-manifolds, respectively generated by `solid0` and `solid1` $\mathbb{R}^3 \rightarrow \mathbb{E}^3$ maps, defined by transfinite Bézier blending of two surface maps in Script 12.6.8. Notice that `dom3D` is a suitable partition of the 3D interval $[0,1]^3$. Notice also that the objects named `c1`, `c2`, `surf0` and `surf1` are defined in Script 12.6.7. The `spiralhelix` generator is given in Script 12.6.5.

Script 12.6.8 (Doppia superficie elicoidale)

```
DEF c3 = spiralhelix:<3.15,1,2>;
DEF c4 = c3 vectSum <K:0, K:0, K:1>;

DEF surf01 = surf0 vectSum <K:0, k:0, K:0.15>;
DEF surf11 = Bezier:S2:< c3, c4 >;

DEF solid0 = bezier:S3:< surf0, surf01 >;
DEF solid1 = bezier:S3:< surf1, surf11 >;
DEF dom3D = dom2D * intervals:1:1;

DEF out = (STRUCT ~ [MAP: solid0, MAP: solid1]): dom3D ;
VRML:(out CREASE (PI/5)):'prove/out5.wrl';
```

Suggestion The reader is warmly invited to modify the scripts given above, in order to generate a lateral spiral wall belonging to a reversed conical surface, resembling the well-known design of the Guggenheim Museum in New York City. Finally, make the model solid by using again the transfinite Bézier approach.

12.6.3 Roof design for a sports building

Here we introduce a study project, shown in Figure 12.28, concerning the preliminary design of the roof of a building for indoor sport meetings. We suggest generating the building cover by using the same approach discussed in the previous section. The more interesting part of the project concerns the modeling of the lateral columns, linked to

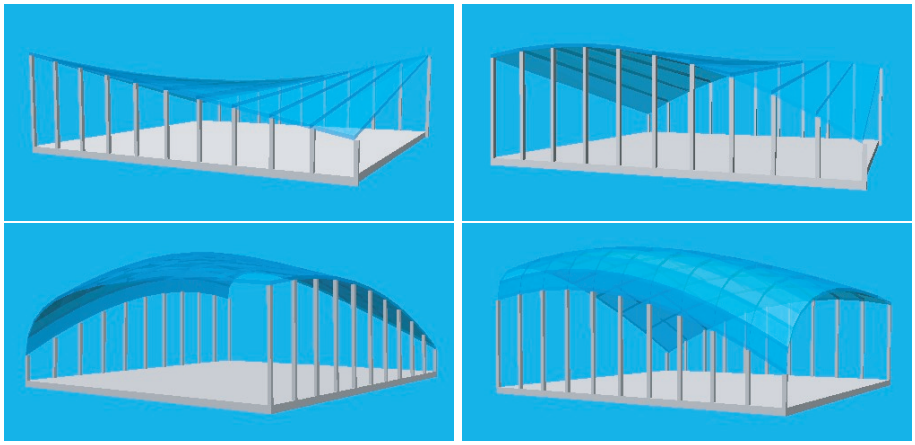

Figure 12.28 Some preliminary designs with roof covers of different geometry

the rhythm of the roof partitioning. We suggest using the curve maps interpolated to crate one of the roof surfaces as the generators of the data that produce and locate the columns, according to Figure 12.28.

No coding is given in this case. The reader is invited to reuse as much as possible of the implementation of the previous section. Notice that the authors' coding of this building is completely parametrized with respect to the degrees of the two roof directions and to the number of lateral columns. The roof's transparency effect can be obtained by loading the `colors` library, and by some statements like

```
DEF mycolor = RGBCOLOR:< 0.2,0.6,1 >;
DEF GLASS = BASEMATERIAL:< mycolor,mycolor,0.2,BLACK,0.2,0.6 >;
   roof MATERIAL GLASS;
```

where `roof` is a `PLaSM` object of polyhedral complex type.

12.6.4 Constrained connection volume

We discuss in this section the geometric design of curved ducts with a variable cross-section by transfinite interpolation with integro-differential constraints. This approach is based on piecewise multivariate transfinite interpolation of assigned cross-sections, via combination of section-generating functions with univariate cubic Hermite's polynomials. The volume mapping thus produced is composed of a local section scaling extracted from a one-parameter family of affine transformations, where the diagonal coefficients depend on the ratio between the areas of the starting and current sections. An appropriately chosen point sampling of the duct generated by the composition of volume mapping and section scaling may be used to generate the cell decomposition of the duct volume with tetrahedral elements. Such elements are used for numerical simulation of a fluid-dynamics problem.

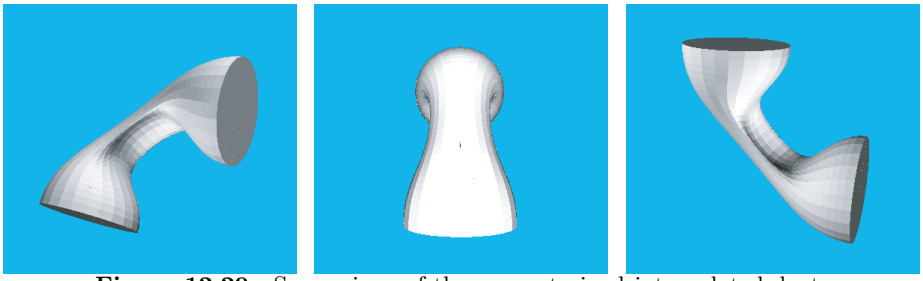

Figure 12.29 Some views of the unconstrained interpolated duct.

Problem statement

Let two smooth parametric surfaces $\boldsymbol{S}_0$ and $\boldsymbol{S}_1$ be given, with

$$\boldsymbol{S}_0, \boldsymbol{S}_1 : [0,1]^2 \to \mathbb{E}^3.$$

Let also two fields $\boldsymbol{N}_0$ and $\boldsymbol{N}_1$ of vectors normal to the surfaces $\boldsymbol{S}_0$ and $\boldsymbol{S}_1$ be assigned, with:

$$\begin{array}{lcll} \boldsymbol{N}_0 & : & [0,1]^2 \to \mathbb{R}^3 & : \quad \boldsymbol{N}_0(u,v) = h(\partial_u \boldsymbol{S}_0(u,v) \times \partial_v \boldsymbol{S}_0(u,v)), \\ \boldsymbol{N}_1 & : & [0,1]^2 \to \mathbb{R}^3 & : \quad \boldsymbol{N}_1(u,v) = h(\partial_u \boldsymbol{S}_1(u,v) \times \partial_v \boldsymbol{S}_1(u,v)), \end{array}$$

with $h \in \mathbb{R}$.

Our goal is to generate the solid obtained by cubic Hermite's interpolation of surfaces $\boldsymbol{S}_0$ and $\boldsymbol{S}_1$ with normal vector fields given by $\boldsymbol{N}_0, \boldsymbol{N}_1$. In other words, we want to generate the vector function $\boldsymbol{V}$, depending on a triplet of real parameters (u,v,w), and defined as:

$$\boldsymbol{V} : [0,1]^3 \to \mathbb{E}^3, \qquad \text{such that:}$$

$$\begin{array}{ll} \boldsymbol{V}(u,v,0) = \boldsymbol{S}_0(u,v), & \boldsymbol{V}(u,v,1) = \boldsymbol{S}_1(u,v), \\ \partial_w \boldsymbol{V}(u,v,0) = \boldsymbol{N}_0(u,v), & \partial_w \boldsymbol{V}(u,v,1) = \boldsymbol{N}_1(u,v), \end{array}$$

under the constraint that the area of given "cross-sections" (for w fixed) be constant and equal to the area of initial section:

$$\text{Area}(w) = \int_{\boldsymbol{V}(U \times \{w\})} dS = \int_{\boldsymbol{S}_0(U)} dS = \text{Area}(0), \qquad \text{for each } w \in [0,1].$$

where $U = [0,1]^2$. In the following we will refer to the function $\boldsymbol{V}$ above as *volume map*.

Approach Of course, the more difficult part of the problem is given by the constraint of constant area for each cross-section of the solid generated by the volume map $\boldsymbol{V}$. The volume map without such constraint would be constructed in a natural way as a cubic Hermite's transfinite blending of maps $\boldsymbol{S}_0(u,v)$, $\boldsymbol{S}_1(u,v)$, $\boldsymbol{N}_0(u,v)$ and $\boldsymbol{N}_1(u,v)$,

according to the approach described in [BCP00]. The solution described here utilizes methods of transfinite blending, i.e., of interpolation in functional spaces.

The basic *volume map* is thus given by:

$$\boldsymbol{V}(u,v,w) = \begin{pmatrix} h_0^3(w) & h_1^3(w) & \dots & h_3^3(w) \end{pmatrix} \begin{pmatrix} \boldsymbol{S}_0(u,v) \\ \boldsymbol{S}_1(u,v) \\ \boldsymbol{N}_0(u,v) \\ \boldsymbol{N}_1(u,v) \end{pmatrix}$$

with

$$\boldsymbol{S}_0, \boldsymbol{S}_1, \boldsymbol{N}_0, \boldsymbol{N}_1 \in \mathbb{P}_2^n \subset \{\mathbb{R}^2 \to \mathbb{R}^3\}$$

$$\boldsymbol{V} \in \mathbb{P}_3^n \subset \{\mathbb{R}^3 \to \mathbb{R}^3\}$$

Actually, the `PLaSM` implementation is a mapping of this kind:

$$\boldsymbol{M}(u_1,\dots,u_d) = \begin{pmatrix} h_0^3(u_d) & h_1^3(u_d) & h_2^3(u_d) & h_3^3(u_d) \end{pmatrix} \begin{pmatrix} \boldsymbol{S}_0(u_1,\dots,u_{d-1}) \\ \boldsymbol{S}_1(u_1,\dots,u_{d-1}) \\ \boldsymbol{N}_0(u_1,\dots,u_{d-1}) \\ \boldsymbol{N}_1(u_1,\dots,u_{d-1}) \end{pmatrix}$$

with

$$\boldsymbol{S}_0, \boldsymbol{S}_1, \boldsymbol{N}_0, \boldsymbol{N}_1 \in \mathbb{P}_{d-1}^n \subset \{\mathbb{R}^{d-1} \to \mathbb{R}^q\}, \qquad d-1 \le q$$

$$\boldsymbol{M} \in \mathbb{P}_d^n \subset \{\mathbb{R}^d \to \mathbb{R}^q\}, \qquad d \le q$$

so it works in the general case of d-variate manifolds.

Constrained Volume Map

In our approach the duct internal volume is generated by two surface interpolation steps, using three given surfaces. The desired volume map is thus obtained by the union of a first solid map interpolating the base section surface with a middle section surface, and a second solid map interpolating the middle section surface with the rotated and translated base section surface (see Figures 12.30b and 12.30c). An example of the final constrained duct interior is illustrated in Figures 12.32a, 12.32b and 12.32c, showing it from different points of view.

Constant-area constraint

In the first phase of computation an initial sequence of interpolated sections is generated. Each section corresponds to one of the w values of a user-specified uniform discretization. Successively, a proper affine transformation is applied to each section to satisfy the constant cross-section constraint.

First we get the solid $\boldsymbol{V}([0,1]^3)$ generated by the basic *volume map* applied to the standard 3-cube, to obtain an unconstrained duct. Then a family

$$\boldsymbol{Z} : [0,1] \to \mathrm{aff}\{\mathbb{E}^3\} : w \mapsto \boldsymbol{Z}(w)$$

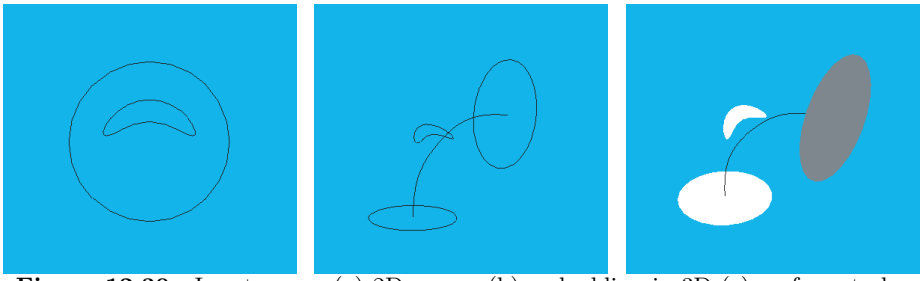

Figure 12.30 Input maps: (a) 2D curves (b) embedding in 3D (c) surfaces to be interpolated

of affine transformations, depending on a parameter $w \in [0,1]$, is properly applied to each cross-section. Each $\boldsymbol{Z}(w)$ is an *uniform dilatation* in the x, y directions of a local frame $(\boldsymbol{e}_x(w), \boldsymbol{e}_y(w), \boldsymbol{e}_z(w))$, with

$$\begin{aligned}
\boldsymbol{e}_x(w) &= \frac{\boldsymbol{q}_x(w)}{\|\boldsymbol{q}_x(w)\|}, & \boldsymbol{q}_x(w) &= \boldsymbol{V}(\epsilon,0,w) - \boldsymbol{V}(0,0,w),\\
\boldsymbol{e}_y(w) &= \frac{\boldsymbol{q}_y(w)}{\|\boldsymbol{q}_y(w)\|}, & \boldsymbol{q}_y(w) &= \boldsymbol{V}(0,\epsilon,w) - \boldsymbol{V}(0,0,w),\\
\boldsymbol{e}_z(w) &= \frac{\boldsymbol{q}_z(w)}{\|\boldsymbol{q}_z(w)\|}, & \boldsymbol{q}_z(w) &= \boldsymbol{V}(0,0,\epsilon+w) - \boldsymbol{V}(0,0,w),
\end{aligned}$$

and $\epsilon \to 0$. Notice that the local frame $\{\boldsymbol{e}_x(w), \boldsymbol{e}_y(w), \boldsymbol{e}_z(w)\}$ is extracted from the *tangent 3-manifold* to $\boldsymbol{V}[0,1]^3$ at $\boldsymbol{V}(0,0,w)$.

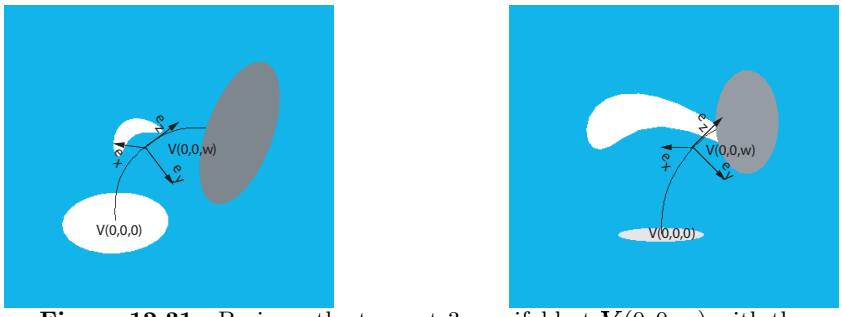

Figure 12.31 Basis on the tangent 3-manifold at $\boldsymbol{V}(0,0,w)$ with the corresponding cross-sections $B(w)$ and $(\boldsymbol{Z}(w)(B(w))$.

Each $\boldsymbol{Z}(w)$, $w \in [0,1]$, is applied to the point set $B(w) = \boldsymbol{V}([0,1]^2 \times \{w\})$. See Figures 12.31a and 12.31b to clarify this point.

The family of maps $\boldsymbol{Z}(w)$ is defined by composition of elementary affine transformations depending on the parameter w. In particular, as usual in graphics, translations $\boldsymbol{T}(\boldsymbol{t}(w))$ and inverse $\boldsymbol{T}(-\boldsymbol{t}(w))$, rotations $\boldsymbol{R}(w)$ and inverse $\boldsymbol{R}^T(w)$, and a scaling $\boldsymbol{S}(s(w), s(w), 1)$ are composed together:

$$\boldsymbol{Z}(w) = \boldsymbol{T}(-\boldsymbol{t}(w)) \circ \boldsymbol{R}^T(w) \circ \boldsymbol{S}(s(w), s(w), 1) \circ \boldsymbol{R}(w) \circ \boldsymbol{T}(\boldsymbol{t}(w))$$

with

$$\boldsymbol{R}(w) = \begin{pmatrix} \boldsymbol{e}_x(w) & \boldsymbol{e}_y(w) & \boldsymbol{e}_z(w) \end{pmatrix}^{-1} = \begin{pmatrix} \boldsymbol{e}_x(w)^T \\ \boldsymbol{e}_y(w)^T \\ \boldsymbol{e}_z(w)^T \end{pmatrix}$$

$$\boldsymbol{t}(w) = \boldsymbol{V}(0,0,0) - \boldsymbol{V}(0,0,w),$$

and where

$$\boldsymbol{S}(w) = \begin{pmatrix} s(w) & 0 & 0 \\ 0 & s(w) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

with

$$s(w) = \sqrt{\frac{\text{Area}(0)}{\text{Area}(w)}} = \sqrt{\frac{\int \boldsymbol{S}_{0([0,1]^2)}\, dS}{\int \boldsymbol{V}_{([0,1]^2 \times \{w\})}\, dS}}$$

The surface integral is computed using a general polynomial-integrating algorithm described in [Ber91]. The algorithm allows an efficient computation of integrals of polynomial functions over polyhedral domains of any dimension. This is easily computed in `PLaSM` by a single primitive

`INTEGRAL:pol_complex:<`$\mathtt{i}_1$`,`$\mathtt{i}_2$`,...,`$\mathtt{i}_d$`>`

which returns the value of the domain integral of the monomial

$$I^{i_1 \cdots i_d}_{x_1 \cdots x_d} = \int_{\texttt{pol\_complex}} x_1^{i_1}\, x_2^{i_2}\, \ldots, x_d^{i_d}\, dV.$$

The domain may not have a full dimensionality, e.g. a piecewise-linear curve or surface in $\mathbb{R}^3$. In particular, when the expression is

`INTEGRAL:pol_complex:0`

the volume of the input complex is computed. When the input is the piecewise linear approximation of a curve or surface its length or surface area are computed.

Maps composition

The *constrained volume map* $\boldsymbol{V}^*$ is now obtained by composition of the *volume map* $\boldsymbol{V}$ previously discussed with a family of affine transformations depending on one parameter. In particular, we have that

$$\boldsymbol{V}^* : [0,1]^3 \to \mathbb{R}^3$$

is easily defined as:

$$\boldsymbol{V}^*(u,v,w) = (\boldsymbol{Z}(w) \circ \boldsymbol{V})(u,v,w)$$

with $\boldsymbol{V}$ cubic transfinite Hermite's interpolation of input maps, and $\boldsymbol{Z} : [0,1] \to \text{aff}\{\mathbb{E}^3\}$, the family of affine transformations previously given.

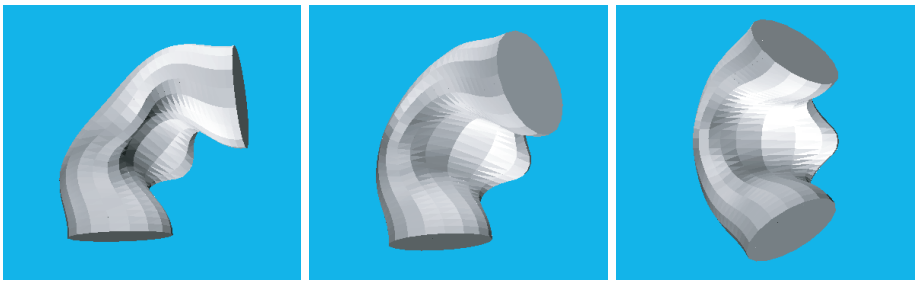

Figure 12.32 Some views of the final constrained duct

Example

We aim to generate a duct interpolating two extreme circular sections lying on two orthogonal planes, and passing through any intermediate cross-section, defined by a closed Bézier curve. In this approach, we generate independently two duct segments by interpolating from both the extreme sections to the intermediate one. First, we give the plane curves defining the sections, then we build the 3D surfaces corresponding to the section interiors, and later we interpolate the section maps with assigned normal vector fields by using cubic Hermite interpolants. The pipeline of affine transformations that satisfy the constant cross-section area constraint is finally composed with the volume map.

Note We have discussed a programming approach to the modeling of free-form ducts with cross-sections of constant area, based on the composition of transfinite volume maps with section scaling aiming to satisfy such integral constraint. The details of the implementation would require a quantity of code that is not reasonable to insert in a book section. Anyway, the authors found it interesting that the complete implementation amounts to three pages of `PLaSM` code, and that it was developed and tested in four days.

Programming approach to features The geometric modeling of ducts with variable cross-section allowed us to evaluate the use of `PLaSM` in generating complex form features by a fully parametrized functional programming approach. The solid duct generation here described can in fact be completely parametrized with respect to number, position and orientation of the given key-sections, and even to their shape. Furthermore, this transfinite solution can also generate an optimal decomposition of the duct interior with tetrahedral elements. We like to emphasize that our geometric programming not only allows definition of geometric objects in a fully parametrized way, but also definition of compact new methods for geometric shape generation, even when subject to constraints of great mathematical complexity.