Teaching of Image Synthesis in Functional Style

Jerzy Karczmarczuk
Dept. d'Informatique, Université de Caen, France
karczma@info.unicaen.fr

Abstract

We have taught the 3D modelling and image synthesis for computer science students (Master level), exploiting very intensely the functional *style* of programming/scene description. Although no pure functional language was used, since we wanted to use popular programmable packages, such as POV-Ray, or the interactive modeller/renderer Blender, scriptable in Python, we succeded in showing that typical functional tools, such as higher-order functional objects, compositions and recursive combinations are useful, easy to grasp and to implement. We constructed implicit and parametric surfaces in a generic way, we have shown how to transform (deform) and blend surfaces using functional methods, and we have even found a case where the laziness, implemented through Python generators, turned to be useful.

We exploited also some functional methods for the image processing: creation of procedural textures and their transformation.

Categories and Subject Descriptors D [*1*]: 1

General Terms Algorithms, Design, Languages

Keywords Image synthesis, Functional style

1. Introduction

The teaching of functional programming *techniques* at the University level falls sometimes into a methodological trap: the FP is *separated* from the rest of the curriculum; we often teach functional languages and tools during the first two years, with plenty of pedagogical examples, and then our students discover that the algorithms and practical exercices usually implemented in other languages have not too much in common with the initial pedagogical approaches, since when the research domains of teachers are far from FP, their teaching methodologies are different as well; later on, the teaching of compilation, of artificial intelligence techniques, etc. is often far from functional methods...

We have attempted, for several years, to exploit a different strategy, more integrated with other domains we taught, where functional tools were simply ... tools, used in specific contexts, without raising the question of their suitability in general, but following the idea that if our students learnt some functional techniques/style, they should apply them.

FDPE'05 September 25, 2005, Tallinn, Estonia.

- We used functional tools in compilation (not only classical parsing tools, but the construction of functional virtual machines [1], and typing); the full course was based on Haskell.
- The teaching of the 'scientific programming'(using various languages) exploited streams for solutions of differential equations, signals and random generators, insisted on the functional presentation of the FFT algorithms (or wavelets), on functional aspects of the automatic differentiation [2], etc. We gave projects on the simulation of waveguide models of musical sound generators using lazy streams, based on our paper [3].
- Our courses on image synthesis and processing used very intensely several functional tools for the generation of parametric surfaces and curves, and also for the texture generation, etc. Those graphical applications are the subject of this presentation.

The main idea was to abandon the forcing of the usage of functional *languages*, and to *use an existing, multi-platform, popular and free software*, possessing reasonably complete computational kernels, decent interfacing layers, and extension faculties through available libraries and user scripts (or full-fledged programs), and show how to exploit functional *style* for obtaining immediate practical results.

We decided to detach the programming style from the language, and we used the scene description languages, such as POV-Ray [4], or integrated scientific packages as Scilab as 'functionally' as possible. We exploited also Python, as a stand-alone programming tool equipped with some graphical libraries, but also as the scripting language of the 3D modelling/rendering package Blender [8]. Although Python is not an "accepted functional language", writing not too exotic programs in functional style is relatively straightforward. In this context the term 'functional' means the expression-oriented programming style, with serious usage of higher-order functionals, comprehensions (and maps) instead of loops, and — when possible — the creation of functional objects (closures) reused elsewhere. We didn't insist on the 'purity' of the basic constructions. Formerly we have tried a similar, quite satisfying, projects in an even more imperative setting – the creation and transformation of VRML scenes through scripts in Java and JavaScript. But functional techniques proved once more to be simply more elegant and more universal.

Of course, the usage of functional, as declarative as possible style was not 'implicit', but manifest. Students were reminded of techniques learnt formerly through Scheme and Haskell, even if the context contained imperative elements.

This was an experience not without danger; there were at least two lines of criticism we met:

- Typical, concrete exercices in graphics/imagery are too complicated to be solved using functional approach only without severe inefficiencies. The programming style becomes eclectic, polluted by some imperative constructs, which goes against the didactic aims of the project.

- Students see that practical functional programming is feasible using various languages, which may decrease the popularity of specifically functional languages even more...

Those issues remain open, but we noticed that the productivity of students increased substantially, when they learned how to combine functional methods with object-oriented tools. As said above, we didn't forbid such imperative forms as loops for basic constructs, but we encouraged the students to hide them in various map- or fold-like constructs.

In our opinion this liberal attitude was a good decision. It is easier to appreciate some programming paradigms, when one does not feel constrained by them. People convinced that the functional *style* is nice and powerful in practical contexts, accept easier the idea that languages which insist on this style are good ones as well...

We underline once more that we were **not** interested in teaching of FP, but on using functional tools practically. In the concerned domain we insisted on close relation between functional entities and graphical objects, on combination and transformation thereof. We avoided too complicated constructs, the abstraction level used was rather moderate.

2. Using POV-Ray as a Functional Language

2.1 Simple recursion

A standard game showing the power of recursion is the creation of fractals (von Koch or IFS style). It took the students 10 minutes to convert from Scheme the program which generates the "fern" in Fig. 1.

```
#macro Fern(p0,p1,r,n)
 #if (n=0) cone{p0,r,p1,0.8*r}
 #else
  #local n1=n-1;
  #local d=p1-p0;  #local p2=p0+0.5*d;

  union{cone{p0,r,p1,0.8*r}
   Fern(p2,p2+0.4*vrotate(d,-45*z),0.9*r,n1)
   Fern(p1,p1+0.3*vrotate(d,45*z),0.8*r,n1)
   Fern(p1,p1+0.9*vrotate(d,3*z),0.8*r,n1)}
#end  #end
```

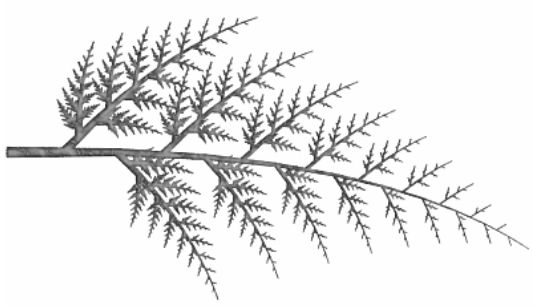

Figure 1. A "fern" in POV-Ray

Then, it was a matter of less than one hour to model a more realistic model, such as on the Fig. 2, and created by the following recursive macro:

```
#macro Obj(a,c)
 object{Rectree(i+1,l1,s1)
        rotate Rz rotate (a+20*R)*y
        translate c*l1*y}
#end
#macro Rectree(i,l,s)
 #if(i<imax)
  #local l1=l*r;
  #local s1=s*w;
  #local R=2*rand(S)-1;
  #local Rz=(30+10*R)*z;

  union{cone{<0,0,0>,s,<0,l1,0>,s1
        texture{T_Wood7}}
    Obj(40,1) Obj(130,0.8) Obj(220,0.75)
    Obj(310,0.667)}
 #else
  triangle{<0,0,0>, <1,0,1>, <-1,1,0>
      scale 22*s1 pigment{color Green}}
 #end
#end
```

Figure 2. A recursive tree

This program served also to pinpoint the essential difference between true functions and macros: the auxiliary macro `Obj` uses *literally* identifiers which are semantically internal to `Rectree` although `Obj` is external to it. Lexical substitution is not the same as the binding of local variables. Several errors produced during the extension of this program, adding more variations, improving leaves, etc., finally gave rise to a complete project whose aim was to write in *another language* the generator of trees for POV-Ray. Most students — almost obviously — chose a functional language!

The recursive macros, with local variables make POV-Ray an ***almost*** functional language. Globally, being a scene description language, POV-Ray has naturally a declarative flavour. There are anonymous functions, and conditional expressions ("C" style: `Bool ? thnxpr : elsxpr`). The recursive instances return "values", POV-Ray objects, which may be composed, transformed, decorated (e.g., textured), etc. Of course, the "composition" of 3D objects (e.g., the Constructive Solid Geometry operations: union, difference, blending etc.) is not the same as the functional composition, they are *data* combinations, but the relation between the two is pedagogically meaningful.

POV-Ray is not a full-fledged functional language, the higher-order programming is somehow clumsy, mainly because of scoping problems. Functions are not recursive, macros are, and the lexical substitution demands much attention to avoid identifier trapping errors. But a macro can assign a parameter and thus, implicitly return a function assigned to this parameter. Many functional tools

are implementable, but it is obvious that the students **must** be acquainted with the "true" functional techniques and languages first!

Here we focussed on simple recursion, and on the construction of complicated geometrical distributions, permitting the deformation of implicit surfaces, or the deformation of randomly generated configurations, like the "galaxy" in Fig. 3.

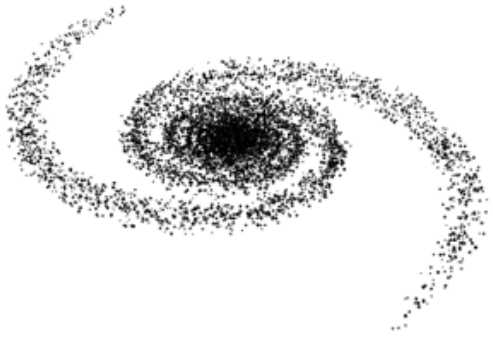

Figure 3. A "galaxy" in POV-Ray

2.2 Deformations

We did a lot more, e.g., constructing small functional L-system packages, or experimenting with various deforming/blending strategies for implicit surfaces [5] represented as equations $F(x, y, z) = 0$. (Actually, $F(x, y, z)$ represents a bit more than the surface: it splits the space into the "interior" with $F(x, y, z) < 0$, and the exterion for F positive.)

For example, in order to make an object (here: a box) distorted by an axial torsion as in Fig. 4, we programmed

```
#macro Torsion(p,fnobj) //fnobj is a funct. obj.
 #local rotx=function{x*cos(p*y)+z*sin(p*y)}
 #local rotz=function{-x*sin(p*y)+z*cos(p*y)}

 isosurface{
  function{fnobj(rotx(x,y,z),y,rotz(x,y,z))}}
#end

// (Extruded square)
#declare bbox=function{max(abs(x)-1,abs(z)-1)}

Torsion(1.4,bbox)
```

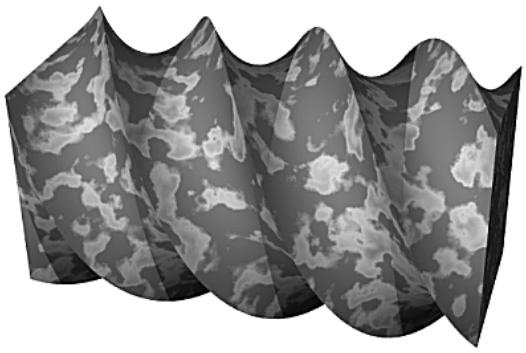

Figure 4. A screw as implicit surface

Of course, the same macro could add torsion to other implicit (iso-) surfaces. Dozen of other distortions have been proposed and implemented. It was an occasion to learn the fact that if we want that the points $\vec{x}$ composing a figure undergo a transformation $\vec{x} \rightarrow \mathbb{R}\vec{x}$, then the surface represented as $F(\vec{x}) = 0$ must be transformed contra-variantly, and changes into $F(\mathbb{R}^{-1}\vec{x}) = 0$. This means unfortunately that several useful distortions are difficult to implement because finding the inverse transform may be very complicated, but it was an occasion to use iterative approximations.

2.3 Blending

A particularly interesting subject was the functional construction of the CSG objects: union or intersection of implicit surfaces, with blending (smoothing) functions, which produced the effects like that on Fig. 5.

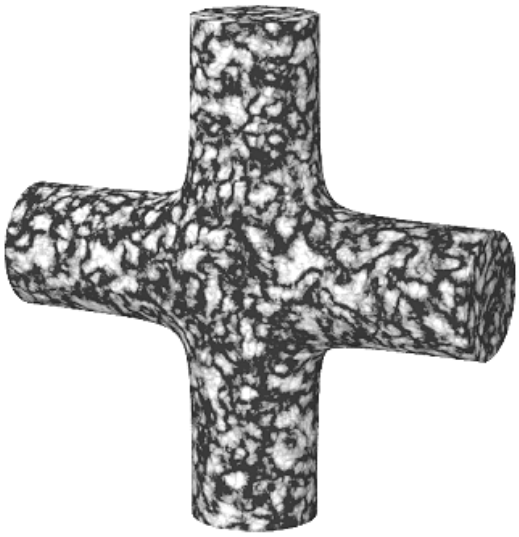

Figure 5. Blended union of two cylinders

Several interesting examples have been based on the s.c. R-functions proposed by Rvachev [6]. For example, if F_1 and F_2 represent two implicit surfaces, then $F = F_1 + F_2 - \sqrt{F_1^2 + F_2^2}$ is their union. Adding to this form a term which is very small everywhere apart from the region where F_1 *and* F_2 are close to zero, e.g. $d(F_1, F_2) = a_0/(1 + (F_1/a_1)^2 + (F_2/a_2)^2)$, or some Gaussian, the transition between functions is smoothed. In order to vary a little the procedure, we coded the blended union using the Ricci approach [7]: $f = (f_1^{-k} + f_2^{-k})^{-1/k}$, where f is a function whose value at the surface is equal to 1. So, for the standard representation, the POV-Ray program which generated the Fig. 5 took the form:

```
#declare axcyl=function(a,b){a*a+b*b-1}
#declare cylx=function{axcyl(y,z)}
#declare cyly=function{axcyl(x,z)}

#macro Blend(f1,f2,k)
 #local g1=function{f1(x,y,z)+1}
 #local g2=function{f2(x,y,z)+1}

 function{
  pow(pow(g1(x,y,z),-k)+
      pow(g2(x,y,z),-k),-1/k)-1}
#end
```

```
isosurface {
  Blend(cylx,cyly,1.7)
  ...
}
```

The CSG objects exist in POV-Ray as language-embedded constructs. But we taught image synthesis *algorithms* and their implementation, not the usage of POV-Ray, so a higher level reconstruction of some techniques served the same purpose as the construction of meta-interpreters of Lisp or Prolog in those languages. (In fact, a full ray-tracer may be and has been constructed in the language of POV-Ray, but this was not so interesting from the declarative point of view).

To summarize, the language of POV-Ray permits to define generic constructions, parameterized, recursive macros, and arbitrary composition of functions which represent implicit (or parametric) surfaces. The declarative style of scene definition is natural, and inspiring. In case of more complicated programs, which become too inefficient, the *natural* solution is to construct *functional* programs in any "decent" language, whose output is a scene specification for POV-Ray. We can say thus that the second line of the criticism of our approach – that students will not use functional languages, having other tools at their disposal – is not so serious.

3. Blender and Python

3.1 Recursion and comprehensions

We got some practical experience using Python scripts for driving the 3D modelling/rendering package Blender [8]. Since Blender is an interactive modeller, we might suppose that constructing a fractal pyramid, like this in Fig. 6 is a rather painful task, while a recursive script producing it is straightforward, provided we know how to displace (and scale) 3D objects. (We must acknowledge that straightforward is not synonymous with trivial. . .)

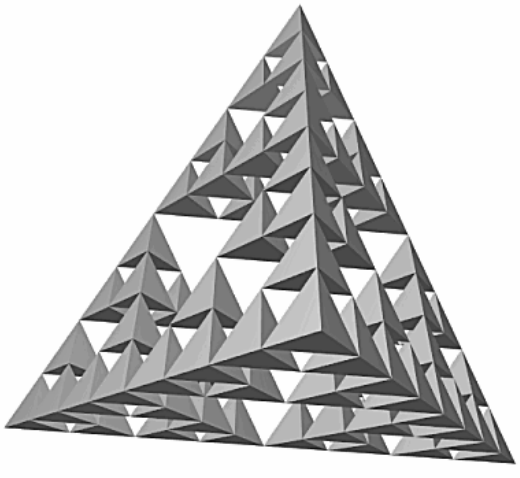

Figure 6. Sierpiński pyramid in Blender

The essential code for this object begins with an auxiliary function which 'adds' two tuples element-wise. Note the usage of pattern matching. Actually the addition in our example is the concatenation, since the elements of the tuple are two lists, containing the vertices and the facettes of tetrahedra returned by the (omitted) function `tetra`. The recursive clause is as horrible as is to be expected from a functional program which uses in one expression two maps (list comprehensions) in the form `[f(z) for z in zlist]`, and the combinator `reduce` (in Haskell: `fold`). The parameter `lp` is a list of position of vertices of a tetrahedron; they define the positions of the recursive instances of the Sierpiński sponge. We believe that it is one of the shortest programs we have seen, which generates this object. . .

```
def sp((a,b),(c,d)):
    return (a+c,b+d)

def sierpyr1(n,lp):
   if n==0:
      return tetra(lp)
   else:
      return
       reduce(sp,
              [sierpyr1(n-1,[p+lp[k]
                             for p in lp])
               for k in range(0,4)],
              ([],[]))
```

3.2 Closure export

However, in this context we wanted much more than just a recursion, we wanted to exploit seriously some generic, higher-order programming. The construction of parametric curves and surfaces is a particularly good target for such methods, especially if we prefer to convey algorithms *easy to memorize*, rather than "raw" and boring, although more efficient. How to construct a curve $c(t)$, a vector function of one parameter, which is a cubic spline passing between the points p_0 and p_1 for t in $(0, 1)$?

Almost all students can reconstruct in 15 minutes a *quadratic* spline passing through 3 given points, but higher polynomials are much more cumbersome. The functional solution is particularly simple and intuitive. We can construct a cubic function by the linear interpolation of two quads. Here the possibility of export a closure from a function is very useful. Here is the whole code permitting to construct `c = cubic(pm,p0,p1,p2)`. The functions `quad` and `cubic` are constructors (generators) of functional objects representing the curves, not the curves themselves. The points p_m, p_0, p_1, p_2 are knots, values of the curve for $t - 1, 0, 1, 2$.

```
def quad(pm,p0,p1):
  a,b = 0.5*(p1-pm),0.5*(p1+pm)-p0
  def q(t):
    return p0+t*(a+b*t)
  return q
def cubic(pm,p0,p1,p2):
  def c(t):
    return (1-t)*quad(pm,p0,p1)(t)+
         t*quad(p0,p1,p2)(t-1)
 return c
```

The pattern: construct `(def) f(x)` inside another function, and return `f` from it, became ubiquitous in Python. Not only in our exercices, but in the world of Python programmers working in scientific (and some other) domains. Despite the manifest attitude of some creators of Python, who want to remove too much of "functionalisms" for the sake of the simplification of the language, we believe that one of the reasons for its success is a reasonably good functional layer. . . The translation of a curve along an axis goes as follows:

```
def transl1(cv,ax,a):
    def f(t):
        return a*ax+cv(t)
    return f
```

We defined, of course, also the translation and the rotation of surfaces, and other lifted operations. Python has anonymous func-

tions (lambdas), it can simulate conditional expressions through the Boolean shortcuts, e.g., `a = Btest and thxpr or elsxpr`, it has list functionals such as map, filter and reduce (fold), the list/generator comprehensions, and the last versions have rationalized the scope issues permitting to export closures in a straightforward, secure way. Here is another example, a spiral:

```
def spiral(r,dz):
    def f(t):
        return Vec(r*cos(t),r*sin(t),dz*t)
    return f
```

We see here that the function returns a Python *object* — a 3D vector for which we constructed a relatively complete library of basic operations, including rotations, scalar products, etc. The call to the function `cubic` returns such vector, but note that within the definition of this function this is not explicit. The dynamic, object-oriented typing of Python facilitates the coding; it suffices that such values as `p0` etc., may be added or multiplied by scalars.

3.3 Parametric surfaces

In such a way, after having constructed a library containing e.g., the rotation of a vector about an axis: `rot(v,ax,angle)`, the definition of a figure of revolution, a *parametric surface* which is a function of two parameters - coordinates:

- a parameter along the generator, the curve which will sweep the space by rotating about the axis, and
- the angle of this rotation,

becomes quite straightforward.

We define not directly the surface, but — as in the case of curves — its *constructor* (or: generator, but we don't want to confuse this term with the generating curve…), specified by this generating curve `cv` and the axis `ax`. Here is the construction of this generic revolution constructor:

```
def revol(cv,ax):
  def f(s,phi):
    return rot(cv(s),ax,phi)
  return f
```

Making a generalized cylinder, or a linear extrusion of a generator curve along an axis is even simpler. Again, a constructor:

```
def extrude(cv,ax):
  def f(s,t):
    return t*ax+cv(s)
  return f
```

Being able in such a way to 'lift' vector operations onto the domain of functions, permits to compose those operations quite easily. Here is the canonically oriented torus, with its axis along z (`AZ`).

```
def torus(rr,r):
    gen=transl1(crotate(circle(r),AX,PI/2),
                AX,rr)
    return revol(gen,AZ)
```

Such sea-shell as in Fig. 7 is a program of 5 lines, provided we use another function from our library, the Catmull-Rom spline which iterates `cubic` over a list of points specifying this splite.

```
def catrom(lp):
    pm=quad(lp[2],lp[1],lp[0])(2)
    pp=quad(lp[-3],lp[-2],lp[-1])(2)
    lp=[pm]+lp+[pp]
    fp=[cubic(lp[n],lp[n+1],lp[n+2],lp[n+3])
        for n in range(0,len(lp)-3)]
    def f(t):
        k=int(t)
        return fp[k](t-k)
    return f
```

Note the usage of list comprehension, and also of the fact that we constructed a *list of functions*. The result returned by this constructor is a closure which selects the appropriate segment function according to the parameter. This is commented below.

The transformation of the generating curve is a homothety composed with rotation. Concretely, the generating curve upon the rotation by the angle φ is scaled by the amount $\exp(\kappa \cdot \varphi)$. The origin of the coordinate system is the singular point, the top of the shell, so this scaling translates the curve.

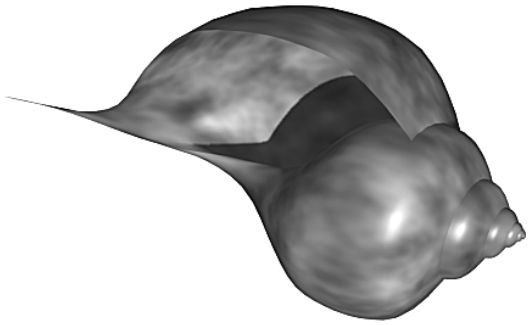

Figure 7. A sea-shell in Blender

Dozens of other sea-shells, snails, etc. have been generated. We used those functional contraptions not only to generate, but also to deform other functional surfaces, such as the deformed sphere in Fig. 8. The generic deformation functional of a surface `sf` through

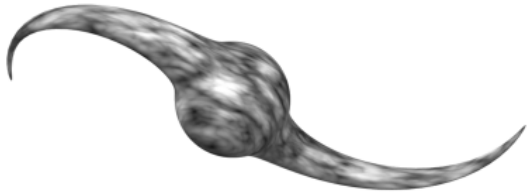

Figure 8. A deformed sphere

a function `df` which acts on points, is as simple as that:

```
def deform(sf,df):
    def f(s,t):
        p=sf(s,t)
        return df(p)
    return f
```

The deforming function was a composition of an elongation in z: $z \to z \cdot (1 + fe^{-\alpha(x^2+y^2)})$, and a rotation, whose angle depended on the transversal radius: $x^2 + z^2$; a similar function was used to create the "galaxy".

Many other generic constructs are easily programmable, for example the surfaces which interpolate between two curves:

```
def intpol(c1,c2):
    def sf(s,t):
        return (1-t)*c1(s) + t*c2(s)
    return sf
```

Other constructions, such as the Coons, 4-curves interpolation surfaces, are equally easy. Already some years ago we tried to use such techniques quite intensely [9], but we wanted then to exploit the elegance of a *typed* functional language (Clean). Since the interfacing, the 3D plotting, texturing, etc. was a bit painful, we used Clean to generate sampled points and other data stored in a file, and plotted then by Matlab (or Scilab). We observe that the current approach is easier, more comfortable for students (less tools to master). The dynamic typing offered by Python plays here a positive role, despite the known advantages of static typing for the debugging, which in a pedagogical context is a severe headache....

We acknowledge the existence of other projects combining several tools, for example the nice package Haven [10] of Anthony Courtney, which combines Haskell as a generating tool with Java for the rendering of the Scalable Vector Graphics scene descriptions.

3.4 Surfaces out of surfaces

Construction of a knot, like in Fig. 9 is not extremely easy, even if the students are told that this is a tube which follow the 'director' curve which wraps around a torus.

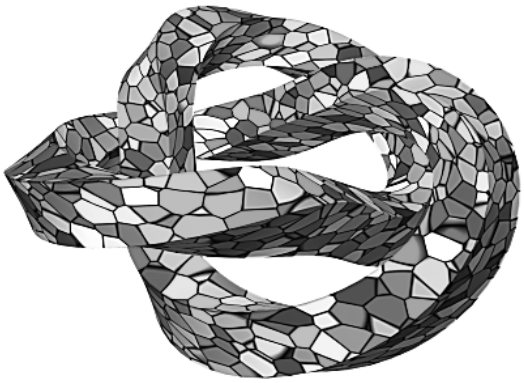

Figure 9. A toroidal knot in Blender

There are three ingredients in this construction, all three functional. First, we make a torus as a classical revolution surface, whose generator is a circle. Then, a *curve* is defined by constraining the torus angular parameters (ϑ, φ): $\vartheta = 2/3 \cdot \varphi$, and finally a *tube* is constructed with this curve as its director, and any generator, e.g., a square.

The construction of squares, of general splines, and other curves defined segment by segment was another exercise in functional programming. If the call to the functional generator `sg(p0,p1)` returns a curve function — the straight line passing by the specified points, the generator of the square take four corners, and constructs a *list of functions*: `sq=[sg(p0,p1, sg(p1,p2), sg(p2,p3), sg(p3,p0)]`. Then for t in $[0, 4]$ we take `k=int(t)`, and we return `sq[k](t-k)`, avoiding all case analysis. This is conceptually quite trivial, but convincing students that the construction of data containing functional objects is natural and useful, takes some time. When they grasp this idea, the functional construction of L-systems becomes considerably easier.

The example above, the construction of tubes posed another pedagogical challenge. A tubular object is a useful generic construction, which takes two curves, the generator and the director, and one fixed vector `n`, typically a normal to the generator plane. Then the generator is translated along the director, and rotated in such a way that synchronously rotated `n` coincides with the local tangent of the director. There are two distinct problems here, the construction of the tangent, and the choice of the axis/angle of rotation.

We could thus apply the strategy already known — making a functional which takes a function as parameter, and returns another function, its derivative computed numerically. The rotation problem *is* a challenge, requiring a decent knowledge of the Frenet frame computations, permitting to adopt a strategy which avoids too much torsion (which is visible in Fig. 9). We couldn't enter the full game of differential geometry, but several functional constructs which specified normals, etc. have been proposed. Although we couldn't develop the technique, we mentioned the possibility to use the *automatic differentiation technique* to compute the gradients [2]. Our surface constructor library contains this module implemented in Python as well.

We wanted to be able to use standard arithmetic operators for functions, so that `f=f1+f2` mean `f(t) = f1(t)+f2(t)`, etc. But Python does not permit such overloading of the addition operator, so we applied a known trick — Python *objects* have been constructed, for which we could define those overloaded operators (the functions `__add__` etc.), and also we overloaded the `__call__` methods permitting the object `q` to act as a function: the form `q(x)` is converted by the compiler into `q.__call__(x)`. The effective function was embedded within such an object, and invoked indirectly through the overloaded `__call__`. Then, the codes for the extrusion objects, etc. become significantly shorter.

We were thus able to convince the students on a practical set of examples that functional and object-oriented techniques may, and should go together. This version of our vector/curve/surface library is still in an experimental stage, since our department begins right now to teach Python to computer science students in a more organized way.

3.5 Python generators and laziness

Lazy lists, trees, and other co-recursive constructions are inherent to pure functional languages, and usually absent from languages with mutable data, since the delayed evaluation in such a context leads to ambiguities. Notable exceptions are lazy streams in Scheme and in Caml, but their usage is relatively rare and restricted.

The basic idea of constructing an "infinite" data structure, defined through a recursion without terminating clause, and consumed incrementally, with the *by-need* instantiation (and memoization), can be programmed in Python, thanks to the concepts of ***generators***. How can we use them for computer graphics?

A small fragment of our course on image synthesis was devoted to the sampling/polygonization of implicit surfaces. We discussed the relevant theory (variants of the marching cubes algorithm, etc., see [5]), but we found it useful to propose a simplistic polygonizer based on octrees, in order to be able to work with implicit surfaces within Blender. The idea of a lazy functional implementation of such a polygonizer can be found in [11].

Structurally an octree is a 8-fold tree, whose root corresponds to a cube, and its branches — to the 8 smaller cubes obtained by the triple binary subdivision of the root. The algorithm starts with the embedding of the implicit surface $F(\vec{x}) = 0$ in a cube, and performing a few (1 – 2) initial subdivisions. Those sub-cubes which cut the surface, i.e. whose vertices give different signs of F are subdivided further, up to the desired precision. The cubes "inside" or "outside" remain undeveloped. The points of intersection of the surface with the cubes' edges are then assembled into polygons, and Blender does the rest.

We shall not present the whole package, just the procedure-generator which constructs an infinite octree. It is parameterized by the subdivision length, and by the root cube. A cube is an object whose attribute is `vs`, the list of vertices (coordinate vectors). Its constructor, `Cub` is parameterized by its two opposite corners.

The octree itself is an object containing the root cube and `br`, the list of branches, which are octrees.

```
def mkoct(n,cub):
  lfd,lfu,lbd,lbu,rfd,rfu,rbd,rbu = cub.vs
  ct=0.5*(lfd+rbu)   # The center
  br=map(lambda (v1,v2): mkoct(n+1,Cub(v1,v2)),
       [(lfd,ct),(0.5*(lfd+lfu),0.5*(lfu+rbu)),
        (0.5*(lfd+lbd),0.5*(lbd+rbu)),
        (0.5*(lfd+lbu),0.5*(lbu+rbu)),
        (0.5*(lfd+rfd),0.5*(rfd+rbu)),
        (0.5*(lfd+rfu),0.5*(rfu+rbu)),
        (0.5*(lfd+rbd),0.5*(rbd+rbu)),(ct,rbu)])
  yield Oct(n,cub,br)
```

Note the endless recursion (8-fold, inside the `map` functional), rarely seen outside such languages as Haskell or Clean... In fact, the presence of the keyword `yield` makes this procedure a *generator*. Its call, e.g., `oc=mkoct(0,unitCube)` returns a "suspended object", instantiated through the method call `oc.next()`. The consumer procedure doesn't really care about that, since it obtains the access to the k-th branch of the argument by the call `b=branch(oc,k)`, defined as follows:

```
def branch(oc,k):
    x=oc.br[k]
    if isinstance(x,GeneratorType):
        x=x.next()
        oc.br[k]=x
    return x
```

and which ensures the development by need with memoization. Of course it *can* be done differently, but the lazy approach is compact, the case analysis during the consumption process is much simpler. The consumer procedure `consume(n,F,oc)` is of course sensitive to the depth n. If it is greater than the limit, the cube is split into 6 simplexes (tetrahedra), and the procedure returns the list of polygons. In the recursive case the only operation which is necessary reduces to

```
...
return sum([consume(n+1,F,branch(oc,k))
            for k in range(0,8)],[])
```

(The function `sum` is a generic adder, which folds the overloaded `(+)` operator; in the current case it is the concatenation.)

The current version of the polygonizer produces the results somehow ugly, and needs further development. The advantage of having this within such modeller as Blender is the possibility to process them further interactively.

Another project which will be developed next year consists in treating the implicit surface function as a "force field", which constrains a set of initially free particles to locations near the surface [12]. The particles can then be used for the reconstruction of a mesh, and also for texturing.

4. Functional Techniques in Image Processing

The course on image synthesis was completed by a short introduction to image processing methods. Here we have worked with the image-processing package Gimp [13], which is extensible through scripts written in Scheme (SIOD). So, in principle, the functional programming could be exercized quite extensively, although its direct applications were limited, since the aim of the course was just to introduce the basic filtering, morphological operations, and some manipulations in colour spaces (which would facilitate the segmentation).

The generation and transformation of images considered as functions $f(x, y)$ whose codomain is the pixel colour, has been extensively studied, see e.g., [14, 15], and we have shown to students several dozens of images generated by programs in Clean and in Matlab. We used them as texture patterns for the 3D synthesis.

Gimp permits to draw lines, rectangles, ellipses, etc., so we could easily construct all standard Koch-style fractals, 2D L-systems, etc. (no need to show ubiquitous examples), although in this context Gimp doesn't offer anything remarkable, as compared with fuller Scheme implementations, offering a complete graphic support, such as DrScheme [16]. It is impractical to draw figures pixel by pixel, for efficiency reasons.

On the other hand, it was rather easy to construct arbitrary functions: *gradients* which generated 1-dimensional (cartesian or polar; x, y or r) slices of grey images, which were then used as *displacement maps*. The Scheme (Script-Fu) layer in Gimp provides more than 100 useful functions. For example, a linear gradient demanded just the creation of an empty gradient object, and seting its left and right colours. The linear interpolation between was automatic.

Their usage is the following. The target image is scanned point by point, and for each pixel we accede to the pixel within the source image, which has the coordinates of the target pixel *modified* by the *value* of the corresponding pixel of the displacement map. (So, the displacement map should in general be a 2-component image).

A simple *linear* (the value of the pixel is proportional to its relevant coordinate), black-white horizontal gradient interpreted as the displacement function $f(x) = ax$, could give the displacement $x \to x + ax = (1 + a)x$, a scaling transformation.

A linear gradient in x used as the y displacement, together with its transpose acting on x, permitted to construct rotations (with some rescaling): $x \to x + ay$; $y \to y - ax$ (simultaneously) of any image. By composing the linear gradient with radial weights also defined as a gradient fonction, it was easy to obtain radially weighted rotations, effects like the right image in Fig. 10.

Figure 10. Displacement mapping in Gimp

Of course, there is nothing extraordinary in this whirl effect, available often as a primitive in many image processing packages. But we wanted to convey its mathematical background and its implementation.

Much more complicated functions have been constructed with function compositions (displacement maps which distorted other displacement maps, but we couldn't go too far since Gimp uses 8-bits per colour, so the discretization noise became quickly unbearable. For more consequent projects we abandoned Gimp in favour of other packages (e.g. Matlab, which unfortunately could not be distributed freely to our students). The implementation of some functional tools in Java for ImageJ [17] is under investigation. Again we shall try to convince students to *think* functionally, independently of the language used.

5. Conclusions

The usage of functional methods in computer graphics is an everlasting issue. Already in [18] we saw many interesting ideas on

how to compose functional objects representing parametric graphical entities. Since then we have seen many projects in Haskell and other languages, too numerous to cite. Unfortunately, serious attempts to use functional techniques as a more or less coherent and serious teaching methodology in the domain of computer graphics seem to be rare.

Since we teach the image synthesis already for some years, the project is in constant development, but we can already say that it was an immense fun for everybody, and it occupied us for almost four months of the semester; it cannot be presented fully here. In a pedagogical context, where the time alloted to one exercise is limited, students could not concentrate themselves simultaneously on the algorithmic side of the problem, and on the interfacing. When they had to use primitive libraries (OpenGL, etc.) their visual results were too often ugly, because the programs were too simple.

By programming POV-Ray or Blender they could spend more time on the geometric design through parametrization, recursion, etc., and they could obtain something much nicer, and thus more encouraging, quite fast. We underline the fact that we had a concrete pedagogical program to realize in collaboration with other people, and we could not sacrifice the images synthesis/processing techniques just to have more time for playing with functional tools.

The curriculum of Master-1 (4-th year) in Computer Science includes a personal programming project which *should* occupy the student usually during 5 – 6 months. We delegated to those projects some more involved problems, such as the functional construction of an automatic differentiation package inspired by [2] in the context of graphics, the polygonization of implicit surfaces, and many others. Other projects, shorter but more complicated (such as the texture reconstruction using the Heeger technique [19], or the implementation of active contours — all implemented as functionally as possible) have been proposed to Master-2 (fifth year) students. This didn't work always as expected, but we can affirm the following.

- Our somewhat eclectic approach, combining the high level functional code with the necessity of implementing some imperative constructs, was a *good thing*. It permitted to our students to split the global problem in layers, to control better the interplay between various programming styles. In a sense, the functional methodology, taught formerly through Scheme and Haskell, ceased to be a "religion", and became a "weapon"...
- Because of the fact that image synthesis in general is an interesting topic to learn (it was by far the most popular optional module), the interest of programming graphic algorithms in a structured and fast way thanks to functional methods was considerable. It was a pleasure to observe that students themselves proposed such programming projects as the construction of a ray tracer, or a texturer/shader in a functional language.

So, the presented philosophy seems promising, the project will continue, and we will try to adapt it to *free* programming tools available for the students, since we do not feel restricted to any concrete language or package.

References

[1] Jerzy Karczmarczuk, *Functional Low-Level Interpreters*, Functional and Declarative Programming in Education, Pittsburgh, (2002). Available also from `www.info.unicaen.fr/~karczma/arpap/Fdpe02/fumach.pdf`.

[2] Jerzy Karczmarczuk, *Functional Differentiation of Computer Programs*, Higher-Order and Symbolic Computation **14**, pp. 35–57, (2001).

[3] Jerzy Karczmarczuk, *Functional Framework for Sound Generation*, Proceedings, Practical Aspects of Declarative Languages, PADL'05, Long Beach, Springer LNCS 3350, pp. 7–21, (2005).

[4] POV-Ray: collective work, which began with David Kirk Buck and Aaron Collins. Distribution and documentation: Web site `www.povray.org`

[5] Jules Bloomenthal (ed.), *Introduction to Implicit Surfaces*, Morgan Kauffman, (1997).

[6] V. L. Rvachev, *Theory of R-functions and Some Applications* (in Russian), Naukova Dumka, (1982). See also A. A. Pasko, V. Adzhiev, A. Sourin and V. V. Savchenko, *Function representation in geometric modeling: concepts, implementation and applications*, The Visual Computer **11**:8, pp. 429–446, (1995).

[7] A. Ricci, *A Constructive Geometry for Computer Graphics*, The Computer Journal **16**:2, pp. 157–160, (1973).

[8] Ton Roosendaal and others; Web site `http://www.blender3d.com`. See also T. Roosendaal, C. Wartmann, *The Official Blender Gamekit: Interactive 3-D for Artists*, No Starch Press, (2003).

[9] Jerzy Karczmarczuk, *Geometric Modelling in Functional Style*, Proc. of the III Latino-American Workshop on Functional Programming, CLAPF'99, Recife, Brazil, 8-9 March 1999.

[10] Antony Courtney, *HAVEN, Scalable Vector Graphics for Haskell with GCJNI and Java2D*, accessible through the Haskell Web site.

[11] Th. Zörner, P. Koopman, M. van Eekelen, R. Plasmeijer, *Polygonizing Implicit Surfaces in a Purely Functional Way*, 12 Intl. Workshop on the implementation of functional languages, IFL'00, Springer LNCS 2011, pp. 158–175, (2000).

[12] Andrew P. Witkin, PAul S. Heckbert, *Using Particles to Sample and Control Implicit Surfaces*, Procs. of the 21st SIGGRAPH Conf. on Computer Graphics and Interactive Techniques, pp. 269–277, (1994).

[13] Peter Mattis, Spencer Kimball and many others; Web site `http://www.gimp.org` for the software and the documentation (regularly updated...).

[14] Conal Elliot, *Functional Image Synthesis*, Bridges 2001 Conference: Mathematical Connections in Art, Music and Science, Winfield, July 27–29, 2001. See also *Functional Images*, a chapter in *Fun of Programming*, ed. Oege de Moor, Jeremy Gibbons, Oxford (2003).(Palgrave series Cornerstones of Computing).

[15] Jerzy Karczmarczuk *Functional Approach to Texture Generation*, PADL (2002), Portland, Springer LNCS 2257, pp. 225–242.

[16] The PLT-Scheme Web page, `www.plt-scheme.org`; full distribution and documentation.

[17] *ImageJ: Image Processing and Analysis in Java*, Web page `rsb.inf.nih.gov/ij`.

[18] Peter Henderson, *Functional Programming, Application and Implementation*, Prentice-Hall, (1980).

[19] D.J. Heeger, J.R. Bergen, *Pyramid-based Texture Analysis/Synthesis*, Procs. SIGGRAPH '95, pp. 229–239, (1995).