Geometric Modelling in Functional Style

Jerzy Karczmarczuk

University of Caen, France

Abstract. Modelling of 3D objects is a venerable subdomain of the applied geometry, too often coded in a crude way, with elegance sacrificed for the sake of efficiency. We propose to exploit the modern lazy functional programming to generate 3D surfaces, insisting upon the genericity of the underlying mathematics. We show how to generate *easily* the generalized sweeps thanks to the algorithmic differentiation procedures. The interest of this paper is mainly methodologic, but all solutions are practical, although they are extremely short. We show a simple but decent technique of parametrizing curves through their chord-length, using lazy series development. The relation between functional programming methods and the computer graphics world is concisely reviewed.

1 Introduction

Being well adapted to the manipulation of abstractions, functional languages bridge the gap between the computer codes, and the realm of formal mathematical objects. This gap is still excessive, and may be harmful. Such is the situation in the field of image synthesis and modelling, where for efficiency programs use several optimization tricks not always easy to explain, and where the portable representations of 3D objects are often of very low level (B-representation: lists of vertices which form polygonal surfaces, e.g. the DXF, OOGL or NFF formats.)

On the other hand, a high level, linguistic description of 3D objects is not effective. We might write `cylinder` with some parameters in, say, VRML, but this is just a *symbolic* representation, which will be parsed, put into the correspondence with an internal object, transformed, converted into a mesh or an equation adapted for a ray tracer, and then rendered. The union or intersection of volumes within the CSG formalism are also symbolic descriptions. Again, for teaching purposes this is too superficial, as it doesn't say anything about how to make a modeller or a renderer, the distance between the descriptive entities and their semantics is too long.

We try here to advocate a purely functional approach for representing rendered objects and scenes. The surfaces will be represented directly by their equations, implicit: $F(\boldsymbol{x}) \equiv F(x,y,z) = 0$, or parametric: $\boldsymbol{x} = \boldsymbol{x}(u,v)$, depending on the rendering schema. We shall heavily use the higher-order functions to represent extrusions and other generalized sweeps, and we show how the usage of the algorithmic differentiation techniques (see [1, 2], and references therein, mainly [3]) might simplify *enormously* the manipulation of tangents to trajectories, normals to surfaces, deformations, etc. We present a small, pedagogic graphic modelling package implemented in Concurrent Clean [4]. Our objectif is to show how to exploit laziness, higher-order functions, polymorphism, etc., and the possibility to code geometric objects in a *generic* way, at a high level of abstraction, and independently of the concrete coordinate system.

We are interested more in the *generation* of the geometrical objects than in their static *representation*. This is far from being a first attempt to apply the functional methodology in the field of graphics and modelling. P. Henderson in [5] gave a functional description of pictures and their generation, and in [6] offered an elegant functional approach to 2-dimensional recursive geometric objects. In 1987 people noticed [7] the applicability of functional methods to hierarchic decomposition of graphical objects, and in 1990 D. Sinclair [8] has shown how the functional paradigm facilitates the construction of the CSG trees. Slightly later a more comprehensive paper [10] on the subject has been written, and followed by others, e. g. [11]. See also more recent papers on the functional approach to animation, e. g. [12], or [13]. A slightly similar philosophy to ours, although using very different techniques, has been presented in [14].

1.1 Example

The Fig. 1 shows a visually simple, but structurally intricate deformation of a vase constructed as a parametric revolution surface. If $\boldsymbol{v}(s)$ is a parametric curve representing a vertical path along the surface, the vase at the left is given by $\boldsymbol{x}(\phi, s) = \mathbf{R}(\phi)\boldsymbol{v}(s)$, where $\mathbf{R}$ is the rotation operator parametrized by the angle ϕ. Any

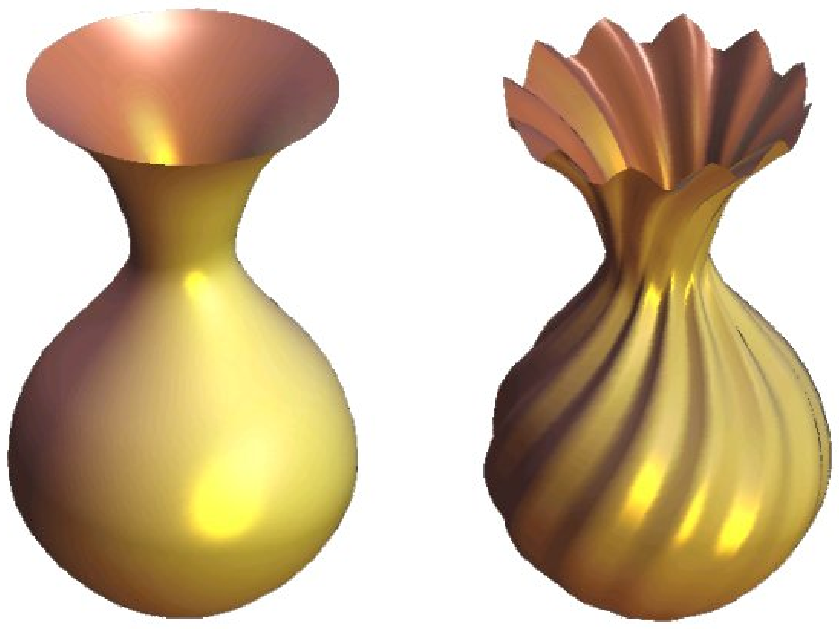

Fig. 1. A Revolution Surface and its Distortion

classical modeller will apply a similar formula, although it will *begin* by discretizing the generating curve, and constructing the mesh by the rotation of the vertices. The deformations become then difficult, because usually a *different* precision is needed. In our approach we construct from the basic surface in its functional form a different, oblique generator by cutting the surface by a non-vertical plane, and we compose the rotational sweep operator $\mathbf{R}$ with a periodic rescaling, which ondulates the surface. All this requires 5 lines of code. The discretization and rendering comes after.

2 Principal Data Types and Mathematical Entities

2.1 The basics

We assume that the reader is able to follow the basic syntax of a typical functional language. The specificities of Clean used here are rare. The first thing we will do is to declare generic 3D vectors as a new datatype: `:: Vec a = V a a a` which defines `Vec` as a record with three fields of *arbitrary* type `a`. This genericity is needed mainly because the components will be not only numeric; they may also be *lazy differential chains* explained in the next section, which represent expressions together with *all* their (scalar) derivatives needed to compute tangents to trajectories, normals, etc. Obviously, in order to define the vector algebra, the type system of Clean will require that the component type `a` admit the standard arithmetic operations (overloaded). Then the type `Vec` is an instance of the standard arithmetic operations as well. The scalar `(.*.)` and the vector `(/\)` product are defined naturally, for example

```
(/\) infix 7 :: (Vec a) (Vec a) -> Vec a | *, - a
(/\) (V x y z)  (V a b c) = V (y*c-z*b) (z*a-x*c) (x*b-y*a)
```

The first line of the declaration above is the type speification, `(/\)` is an infix operator acting on two vectors, and returning a vector. Their components *must* have the subtraction and multiplication defined. The vector manipulations inherit the overloading, which permits to lift vector operations to vector functions. We define also another overloaded operation (admittedly not very fascinating) – a multiplication of a vector by its associated scalar: `(*>)`, which exploits the constructor classes in Clean:

```
class (*>)  infixr 7 t :: a (t a) -> (t a) | * a
instance *> Vec
 where (*>) a (V x y z) = V (a*x) (a*y) (a*z)
```

The class declaration means that `(*>)` may be defined for *any* “container” type whose components may be multiplied. Later on we will use a similar operator for the power series. A similar division `(>/)` is defined also, so we can define the normalization. Also: the component extraction, rotations, conversion to quaternions, and whatever you can think of, including the projective (homogeneous) representation, which we shall not use in this paper. Although the reader will be hardly impressed by the 3D rotation formula, we present it here for training. Here it is, the rotation of the vector $\boldsymbol{v}$ about the axis $\boldsymbol{n}$ by the angle ϕ in a generic, coordinate-independent style

```
vrot n phi v
 # vpar = (v.*.n)*>n  // parallel projection; doesn’t change
 # vort = v - vpar    // perpendicular component; 2D rotation
 = vpar + cos phi *> vort + sin phi *> (n /\ vort)
```

The # symbol in Clean introduces local assignments.

2.2 Some surface generators

The most universal methods of obtaining parametric surfaces is by sweeping: displacing one curve called the *generator* along a specific trajectory (the director), composed eventually with other transformations. The classical extrusion $\boldsymbol{x}(s,t)$ is a translational sweep by t of $\boldsymbol{v}(s)$, where the director is a straight line defined by its direction $\boldsymbol{n}$, a normalized vector.

```
stransl fgen n s t = fgen s + t *> n    // n is the axis

// Example
gen s = V (4.0*cos s) (4.0*sin s) (0.4*sin(8.0*s))
ax = normalize (V 1.0 1.0 2.0)
surf1 = stransl gen ax
```

In order to rotate a curve about an axis, we need only to parametrize the rotation procedure, which is a one-liner. Here the trajectory is a parametrized vector.

```
srevol f n s phi = vrot n phi (f s)
// Example
gen s = V (2.0 + 0.1*cos(12.0*s)) s (2.0*s)
ax = normalize (V 0.4 0.0 1.0);  surf2 = srevol gen ax
```

The examples are shown on Fig. 2 In order to plot a parametric surface we sample it creating a grid, and then

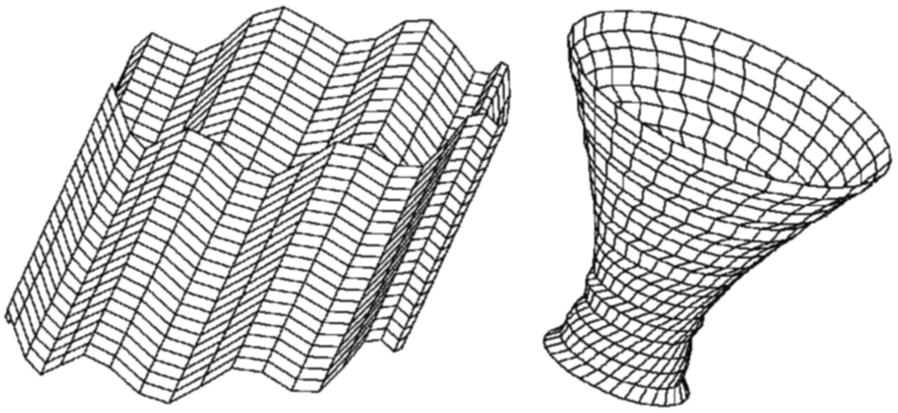

Fig. 2. Extrusion and Rotational Sweep

plot the result using an adequate projection, and some hidden-line removal techniques, Phong interpolation, and other rendering tricks which have nothing to do with modelling. A generic projection of a vector $\boldsymbol{p}$ on a plane given by the implicit equation $\boldsymbol{a} \cdot \boldsymbol{x} = d$ is the point $\boldsymbol{p_0} = \boldsymbol{p} - (\boldsymbol{a} \cdot \boldsymbol{p} - d)\boldsymbol{a}$. In order to use the Clean graphical output, the resulting points are transformed to standard Clean 2D `Vectors` by rejecting one superfluous coordinate (parallel to $\boldsymbol{a}$), choosing the planar orientation, rescaling and rounding. The wire-frame plot without the hidden-line removal is trivial. More elaborate schemas are quite elegant, but irrelevant for the picture generation, so actually we check the generators by pipelining the sampled grids to an external renderer, planning in the future to attach the OpenGL rendering procedures to the Clean (or Haskell) code. In their actual versions these languages are *not* adapted for image synthesis.

2.3 Splines

Almost all serious manipulations of a model represented by the discrete mesh force the modeller to interpolate the points using e. g. Catmull-Rom splines for the contours, NURBS for the surfaces, etc. But we can easily do that once at the beginning of the modelling. A cubic polynomial $\boldsymbol{v}(t)$ which passes through the segment of 4 points, and interpolates p_0 and p_1 for $t \in [0,1]$ is given by

```
splseg pm p0 p1 p2
 # c=0.5*>(p1+pm)-p0
 # d=(p2-p0-4*>c+pm-p1)>/6
 =(\t -> p0 + t*>(0.5*>(p1-pm)-d+t*>(c+t*>d)))
```

and the generator of the spline function spanned by $p_0, p_1, \ldots, p_n$ constructs iteratively *a list of spline segments*, returning the function which selects the appropriate one. Then it is straightforward to generate manually a nice spline representing the axial cross section of a sea-shell presented on Fig. 3, and composing the rotation with the homothetic rescaling by $\exp(a\phi)$ where ϕ is the rotation angle produces in no time the Fig. 4.

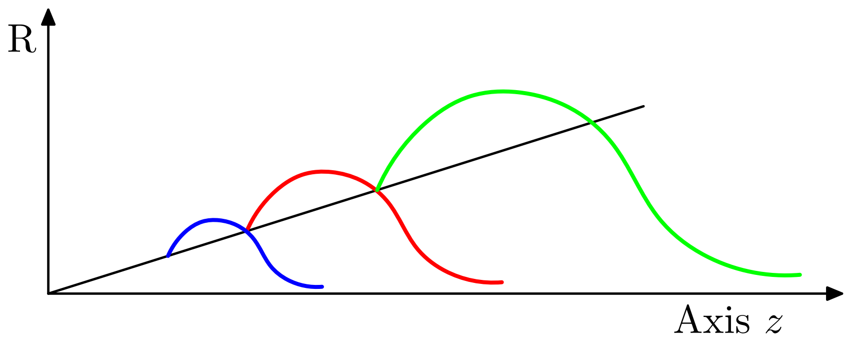

Fig. 3. Sea-shell Generator

3 Lazy Differentiation, Tangents and Tubes

3.1 What is Algorithmic Differentiation?

Although many people think that the differentiation is an *analytic* operation, whose concrete implementation requires either the numerical approximations or some symbolic formula manipulation, it is known that the issue is strictly algebraic, and everything can be done within the – augmented – local algebra of arithmetic operators.

We just sketch the one-dimensional case (trivially generalizable). We need only the existence of one additional linear operator, the **derivation** d, which acts effectively on normal expressions, and fulfills the Leibniz rule: $d(u \cdot v) = u \cdot d(v) + d(u) \cdot v$. For “normal” numbers d yields zero, they are differential constants. In order to construct a non-trivial instance of such an algebra, we introduce a new data type which generalizes numbers, but we shall never do any symbolic manipulations.

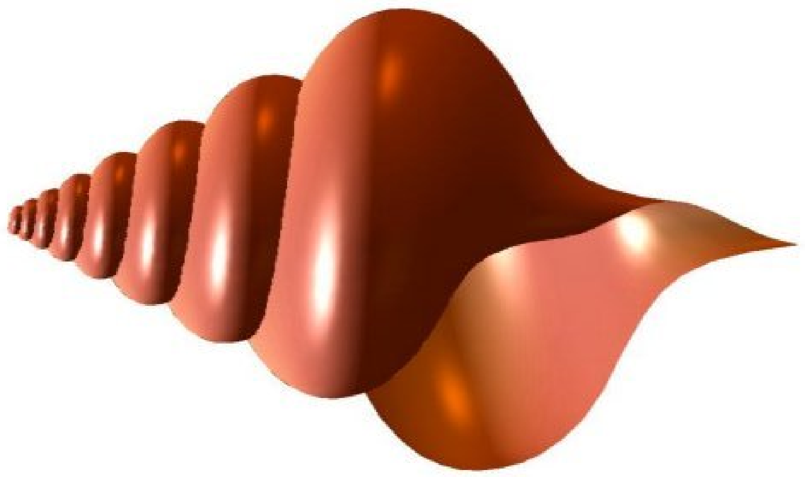

Fig. 4. Generalized Rotation

Our program will manipulate data which structurally are infinite sequences $z = [z_0, z_1, z_2, \ldots]$. The standard numbers, i. e. the explicit constants, say, c in the program are equivalent to sequences $[c, 0, 0, 0, \ldots]$ (in practice we shall use different, specific constructor, not the normal list, the type

```
:: Diff a = C a | D a (Diff a)    // a: usually a Real
```

where the tag `C` is used for constants, and the recursive case generates an infinite co-recursive sequence. We assume that the reader is acquainted with the lazy semantics which permits such construction even without the variant `(C a)`). The second and further elements of this sequence are the *extensional* derivatives of the expression. The *differential variable* x, which in a normal program is just a numeric value, here will have the form $[x, 1, 0, 0, \ldots]$ (coded by a special function as `dVar x = D x (C 1)`). This is a nameless object, a generator of the differential algebra.

Now, if we have two expressions-sequences u and v belonging to this algebra: $u = [u_0, u', u'', u^{(3)}, \ldots]$, and $v = [v_0, v', v'', v^{(3)}, \ldots]$, which we might syntactically represent as lists $u = [u_0 : u_p]$, and $v = [v_0 : v_p]$, then the following definitions hold:

$$u + v = [u_0 + v_0 : u_p + v_p], \quad \text{etc.}, \tag{1}$$
$$u \times v = [u_0 \times v_0 : u_p \times v + u \times v_p], \tag{2}$$
$$u/v = [u_0/v_0 : (u_p \times v - u \times v_p)/(v \times v)], \tag{3}$$
$$\exp(u) = w \quad \textbf{where} \quad w = [\exp(u_0) : w \times u_p], \tag{4}$$
$$\sqrt{u} = w \quad \textbf{where} \quad w = [\sqrt{u_0} : \frac{1}{2}(u_p/w)], \tag{5}$$
$$\log(u) = [\log(u_0) : u_p/u], \quad \text{etc.} \tag{6}$$

In such a way any “normal” numeric function in our program, which is finally defined through standard arithmetic operations, is easily lifted into the differential domain, which is closed. The user defines in his program a numeric function f composed of constants, of *the* variable, and of standard arithmetic operations. When this function is applied as follows: $f([\overline{x}, \overline{x}', \ldots])$, it yields $[f(\overline{x}), f', f'', \ldots]$: the value of the expression together with all its derivatives "for free". The essential part of the full package is very short, it is just the translation of the formulae above, coded within the framework of Clean type classes.

Coding $x \exp(-x^2)$ where `x = D 0.5 (C 1.0)` gives the infinite sequence: 0.3894004, 0.3894004, -1.947002, -0.3894004, 15.96542, etc. without further coding chores. The essential properties of the presented framework are the following:

- The derivatives are computed numerically, and point-wise, there are no symbolic manipulations.
- We don't compute derivatives of *functions*; every *expression* (a piece of data) drags with it, a chain of all its derivative forms with respect to the given *variable*. This is a trivial point, but sometimes delicate to explain…
- The technique is algorithmically efficient, exact (with the machine precision, as the "main" computation), and stable: no specific error propagation takes place.
- It is almost fully automatic, and easier to use than any other differentiation method. The derivation operator is trivial, it is just the tail of the sequence: `df (C _) = C 0`, `df (D _ p) = p`. The gap between the "analytic" operations and the generation of the numerical code is shortened to zero.
- In order to code it efficiently, all the expressions should be composed out of overloaded operators: if we want to multiply the lifted expressions, the basic domain must admit the multiplication and the addition. Also – in Clean – the constants should be overloaded (`fromReal...`), but in the enclosed programs we have simplified this.

3.2 Tangents and normals

We see now the interest of having polymorphic vectors. Their scalar derivatives are given by

```
instance df Vec a | df a      // We say that vectors are differentiable
where
 df (V x y z) = V (df x) (df y) (df z)
```

and the (unnormalized) tangent to a vector which is the outcome of some parametric calculus, is constructed by extracting from such a lifted vector its "main value" (i.e. the heads of the 3 differential chains for the components).

For our work we shall need some more elaborate tools, as can be seen in any book on geometric methods applied to the modelling, e. g. [15]. Of course, if we have a parametric vector function representing a trajectory, say `v s = V (cos s) (sin s) s`, the construction `g s = v (dVar s)` generates the curve $\boldsymbol{v}$ together with $\boldsymbol{v}'$, etc. In order to get the normal to the curve, and its associated curvature, it is necessary to introduce the length parametrization, as for $\boldsymbol{v}(t)$ the normalized tangent is $\boldsymbol{t} = d\boldsymbol{v}/ds$, where $ds/dt = \sqrt{(d\boldsymbol{v}/dt)^2}$. Then $\boldsymbol{n} = d\boldsymbol{t}/ds$. The following procedure computes the normalized tangent and the normal (whose length is the curvature) from a parametric vector lifted into the differential domain.

```
frenet v
 # utn = df v            // Unnormalized tangent
 # nrv = vnorm utn       // Its norm
 # tn = utn >/nrv        // Tangent
 # nrm = df tn >/ nrv    // Normal
 = (vget tn, vget nrm)
```

where `(vget v)` retrieves the main values from the differential vector domain. Of course, the third component of the Frenet frame, the sometimes useful binormal vector is the vector product of the tangent and the normal.

The presented technique is simple to apply, the user should however be careful about the type system. In Clean such a linear trajectory: `lin s = 2.0*s` cannot be lifted to the differential domain, one should write `lin s = fromReal 2.0 * s`. In Haskell all numerical constants are by default overloaded, and the programming is slightly easier.

3.3 Transport of curves along trajectories

We are ready to construct a fairly generic sweep, where the generating curve follows *any* trajectory (producing a generalized cylinder, or a "tube"). The generator gets translated along the trajectory, but it also rotates, so that its orientation keeps pace with the tangent to the director. As an extra bonus we might include in the procedure an additional arbitrary transformation undergone by the generator: it might be a screw-like rotation, or rescaling, which may produce very flexible objects, for example the awful knot shown on Fig. 5. (For those readers whose experience with knots is strictly practical: its director is a projection of a simple torus $T(\theta, \phi)$ with $\theta = (2/3)\phi$.) We have thus to establish the orientation of the generator, usually a vector orthogonal to its

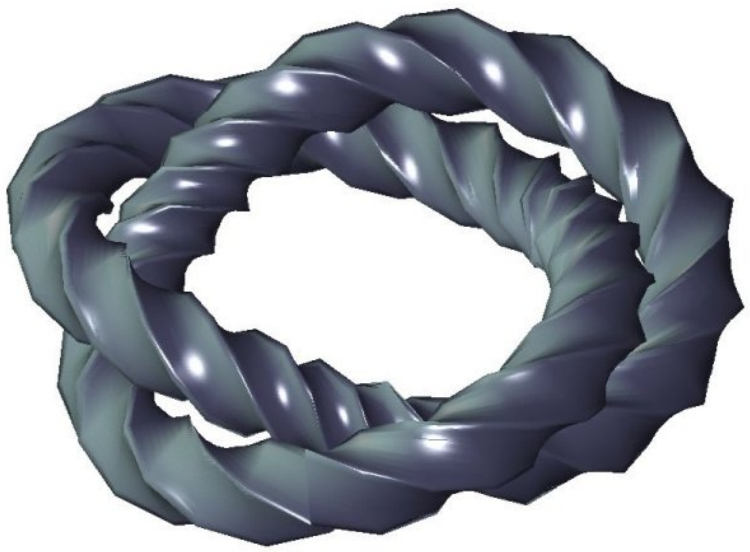

Fig. 5. Generalized Tube

plane. For an arbitrary spatial generating curve in general this is conventional, and for simplicity we add this orientation vector as an external parameter, but for planar curves it suffices to compute its binormal. We shall need only one new function which finds the rotation axis and the angle needed to align a vector $\boldsymbol{u}$ along another (normalized) vector $\boldsymbol{n}$. The rotation axis is $\boldsymbol{a} = \boldsymbol{u} \wedge \boldsymbol{n}$, the norm of this vector gives directly $|\boldsymbol{u}|\sin(\phi)$, and $\cos(\phi)$ is obtained from $\boldsymbol{u} \cdot \boldsymbol{n}/|\boldsymbol{u}|$.

Then, in order to sweep the generator $\boldsymbol{g}(s)$ (with its orientation vector $\boldsymbol{u}$) along a trajectory $\boldsymbol{h}(t)$, the trajectory is lifted to the differential domain, and for every t, $\boldsymbol{u}$ is aligned with the tangent. The generator is rotated using the alignment parameters, and the result is translated by $\boldsymbol{h}(t)$, altogether less than 10 short lines of code.

The transformations can be arbitrarily composed. The parametric form lifted into the differential domain, can be easily deformed, for example by scalar bumps $f(\boldsymbol{x}(s,t))$, as the normals to the surface are there "for free", and such a warp is defined by $\boldsymbol{x}(s,t) \to \boldsymbol{x}(s,t) + \boldsymbol{n}(s,t)f(s,t)$. This requires the usage of partial derivatives, which doesn't need anything new, it suffices to lift only one of the two surface parameters, and to retrieve the appropriate derivative. The resulting furmulae are simple, but the program is a little messy. The study of deformations, as well as the differential analysis of *implicit* surfaces need a coherent multi-dimensional generalization of our lazy differential algebra. This work is in preparation.

A piece of warning seems necessary here. Transforming/composing functions is compact and easy, but might be very inefficient if a complex transformation is applied to the generators in order to generate a dense grid. The "free form" curves and surfaces are often constructed from splines or cubic patches based on a limited number of control points. In such cases it is preferable to transform the points first, and then to reconstruct the curve.

4 Chord-length reparametrization of curves

The result of an arbitrary parametrization of the director $\boldsymbol{p}(t)$ (and of the generator as well, but we want to simplify the problem) spans a non-uniform grid. A uniform sampling in t might be erroneous. The solution is to parametrize the director by its arc length s, defined by the normalization property of the tangent: $|d\boldsymbol{p}/ds| = 1$. We have

$$\frac{d\boldsymbol{p}}{ds} = \frac{d\boldsymbol{p}}{dt} \Big/ \frac{ds}{dt}\,, \quad \text{and} \quad \frac{ds}{dt} = \sqrt{\left(\frac{d\boldsymbol{p}}{dt}\right)^2}. \tag{7}$$

So, the running arc (or chord) length is given by

$$s(t) = \int^t (ds/dt)dt, \tag{8}$$

and we see that its inversion: $t = t(s)$ needed for the reparametrization of $\boldsymbol{p}$: $\boldsymbol{p}(s) \equiv \boldsymbol{p}(t(s))$ needed for the construction of a uniform grid is usually a difficult problem. There is no simple and universal solution, and many sophisticated algorithms have been proposed, together with the brute force (oversampling and numerical integration).

The arc-length parametrization is useful for many different reasons, for example:

– The control set for often used splines, Bézier curves, etc. is usually very non-uniform. If the surface is texture-mapped, and the sampling is based on this control set, the textures get severely distorted, which is seldom satisfactory.
– In order to generate a smooth, constant speed animation, the trajectory should be uniformly sampled.
– Additionnally, an adaptative *non-uniform* sampling is often needed, for example, the Δs between two neighbouring nodes should be smaller if the curvature is large; it might be a monotonous function of the absolute value of the normal.

It would be useful thus to possess a formalism to generate a general, not necessarily uniform chord parametrization. We propose a solution which is far from the "rocket science", but which is easy, does *not* sacrifice the efficiency nor the accuracy, and shows once more the marvels of the pure functional lazy coding, and extended arithmetic. If the sampling points are generated incrementally (i. e. if the distance between the bounds of the integral (8) is not too big, the first approximation for an equally-paced Δt is obviously $\Delta s \cdot (dt/ds)$ and this we get for free when computing the tangent.

If the accuracy of the linear approximation is insufficient, it makes sense to represent $s(t)$ as a power series in t, and to reverse this series, as explained in many textbooks, for example [16]. We sketch here the algebra of lazy power series which has been implemented by us in Haskell, see [17], and converted now into Clean.

We introduce a new (unbound) list-like data structure, a *formal* power series $U(x) = u_0 + u_1 x + u_2 x^2 + \ldots$, where x is a dummy variable which will be replaced by a real value during the final numerical computation. From the algebraic and structural point of view a series is another linear sequence symbolically denoted as $U = [u_0 : \overline{u}]$, but defined with another sequence constructing operator `(:>)` as

```
:: Series a = (:>) infixr 9  a (Series a)
```

Full arithmetic algebra is easily defined for our series:

```
 (+) (x:>xq) (y:>yq) = (x+y) :> (xq+yq)
```

and the same for the subtraction. The multiplication by a scalar overloads `(*>)`, and uses the overloaded `map`. The multiplication and the division of two series, $U = u_0 + x\overline{u} \equiv [u_0 : \overline{u}]$, and $V = v_0 + x\overline{v}$ are algorithmized as follows:

$$U \cdot V = (u_0 \cdot v_0) + x(u_0 \cdot \overline{v} + \overline{u} \cdot v), \tag{9}$$

$$U/V = w_0 + x\overline{w} \quad \textbf{where} \quad w_0 = u_0/v_0,\ \overline{w} = (\overline{u} - w_0 \cdot \overline{v})/v. \tag{10}$$

The term-wise differentiation (which has nothing to do with the algorithmic differentiation presented in the previous sections, and is just a data transformation), and integration, are "`zips`" with $[1, 2, ..]$ either multiplied, or divided into. The integration needs another parameter – the integration constant, which makes the algorithm lazily co-recursive, and permits several fascinating tricks, for example the computation of the series exponential: if $W = \exp(U)$, then $W' = W \cdot U'$, and $W = [w_0 = \exp(u_0) : \int W \cdot U' dx]$. All this is explained in [17], and numerous nontrivial examples are given. The reversal of the power series, i. e. the solution $t = w_1 s + w_2 s^2 + w_3 s^3 + \ldots$ of the equation $s = u_1 t + u_2 t^2 + \ldots$ is coded in 5 lines.

The arc length $s(t)$ is expressed as the power series almost immediately. When we have applied the generator function to the lifted value: `p (dVar tp)` (for $t = t_p$, one given value which is our starting point), we have *automatically* $ds/dt|_{t_p}$: $s_1, s_2, s_3 \ldots$ where s_2 is the second derivative, etc. In fact, without doing anything, we got the Taylor expansion of $s(t)$, as the series integration is trivial, with vanishing integration constant (near zero $\Delta s \propto \Delta t$ must hold). It suffices to zip-divide the sequence of derivatives by the factorials:

$$s(t) = s_1 t + \frac{1}{2} s_2 t^2 + \cdots + \frac{1}{n!} s_n + \ldots \tag{11}$$

with the boundary condition $s(0) = 0$. The value of s_1 is different from zero, otherwise there is a singularity. If s_1 is small and Δs big, and the series converges badly, this is not the fault of the algorithm, but a signal that the curvature (and/or twist etc.) of the curve is too large, and the arc-length parametrization *will* behave badly anyhow. In normal cases the convergence of the reversed series is very good, two to four terms are quite sufficient, especially if the behaviour for large Δs is improved by Padéization.

The solution proposed in [17] consists in reducing the reversal problem to the composition of series: $U(V(x))$, where $v_0 = 0$, so $V = x\overline{V}$. We have

$$U(V) = u_0 + x\overline{V}\left(u_1 + x\overline{V}(u_2 + \cdots)\right), \tag{12}$$

just an infinite, co-recursive formulation of the classical Horner scheme. Dividing (11) by s_1 reduces the equation into a form

$$s = t + u_2 t^2 + u_3 t^3 + \ldots \tag{13}$$

whose solution is $t = s + w_2 s^2 + w_3 s^3 + \ldots$. But (13) can be rewritten as $t = s - u_2 t^2 - u_3 t^3 - \ldots$. We know thus that $t = sM$, where M is given by $M(s) = 1 - sM^2 \cdot (u_2 + u_3 t + u_4 t^2 + \ldots)$ — just a composition of U and $t = sM$ multiplied by M^2. Of course it is auto-referrent, which won't prevent a lazy programmer from sleeping.

The discretization becomes more involved, it is not a `map` anymore. For given Δs the trajectory $\boldsymbol{p}(t)$ is sampled at t_0, Δt is computed from its tangent, and the process iterates at $t_0 + \Delta t$. The technique is reasonably fast, adaptable, and quite cheap from the programming effort point of view.

5 Implicit Surfaces – Short Tutorial

We witness recently an important renewal of interest in the functional implicit representation of surfaces: $F(\boldsymbol{x}) = 0$, see [18, 19]. We cannot treat this topic here, but we will enumerate the properties of this approach, which evidently need a decent functional formalism to code it *easily*. There is one delicate point — the sampling of an implicit surface, the construction of a mesh, or the polygonization is much more involved, and needs efficient random-access data structures.

One particularity of this representation is that it is fundamentally volume-, rather than surface-oriented. The region $F(\boldsymbol{x}) < 0$ may represent the interior of the object, and the value of F may be considered to be the distance from the surface (under an appropriately chosen metric). This volume-orientation makes it especially interesting for the CSG computations.

5.1 Constructions

The coding of the geometric primitives is very compact, and well adapted for the *invariant geometric* parametrization. Here we see some examples:

1. The plane with normal $\boldsymbol{A}$, distant by d from the origin: $\boldsymbol{A} \cdot \boldsymbol{x} = d$ represents an infinite half-space "under" the plane. In such a way there is a *natural* way to represent the polyhedra – through the intersection of these half-spaces.
2. The sphere with radius R at $\boldsymbol{x_0}$: $(\boldsymbol{x} - \boldsymbol{x_0})^2 = R^2$. Now, there is no point in precising the location of an object; it suffices to replace $\boldsymbol{x}$ by $\boldsymbol{x} - \boldsymbol{x_0}$ in order to translate it.
3. The cylinder with the axis $\boldsymbol{n}$ and radius ρ: $\boldsymbol{x}^2 - (\boldsymbol{n} \cdot \boldsymbol{x})^2 = \rho^2$.
4. The cone with axis $\boldsymbol{n}$, and the angle between the axis and the surface α: $(\boldsymbol{n} \cdot \boldsymbol{x})^2 = \boldsymbol{x}^2 \cos(\alpha)^2$
5. The torus with the axis $\boldsymbol{n}$, and the radii R and r : $(R^2 + \boldsymbol{x}^2 - r^2)^2 = 4R^2\left(\boldsymbol{x}^2 - (\boldsymbol{n} \cdot \boldsymbol{x})^2\right)$.

But, as we have underlined before, the real power of functions is that we can combine and transform them. The Boolean (CSG) compositions of implicit volumes are very compact, the beginning of the functional manipulations of these objects date from the sixties, and works of Rvachev (who had no interactive computing tools to make the drawings…). If F_1 and F_2 represent two objects, then $\min(F_1, F_2)$ is their union, and $\max(F_1, F_2)$ – their intersection.

If we prefer the continuous, analytic formulations, more suitable to compute the normals or other differential quantities, we may use – among many others – the following formulae:

$$F_1 \cup F_2 = \frac{1}{2}\left(F_1 + F_2 - \sqrt{F_1^2 + F_2^2}\right) \tag{14}$$

$$F_1 \cap F_2 = \frac{1}{2}\left(F_1 + F_2 + \sqrt{F_1^2 + F_2^2}\right), \tag{15}$$

but, what is very useful, we can easily introduce the blending between the two surfaces by adding to the formulae above a function which vanishes far from the intersection ($F_1 = F_2 = 0$), for example

$$d(F_1, F_2) = \frac{a_0}{1 + \frac{F_1}{a_1}^2 + \frac{F_2}{a_2}^2}, \tag{16}$$

which for two intersecting cylinders gives the Fig. 6. Many other combinations are possible, and in particular

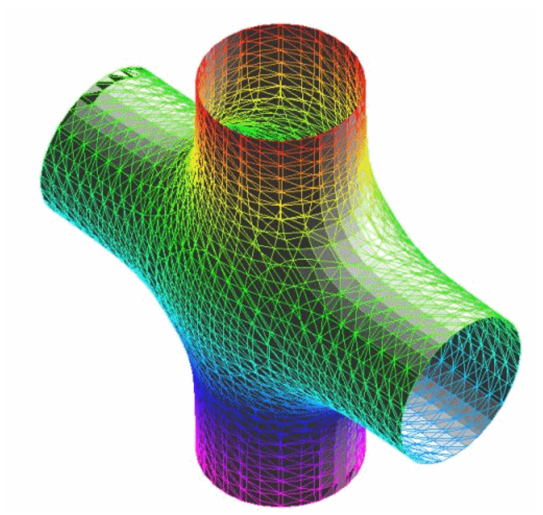

Fig. 6. Blending of Implicit Surfaces

this is the preferred representation to define "blobs" or "metaballs" – soft objects whose surface is defined by a fixed threshold of a "field" $F(\boldsymbol{x})$. Adding two such fields produces a larger blob, sometimes connected, sometimes not, and a negative blob may create a hole within another. These objects are intensely used in modelling animals, but they are rarely implemented *directly* in modellers, which do not operate upon dynamically created functions.

5.2 Transformations and Deformations

In order to displace the surface points: $\boldsymbol{x} \to \widehat{R}\boldsymbol{x}$, it suffices to render $F(\widehat{R}^{-1}\boldsymbol{x})$. This transformation may be the global translation or rotation of the entire object, but it may also depend on $\boldsymbol{x}$, introducing for example the twisting, tapering, local bumping of the surface, or its directional erosion/dilation: $F \to F(\boldsymbol{x} + \zeta\boldsymbol{N})$, where $\boldsymbol{N}$ is the normal to the surface at $\boldsymbol{x}$. The construction of this normal is of course facilitated by our differential data structures.

The CSG operations possess an unsuspected universality! In one-dimensional space $f(x) = (a - x) \cap (x - b)$ is a finite segment, and $f_1(x) \cap f_2(y, z)$ is a Cartesian product permitting to define extrusions. Passing from Cartesian to polar coordinates defines the rotational extrusions, i.e. surfaces of revolution.

The 3D morphing between two objects is just an interpolation between the two corresponding functions; we remind that for any fixed-grid representation this is a nightmare, unless the two polygon topologies are identical.

6 Conlusions

There is no difficulty in finding useful modelling packages, but quite often such systems are rigid and not too pedagogic. The data structures used are of low level, and the complexity of the procedures makes them more appropriate – eventually – for a long-term individual learning than for teaching. The necessity to invest in the high level, functional description of graphic entities has been recognized early, but the hiatus between the formal specifications and concrete representations was always rather strong, and in the world of graphics the knowledge of functional languages is limited. (Sometimes a decent theory has been elaborated only to guide the implementation of a program in “C”, see e. g. [20]). On the other hand, the mathematics needed (for example the (de)implicitization techniques, or the analytic computation of surface intersections) is sometimes so heavy, that the algebraic, symbolic packages are used very intensely. In our opinion they are abused, for example people who need the Gröbner basis computations, etc. often *do not need analytic formulae at all* (they are horrible anyway. . .), they serve just to produce expressions inserted into a numerical program. But the “C” or Fortran languages are too weak to process high-level mathematical entities, so one passes through their symbolic form. Functional languages with their higher order functions, and polymorphic type system offer better tools for designing and coding all kind of graphical entities, without losing from sight the underlying mathematical discipline. The authors of the Clean language have put already a considerable effort on implementing interactive low-level graphic entities. The area seems very promising, and the work is in progress.

Such topic as interaction is *a priori* a different subject from the modelling, but, as noted in [21], a fixed set of control points e. g. a static mesh, is often very cumbersome. If the modeller is able to represent a geometric object as an infinitely malleable surface, and inserts a control point where the user clicks its surface (which is not more difficult than the ray casting algorithm), it is *much* easier to specify, and to solve the constrained variational optimization problems resulting from the user-triggered deformations. This necessity to adapt dynamically the subdivisions of the surface grid to the actual context (local: the curvature, etc., but also external: the position of the light sources) has been recognized a long time ago by specialists on the radiosity rendering.

Finally, slightly beyond the realm of modelling, but vital for the image synthesis: the generation of textures and animation are *par excellence* application fields of functional methods. It can be seen not only in the current literature, e. g. [22], but also in concrete, practical realizations which use Scheme, as [23] and [24], much older than mentioned already [13]. But a coherent, comprehensive approach still needs a good deal of work. Ours is for the moment just a feasibility study.

References

1. Jerzy Karczmarczuk, *Functional Differentiation of Computer Programs*, Proceedings, ACM SIGPLAN International Conference on Functional Programming, (ICFP'98), Baltimore, September 1998, ACM Press, pp. 195–203.
2. Jerzy Karczmarczuk, *Lazy Differential Algebra and its Applications*, Workshop, III International Summer School on Advanced Functional Programming, Braga, Portugal, 12–18 September, 1998.
3. George F. Corliss, *Automatic Differentiation Bibliography*, published in the SIAM Proceedings of *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, ed. by G.G. Corliss and A. Griewank, (1991), but many times updated since then. Available from the *netlib* archives (`netlib@research.att.com`), or by an anonymous ftp to `ftp://boris.mscs.mu.edu/pub/corliss/Autodiff`
4. Rinus Plasmeijer, Marko van Eekelen, *Concurrent Clean – Language Report, v. 1.3*, HILT – High Level Software Tools B. V., and University of Nijmegen, (1998).
5. Peter Henderson, *Functional Programming, Application and Implementation*, Prentice-Hall, (1980).
6. Peter Henderson, em Functional Geometry, Symposium on Lisp and Functional Programming, (1982).
7. F. W. Burton, John G. Kollias, *Functional Programming and Quadtrees*, Univ. of Utah, Dept. of Computer Science, (1987).
8. Duncan C. Sinclair, *Solid Modelling in Haskell*, University of Glasgow Report, (1990).
9. Richard A. Bird, Philip L. Wadler, *Introduction to Functional Programming*, Prentice Hall, (1988).
10. Iain Checkland, Colin Runciman, *Development of a Prototype Geometric Modelling System using a Functional Language*, University of York, (1992).
11. J. R. Dave, P. M. Dew, *A Polymorphic Library for Constructive Solid Geometry*, University of Leeds Report 94.25.
12. Kavi Arya, *A Functional Animation Starter Kit*, J. Funct. Prog. **4**, (1994), pp. 1–18.
13. John Peterson, Conal Elliott, Gary Shu Ling, *Fran 1.1 Users Manual*, (1998), and papers by Conal Elliot, (*Microsoft research*).
14. John M. Snyder, James T. Kajiya, *Generative Modelling: A Symbolic System for Geometric Modelling*, Computer Graphics **26**, 2, (1992), pp. 369 – 378.
15. Christoph M. Hoffmann, *Geometric and Solid Modelling, an Introduction*, Morgan Kaufmann Pub., (1989).
16. D. E. Knuth, *The Art of Computer Programming, Seminumerical Algorithms*, vol. 2, Addison-Wesley, (1981).
17. Jerzy Karczmarczuk, *Generating Power of Lazy Semantics*, Theoretical Computer Science **187**, Elsevier, (1997), pp. 203–219.
18. *Shape Modelling and Computer Graphics with Real Functions*, `http://www.u-aizu.jp/public/www/labs/sw-sm/FrepWWW/F-rep.html`
19. A. Pasko et al., *Function Representation in Geometric Modelling: Concepts, Implementation and Applications*, in SIGGRAPH 1996 Course notes: *Implicit Surfaces for Geometric Modelling and Computer Graphics*.
20. An OBJ3 Functional Specification for Boundary Representation, ACM Conference, (1991).
21. William Welch, Andrew Witkin, *Variational Surface Modelling*, SIGGRAPH'92, ACM, Chicago, pp. 157 – 166.
22. D. Ebert, K. Musgrave, P. Peachey, K. Perlin, and S. Worley, *Texturing and Modeling: A Procedural Approach*, Academic Press Professional, (1994).
23. Spencer Kimball, Peter Mattis, *GIMP: GNU Image Manipulation Program*, `http://www.gimp.org`
24. Steve May, *AL: the Animation Language*, `http://www.cgrg.ohio-state.edu/~smay/AL/`