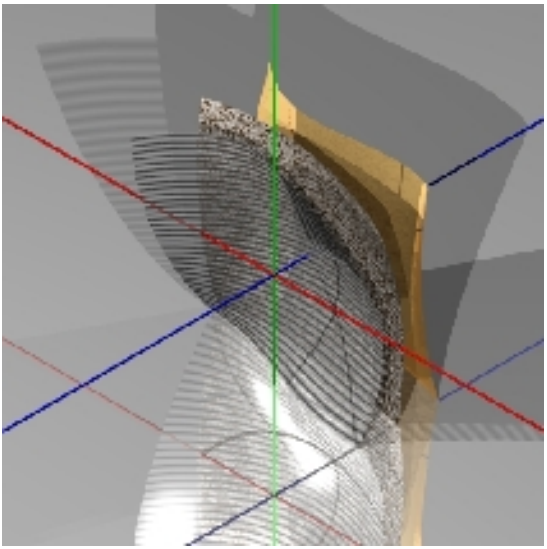
pascalian forms



essay on curved shapes

alain marty

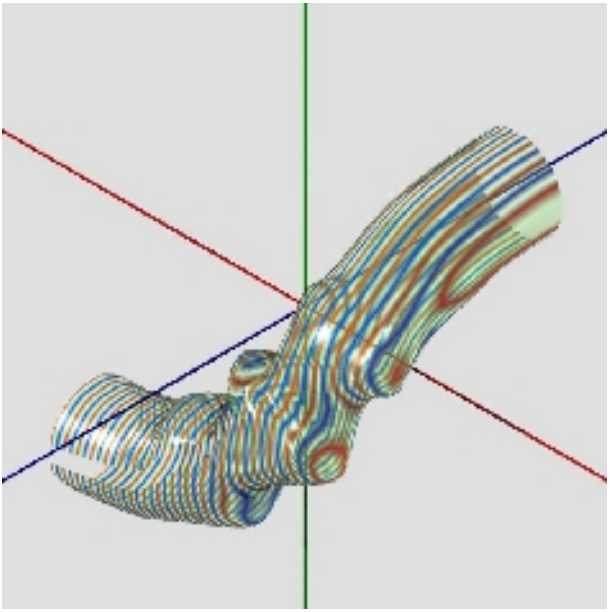
First printed August 2004.

Original edition by « Editions de l'Esperou » based in « ENSAM » (Ecole Nationale Supérieure d'Architecture de Montpellier).

Second Edition in English, December 2006, by « Editions de l'Esperou ». Translation by Angela Kent, grace to the ENSAM's SLA laboratory (Structures Légères pour l'Architecture).

This work was realized with the aid of the French « MCC », Ministery of Culture and Communication, direction of architecture and patrimony, office of architectural and urban research.

Table of contents



tubing modelled on a cubic

preface

Alain Marty's work is the fruit of a particularly exacting author, who has spent a long time on the subject he proposes « Pascalian Forms ». Understanding the dialogue between « Forms and Forces », which is at the core of the « Editions de l'Espérou » collection, requires an open mind, able to link rational and sensitive approaches as part of the same movement.

The author's double training as Architect and Engineer would have been sufficient in itself, but the algorithms proposed here called for an extra quality in order to generate and represent the curved surfaces that are the subject of this work. This quality is the simplicity with which he approaches computer science; and simplicity is something Alain Marty continually advocates in his work as a teacher. It is from his standpoint at the intersection of pertinences, that Alain Marty is able to guide us « straight ahead » along the road of « curved space ».

Robert Le Ricolais, known to light structure specialists for his important work in this field, insisted on « the joys of mathematics », joys that could not emerge, in his view, without « a few tears ». Reading this essay on « curved surfaces » will likewise be a source of pleasure, provided the reader is prepared to invest some time in doing so.

For my part, I am sure that those who undertake the task will find that if, only too often, people mistake the idea for the tool, particularly in the area of computer science, Alain Marty has been able to bring out the geometry and construct a tool to explore its complexity.

It is my pleasure to thank him for this, with these few lines which also bear witness to a meaningful encounter with the Languedoc-Roussillon School of Architecture, within the Light Structures for Architecture team.

René Motro

Senior Lecturer at Montpellier University II

forward

This essay on curved forms is the (provisional) outcome of a long road through the realm of form modelling, starting back in 1968 with a final studies report on freeform Thin Shells, interrupted until the 80s by my work as practising Architect constructing with a T-square and set square, then taken up again, using the first microcomputers that were affordable. Things accelerated in 1990 after making fruitful contacts at the EALR with René Motro, Senior Lecturer at Montpellier II, and Head of the GRSLA laboratory on Light Spatial Structures for Architecture, based at the Languedoc-Roussillon School of Architecture.

This work came out of discussions between « amateurs », with no particular assignment, in complete freedom, outside any course programme or university convention. It is based on a set of knowledge from past studies, self-learning built up on information gleaned from books you can pick up from any city bookshop. It is the result of a persistent desire to arrive at a clear, unitary approach to the wonderful but complex world of « curved forms » and a need to share this thinking with fellow « explorers », whether atypical amateurs or members of an institution, mathematicians or not, theoreticians or practitioners, experts in the Art of Line-drawing of the master builders in the Middle Ages or virtuosos of the new modelling devices in Computer Graphics.

I was lucky enough to discover this interest on the part of René Motro, motivating me to go back to working in the area of control point forms (Bézier, splines), and I am grateful to him for this. I also had the good fortune to meet a team of researchers at the GRCAO laboratory at Montreal University, in 1995, who were sympathetic to my ideas; I am referring here to Giovanni de Paoli and Claude Parisel in particular. And at Montpellier School of Architecture, for the continued and critical interest of Thierry Berthomier, a Structural Morphology buff, and occasionally from the students who were prepared to leave aside their architecture assignments to listen (for a while) to an original presentation of curved forms; I'm thinking particularly here of Vinicius Raducanu who has since gone on to follow his own path. I hope, through this essay, to meet other curved form enthusiasts who can help take this exploration further, who could perhaps use it in their own work, and/or share the road with me for a while. I hope at least that readers will get some pleasure out of this essay.

And I'd like to thank Colette, who was prepared to pretend she believed me every time I promised I only needed fifteen more minutes to « really » finish the study...

Alain Marty

Villeneuve de la Raho, Pyrenées Orientales, France.

July 2004, December 2006

introduction

Construct four equilateral triangles using six matches. That was the question Bernard Weber recalled in his first book on ants, with the added enigmatic reccommendation: « Think different ! », the key to the answer. By thinking differently, the Greeks invented the Theory of Conics, combining in a coherent whole, curves that are as diverse as the circle, the elipse, the parabola and hyperbola, as well as the straight line and the point. Much later, mathematicians invented the complex number field "so that" an N degree equation « always » has N solutions. The Chemist Mendeleev came up with a great classification for the seemingly so different primary elements that make up the known universe, and beyond - a formidable tool for discovering new elements. By writing mechanical and electrodynamic formulas in four dimensional space, Einstein not only succeeded in unifying the fundamental concepts, he also created the conditions for new discoveries, of which the famous formula E = m.c2 is the best known. Illustrious cases...

A complex question posed in today's space, one that is three dimensional and real, can often be reformulated more simply by transposing it « for a while » to an imaginary, slightly more complex space. It's a way of dividing up the problem to master it better. For example, it is easier to study the stability of a thin curved shell when the problem is posed as the curved geometry of this shell, rather than the orthogonal euclidian space surrounding it.

But the geometry of curves and curved surfaces is, in itself, an example of a complex problem that is difficult to model, represent and visualise. Mathematicians of the last centuries admittedly marked it out with wonderful differential formulae, but these are often beyond the field of application of « descriptive geometry » with its use of simple tools like a ruler, a compass or a simple freehand drawing. Beyond these elementary forms of classic geometry, straight lines, circles, planes, spheres etc., one has to admit that nothing can be easily manipulated without the help of computer tools. In its applications to computer graphics and CAD, computer science has provided us with operational modelling for classic geometrical forms, but has also also brought out a new family of curved forms that are very practical to manipulate on the screen (Béziers, Splines, Nurbs, Coons squares,..). But because of the operational orientation of the computer tools developed, these forms cannot be apprehended directly even less manually. The algorithms developed in the imposing and sometimes indigestible literature are complex or hidden in the depths of software black boxes. As far as I know, no unitary approach accessible to the layman has really emerged to bring these forms to the fore and create the conditions to discover new ones.

Such is the aim of the present essay on curved forms.

De Casteljau proposed an algorithm of the same name in 1959, a fundamental recursive algorithm that is stunningly simple, a geometrical construction that is highly intuitive leading to a potent theory, the type of algorithm that is based on a « gestural » approach that seems off-beat, old fashioned and which one could come across several times without ever seeing any more than a simple sketch designed to accompany the algebraic, analytical and matricial formulas that fill the literature on the subject. A simple, almost esoteric sketch, taken from the art of line drawing of builders in the Middle Ages...

The present essay gives this algorithm a central position, using it as a systematic application with the aid of a handful of elementary geometrical operations; it adds some elements to a descriptive geometry of

« pascalian »forms, building a bridge between classical geometrical forms and the new forms that have emerged from computer science, defining forms whose rules ultimately allow freehand drawing, with a piece of string as the only guide... Thus the forms that go by the name of Bézier curves and surfaces are reconstituted simply by using the geometrical operators initially applied to a pair of points in space, and up to any number of these forms.

This unitary approach, which is intuitive and based on analytical formulation reduced to the minimum, simplifies access to the case of forms immersed in other curved forms and to deducing the most complex forms like Splines, Nurbs (whose conics constitute a particular and very important case), tubings, Coons squares, etc...., all with projection, concatenation and simple linear combination operations.

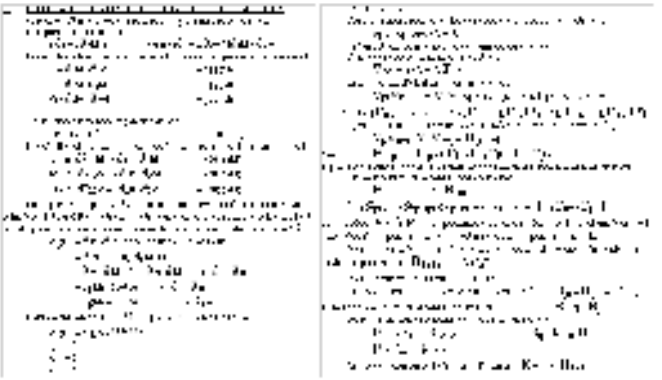
The set of « pascalian », forms, or « pForms », equipped with these operators thus comprises a descriptive geometry, a tool to aid direct and « manual » understanding of forms whose analytical expression can sometimes be very complex, a tool that can help explore new forms up to their higher dimensions.

The document is divided into three main parts :

1.construction, definition : this part presents a progressive approach to pForms bringing out the unitary character of forms that seem at first glance to be very different - such as the cube and the ... cubic - leading to a general definition of pForms ;

2.operations, properties : this part presents the operations that can be carried out on pForms -subdivisions, degree elevations, ...-, and the fundamental properties of pForms - tangent axes, interpolations, embedding in other pForms, ... - ;

3.compositions, applications : this part analyses some compositions specific to pForms, their relations to conics, notably to revolving surfaces, preparing the way for a geometry of curved surfaces.



From differential geometry ..

computer implementation

The exploration of curved forms began by hand, and freehand drawings were the first basis for reflection. Drawing a cubic using trade software does not usually add much information about the internal mechanics of the curves produced; we're simply consuming. But the mind that guides the pencil on the paper or the cursor on the screen often mistakes its desires for reality, so the drawing that comes out may not really match the reasoning process, leading to a false concepion of the envisaged form. Algebraic reasoning then came to assist drawing, but can we really trust a result obtained at the cost of five or six pages of cramped calculations where a single erroneous sign can completely undermine the reasoning?

Although software to manipluate geometrical forms proves to be of little interest in studying their foundations, the remarkable tools of programming language can be used to back up thinking at the most fundamental level. All that reains is to choose the right language and this is no mean feat: choosing the right development tools, the right environment and the right platform all demand a high investment in learning how to use all these elements. Basic, Pascal, Hypertalk, Logo, C, C++, Renderman, Open GL, Java, Java 3D, you need a lot of time to master the vocabulary and syntax, along with the purely geometrical thinking process to be developed - not forgetting the computer tools in the all this maze. It's easy to end up spending more time reinventing the algorithm to fill in a 3D polygon to display in a window, than thinking about the actual nature of the form to be modelled and studied on the screen.

Thus the interest of a tool like POVRAY, open software that works on all platforms based on a RayTracing rendering engine and developed by a community of academics and researchers from all countries, whose site can be found at the following address: http://www.povray.org. Apart from its excellent rendering engine, its many capacities (boolean operations for instance) and numerous predefined objects, POVRAY software uses a real development language ; it is possible to define global and local variables, chains, tables, test and control structures to create loops, macros that behave like real processes and functions, loaded with parameters, reversing values and moreover able to be called up recursively, which is the fundamental property required by the present study. This software is a real gift for the computer graphics researcher and when you consider that the language used is very standard and close to C language, the "universal" language, you "know" that it will always be possible to put your work on any other software platform.

The operators studied were implemented on POVRAY and have been incorporated in a library as part of a 'pFlibs.inc' headed by the 'pFscene.pov' files rendered by POVRAY. This implementation was linked to the developing the basic operators and many others, for easy visualisation of the forms produced, « visual » control of their validity, and the production of high quality images as an end result. Visual control means that the cramped pages of calculations intended to demonstrate such and such a result have been replaced by pages of computer code that are just as complex. The difference is that this computer code constitutes a set of algorithms that are faithfully and effortlessly executed by the computer, as many times as needed and in all possible or imaginable conditions. And the fact that this machine has the annoying tendancy to execute exactly what it is asked to do rather than what we actually want it to do! In general, the result is not quite what we expect, leading to trailing back and forth until we finally get the expected result... either that, or we finally admit that the concept envisaged was ridiculous anyway.

POVRAY has really been a great help in finishing this study, to the extent that the initial syntax used to develop the reasoning in this work, was quickly joined and sometimes replaced by POVRAY's. For instance, in the syntax used to present geometrical concepts, the definition of a parabola defined by three points is written as :

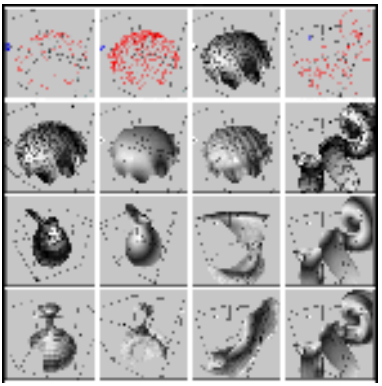
```
pL3 = MIR(p0,p1,p2),
      recursif operator MIR applied to three points
```

on the basis that the recursive operator is assumed to apply to an infinite recursion level. In the syntax used in the POVRAY implementation, the definition and representation of the parabola are written as :

```
#local pL3 = array[3] { p0, p1, p2 }
pFdraw( 1, pL3,
      finesse(3) + courbe(0.01) + ma_couleur(<1,0,0>) )
```

which can be read as : a local variable (pL3, cf notation below in the text) associated with the parabola is stated and defined as a three point table (p0, p1,p2); the macro pFdraw() applies a level 3 recursion to the table (refinement (3)), producing a 9 point table, drawing it in the form of a polyline passing through the nine points, composed of cylinders with a radius of 0.01 (curve(0.01)) coloured red (my_colour(<1,0,0>)). The statement and representation of a form will generally be completely dissociated; a parabola will be defined by the data of a three point array, with no need to choose its representation, the width and colour of the line of course, nor, what's more, the recursion level, to the extent that there is no distinction between the control polygon defined by the initial three points and the parabola generated by infinite recursion.

At the end of the work, readers will find a complete list of the operators that can be used as macros, compiled in a file, « pFlibs.inc » headed by the « pFscene.pov » files to analyse their functioning and produce images or animations.



... to computer tools !

geometry reminder

Summary of this section :

To help situate the « problematic » of curved shapes (curves, surfaces, curved volumes), it would be useful to look at some reference points in elementary geometry and primitive forms, in cartesian geometry and equations, and to attempt a first classification, or unification at least, of curved forms.

01 elementary geometry and primitive forms

While few people have read « Euclid's elements », the sum of which still forms the basis of geometry today, everyone has more or less toiled over some figures traced with a ruler and a compass, equal triangles, Thales theorem and parallels, and it's true that for many people, geometry stops there, at points, straight lines, triangles and circles. A few venture beyond into the corners of polyhedrons, and get utterly lost in the jungle of icosahedrons, rhombic dodecahedrons and other forms prized by the christallographer... as for curved forms like the trajectory of a bird meandering through the sky, the curve of a breaking wave, the hips of Rodin's Danaide, these seem quite beyond any simple method of construction, or attempt at unitary description.

The simplest curved form is the circle, which we know how to trace using a peg and a piece of string of constant length; and the surface of the sphere is constructed in the same way. But seen on the bias, a circle is no longer a circle and apparently nothing is constant any more, neither the distance from each point to the centre, nor the curvature at each point, raising the problem of constructing it directly on a plane with no reference to the circle of which it is the biased view. This curve that we call an « ellipse », presents two symmetries in relation to two orthogonal axes, so the idea came up one day of marking two

points on the big axis, that are symmetrical to the small one and to tighten a continuous piece ofstring into a triangle between these two points and a point on the ellipse. Using trial and error, it emerged that at a given position of the two points, the total length of the string remained constant and these points were called the foci of the ellipse. The construction of a curve generalising the circle and its constant radius was thus found, opening up the world of controlled curves of variable curvature.

But another curve also attracted the attention of geometricians : the curve followed (at a first approximation) by a stone thrown in a bias through the air or thrown by the powerful water jets of a fountain. This curve, called a parabola, did not bend back on itself like the ellipse that it resembled locally, and even seemed to continue toward infinity ! The construction of this curve, which was finally discovered, was not as easy as that of the ellipse and required introducing a point on the symmetrical axis (also called a focus) and a straight line orthogonal to this axis (called a directrix) ; so this was nothing like the ellipse and even less so, the circle. The unification of these curves came about with the « Theory of Conics ».

The « Theory of Conics » revealed by the Greeks, simply and elegantly demonstrates the fundamental relationships between curves that are as different in appearance as a circle, an ellipse and a parabola: all these curves are conics, sections of a cone with a circular base on a more or less sloping plane in relation to the axis of the cone. So that was it. But this required the effort of shifting from the two dimensional space in which the three curves are found to a three dimensional space where it was possible to describe a cone and its intersections with a plane. « Think different ! », that's the key to the solution, as Bernard WERBER reminded us.

And, as is often the case, a window opening onto a new space reveals new things, by sloping the intersecting plane a little more, we discover the two arcs of a new curve, the hyperbola, the asymptote concept and finally a unitary representation of six important elements in geometry : the point, the straight line, the circle, the ellipse, the parabola and the hyperbola.

We are aware of the importance of Pythagoras and his theorem on the right-angled triangle written in modern language as: a2+b2=c2 ; and we are also aware of the despair of Pythagoricians when they applied this formula to measuring the diagonal of a one-sided square and they realised (and demonstrated) that it was impossible to extract the square root of the number 2 in the form of a whole or fractional number; this number was entirely « irrational » and it was unreasonable to talk about it ; period. Hence their deduction that linking the world of geometry with that of arithmatic was totally prohibited (on pain of death) and it was two thousand years before Descartes broke this taboo, systematically associating number relations with geometrical objects and searching for the « equation ».

02 cartesian geometry and equations

In this approach to geometry, the basic idea is to associate a set of numbers called « coordinates » with each point in space, two on the plane (x,y), three in space (x,y,z); each of these numbers could take any value in the]-infinite, +infinite[interval, in a set known as the set of real numbers, R. Let's say that R2 is all the points in a two dimensional space, and R3 all the points in a three dimensional space. And we will define geometrical objects (straight lines, circles, planes,...) by the relations between the coordinates - the equations - compelling the points of these objects to remain in a limited part of R2 or

R3. We will now look at implicit equations, parametric equations and differential equations.

021 implicit equations

The implicit equation of a curve is a relation written as f(x,y) = 0, which has the advantage of a symmetrical treatment of coordinates, which is not the case in the better known explicit form : y = f(x). Related to a pair of axes Ox and Oy, a straight line in plane R2 is where points P(x,y) lie, such that y varies proportionately to x, y/a = x/b, which is usually written in the following explicit form :

```
y = a/b.x + d,
      where d is the ordinate of the point on the Oy axis
```

and in implicit form :

```
f(x,y) = ax + by - d = 0.
```

Likewise, we find the implicit equation of a parabola :

```
f(x,y) = y - ax2 - d = 0,
```

where d is the ordinate of the point on the Oy axis. The equation of an r radius circle centred on the origin of the axes follows after immediately applying Pythagoras' theorem :

```
x2 + y2 = r2,
```

and is written in implicit form as :

```
f(x,y) = x2 + y2 - r2 = 0,
```

Likewise it is easy to find the implicit equation of a sphere in space :

```
f(x,y) = x2 + y2 + z2 - r2 = 0.
```

Finding the equation for a plane is a little more complex. A plane is first defined in R3 space passing through the origin of the axes by means of a unitary vector which is normal at this point N(a,b,c), then a plane parallel to this and found at distance d ; let H be the intersecting point of this plane and the straight line on which N is found. Any point P(x,y,z) on this plane is such that its projection on the straight line containing N is point H, in other words such that the scalar product N•OP = OH, and the equation is written :

```
ax + by + cz - d = 0.
```

What could the implicite equation f(x,y,z) = 0 be of a straight line in R3 space ? No such equation exists, in fact, leading to considering a straight line in space as an intersection of two planes and defining a point P(x,y,z) as the solution to a bilinear equation system :

```
ax  + by  + cz  - d  = 0
a'x + b'y + c'z - d' = 0
```

So we see that in representing the simplest objects as implicit equations, it is hard to find unity of form and general rules ; each form involves a specific approach and this approach depends on the space in which it is found, its dimension, its curvatures...

022 parametric equations

Parametric equations offer more scope for elegantly defining a large number of geometrical forms. This involves an approach whereby the coordinates of the point of a form will be expressed as a function of one or a number of parameters generally confined to a unitary interval [0,1].

That is to say, a space of a given dimension (let's say 2 or 3), O the point of origin of the axes and any two points P0 and P1 of this space ; a point of the segment held by PO and P1 can be expressed in the form :

```
OP(t) = OP0 + P0P1.t       with t € [0.1]
```

an equation expressing vector OP as the sum of vector OP0 and a vector held by P0P1 and of a reduced length of that of P0P1 in t proportion. But the following equivalent writing is preferable, as an equation expressing vector OP as the sum of vector OP0 and vector P0P1 reduced in t proportion. But the following equivalent writing is preferable, as it is more elegant, with no reference to point of origin, and is symetrical in relation to points P0 and P1:

```
OP(t)  = OP0 + (OP1-OP0).t
       = (1-t).OP0 + t.OP1
       = (1-t).P0 + t.P1
```

where P is seen to shift linearly from P0 to P1 when t varies from 0 to 1. Note that this vector equation is equivalent in R3 space to three scalar equations in x, y and z. In fact, writing it this way avoids the problem of the number of dimensions and will be used extensively later. This writing allows a whole family of « linear » forms to be defined simply ; here we can mention - without further explanation at this stage - the portion of the double ruled surface that goes by the pretty name ofhyperbolic paraboloid (also called HP) and defined by any 4 points in space (P00, P01, P10, P11) by the following bilinear expression :

```
P(t) = (1-u).(1-v).P00 + u.(1-v).P01 + (1-u).v.P10 + u.v.P11,
       with u,v €[0.1],
```

and the arc of the parabola obtained from this HP by equalling u and v (u = v = t) and totally defined (controlled) by 3 new points P0, P1, P2 :

```
P(t) = (1-t)2.P0 + 2.u.(1-t).P1 + t2.P2,
with t €[0,1], and P0 = P00, P1 = (P01 + P10)/2, P2 = P11
```

constituting the first features of a family of shapes referred to in chapter 1, shapes that are respectively controlled by 4 and 3 points.

Concerning the circle - centred on the origin and of unit radius - the parametric equation is

immediately based on fundamental triganometric functions :

```
P(t) = [ cos(A) , sin(A) ]        with A € [0, 2.π],
```

which is of no help when we know that these trigonometric functions are transcendent functions, that cannot be expressed in the form of a polynoma with a finite number of terms. By defining a new variable t = tangent(A/2), we can dispense with trigonometric functions and find a rational expression - i.e. written as a quotient, with ratios :

```
x = (1-t2)/(1+t2)
y = 2.t/(1+t2)
```

It is hard to find a link between this expression and that of a right segment, but in chapter 3 we will see how to attach this expression to that of a parabola, thanks to the Theory of Conics...

For the sphere, the classic parametric expression in u and v :

```
P(t) = [ r.cos(u).cos(v), r.cos(u).sin(v), r.sin(u) ]
```

is difficult to transform into a rational form and we will also have to wait until chapter 3 to study this further.

023 differential equations

Some geometrical forms can often only be defined on the basis of so-called « differential » properties. This is the case for the altitude of an apple falling from an apple tree, a marble rolling on a curved surface in gravity-free space, or the equilibrium surface of a sliver of soap hung on a wire.

0231 : the falling apple

The function z(t) that governs the altitude of an apple falling from the branch of an apple tree satisfies three simple rules :

```
z(0)   = h,   initial height of the tree
z'(0)  = 0,   null initial speed
                    (unless there's a mischievous squirrel)
z"(t)  = -g,  weight acceleration (constant)
```

and the solution is a parabolic function z(t) obtained by double integration :

```
z’(t) = -1.g.t
z(t)  = -1/2.g.t2+ h
```

0232 : the rolling marble

The trajectory of a marble rolling without friction and without skidding over a curved surface in

gravity-free space is called a geodesic ; this curve is such that the thrust exerted by the marble on the surface is always perpendicular to the surface at point of contact, equal and opposed to the reaction of the surface, cancelling any tangential force. This property can be expressed mathematically by writing the marble's acceleration as a vector orthogonal to the surface, but some differential geometry is needed to do so (a lot, in fact).

Any point M of a surface is defined in space by the following expressions :

```
M(u,v) = [ x(u,v), y(u,v), z(u,v) ]     with u,v € [0,1]
```

expressions known as surface parametric equations. The same point can also be defined as belonging to a curve immersed in the surface in the form of the following parametric equation :

```
M(t)   = [ u(t), v(t) ]    with t € [0,1]
```

The first derivative (speed) of M(t) in relation to t is expressed in function of the partial derivatives ∂uM and ∂vM at point M(t) on the surface :

```
dM/dt  = ∂M/∂u.du/dt + ∂M/∂v.dv/dt
```

or to simplify writing :

```
M'     = ∂uM.u' + ∂vM.v'
```

the second derivative (acceleration) is then calculated :

```
M"     = dM'/dt
       = d(∂uM'.u' + ∂vM'.v')/dt
       = d(∂uM'.u')/dt + d(∂vM'.v')/dt
       = (∂2uuM.u'+ ∂2uvM.v').u' + ∂uM.u"
       +(∂2uvM.u'+ ∂2vvM.v').v' + ∂vM.v"
       = ∂uM.u" + ∂vM.v"
       + ∂2uuM.u'2 + 2.∂2uvM.u'.v' + ∂2vvM.v'2
```

Knowing that the partial derivatives ∂uM and ∂vM define the vectors tangent to the surface in M(t), all that remains is to express the orthogonality of these vectors with vector acceleration by cancelling their scalar products :

```
M"•∂uM = 0,
M"•∂vM = 0
```

which is the differential system that has to satisfy M(t) for the curve followed to be a geodesic. We do not know how to incorporate this system in the general case, and even for a majority of cases ; only approached solutions can be obtained and the analysis of the properties of these geodesic curves is extremely complex. This is a real shame when you consider that another property of these geodesics is that they are the quickest route between any two points on a surface (under certain conditions of proximity) and they are to curved surfaces what straight lines are to euclidian space, the fundamental geometric entity from which everything, or almost everything, is derived, polygons, triangles, the definition of angles, parallelism, etc... Curved surfaces appear to form a world that is virtually

inaccessible to exploration, if we exclude the computer approach, in which there is more to look at than actually understand.

0233 the sliver of soap

Nature offers us all kinds of equilibrium surfaces, funicular curves, and minimal surfaces. The equilibrium surface of a sliver of soap hung on a wire is such that each point is placed in a mean position in relation to its neighbours; to begin with, let's imagine a « discrete » - discontinuous - mesh projected on a square (x,y) and write the altitude z(i,j) of each point as equal to the arithmetic mean of the altitude of the points in a cross :

```
z[i,j] = ( z[i-1,j] + z[i+1,j] + z[i,j-1] + z[i,j+1] ) / 4
```

also written as :

```
z[i-1,j] + z[i+1,j] + z[i,j-1] + z[i,j+1] - 4.z[i,j] = 0
```

or even :

```
( z[i-1,j] - 2.z[i,j]+ z[i+1,j] ) +
( z[i,j-1] - 2.z[i,j]+ z[i,j+1] ) = 0
```

an expression in which we recognise the sum of two second-order finite differences, leading, when we shift to a continuous mesh, to a partial derivative equation known as a Laplace equation :

```
Δ( z ) = ∂2z/∂x2 + ∂2z/∂y2 = 0
```

This amazingly simple equation (not forgetting the discrete form that is nothing more than an arithmetic mean) is fundamental to mathematical physics, it features in the study of numerous complex problems, from electromagnetism to elasticity. We might ask how such a simple expression can describe the complex forms that a sliver of soap can take. Alan Turing, an English mathematician and logician, considered as one of the founding fathers of computer science and artificial intelligence, apparently used to say « Science is a differential equation. Religion is a boundary condition » . This perhaps explains why the equation, studied by generations of mathematicians has found no solution in the general case, although simply expressible in implicit or parametric form. So we are a long way from having the tools that would be so useful in researching the relations between the conics and geodesics of a minimal surface, to use an example that will be examined later !

Note : open a spreadsheet that uses iterative calculus and can display surfaces ; enter the first formula in a table of say 20x20 cells ; so there are 400 formulas displaying the value zero. In the peripheral cells, replace the formulas with zero values, then somewhere near centre left, enter the value +1 and toward centre right the value -1. Display the corresponding graph, choosing surface type, and watch : a lovely minimal surface comes up stretched on a horizontal rectangular frame, pulled up to the top left and down to the bottom right. By making the « boundary conditions » more complex and using a little imagination, you should manage to recreate the Sagrada Famillia ;-) Alan Turing was right !

03 first attempt at classification

A first rough classification can be made on the basis of this quick overview, by dividing the forms into three families according to « hardness » : solid forms, flexible forms and elastic forms.

031 solid forms

Solid forms are such that any deformation destroys the form ; a sphere in which one point is modified is no longer a sphere, it is no longer a form in which the points are equidistant from the centre and thus no longer complies with its first definition, it is no longer even a continuous surface with a tangent plane at each point. Conics and most surfaces defined by implicit equations behave in the same way, and are actually difficult to manipulate, modify and combine continuously with other surfaces - the fuselage of a plane built with these forms is merely a series of joins of varying strength, with a whole set of consequences in terms of aerodynamics (turbulences), the resistance of the materials (stress concentration) and look (discontinuity of reflections).

032 flexible forms

Flexible forms, on the other hand, will tolerate partial, non destructive deformation, using the control points ; so far, we have mentioned only three forms controlled by a limited set of points, the segment, the hyperbolic paraboloid (HP) and the parabola. Whatever transformations are made to the three points controlling a parabola, it remains a parabola ; the HP will likewise shift from a very flat orthonormed square to a slender, saddle-shaped curved form, finally becoming a curvilinear triangle (the curved side is a parabola) well known by children who play with strings on two lines of points, all this without losing the slightest property. Gaudi's Sagrada Familia in Barcelona is a complex composition of slender hyperbolic paraboloids and parabolas that closely resemble the organic forms dreamed of by the Catalan architect. In chapter 3 we will take a closer look at how to build the fuselage of a plane, from the conic nose to the tailplane via the pear shape following the line of the cockpit, continuously and harmoniously, with the advantages already hinted at in terms of aerodynamic, structural and esthetic properties.

033 elastic forms

Elastic forms are constructed from boundary constraints by maintaining internal equilibrium ; geodesics, minimal surfaces, and Laplace's equation have already been mentioned, along with the difficulty involved in treating this type of geometry. Yet these are precisely the optimal forms in nature, forms that we should be able to understand and really master, and it is for these forms that we have a minimum of easy-to-handle tools.

The present study will focus more particularly on flexible forms, approached in the simplest and most unitary way to show the bridges that can be built between them, with solid surfaces on one side and elastic surfaces on the other. The aim is to try to find a precise or approached description of solid and

elastic forms, starting from flexible forms. The arc of a circle redefined using a flexible curve could finally be transformed into a nice pear shape, or a cam or a drop of water, then warp in space, embed itself in a surface, a surface that slowly tends toward the equilibrium of a sliver of soap.

Having run through this reminder on geography, we are now going to ask how to combine points in space, and first of all, how to construct the midpoint between two points.

1 construction, definition

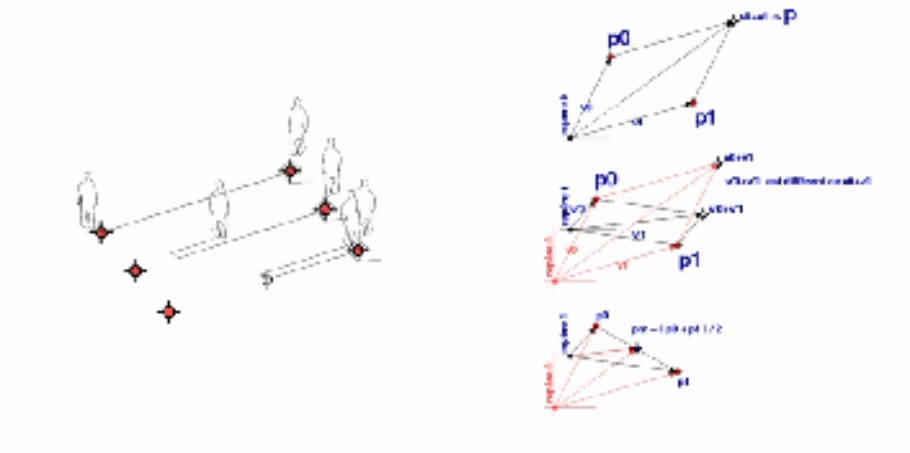
Summary of this section :

This chapter presents a gradually approach to some curve forms, highlights the unitary character of forms that appear to differ greatly - like the cube and the ... cubic - leading to the general definition of pForms.

We propose using a set of points to construct a family of forms (curves, surfaces, volumes, hypervolumes, …) initially generated by the recursive application of a simple operation :

« construct the midpoint of two points »

Let us construct the midpoint between two points avoiding the use of a ruler and compass. On a very flat floor, for example, one method is to stretch a rope between two posts, trace the right segment passing by these two posts with a piece of chalk, then bring the second end of the rope back to the first, following the line left by the chalk, folding the rope into two equal parts whose new end is now at the midpoint of the segment.



figures 1.1 and 1.2 : finding the midpoint between two points, an elementary gesture ; the sum of two points does not produce an invariant point, but a half sum does.

On a curved surface the problem is a little more complex, there is an infinite number of curves linking the two posts and it is advisable to stretch the string by placing it on the curve following the only curved curve, or geodesic, constituting the shortest route between these two points and along which the normal to the cord (in the osculatory plane) is colinear with the normal to the surface. Both cases work on the hypothesis that the string maintains constant length and is infinitely flexible. By avoiding the prior use of a classic construction of the midpoint with a ruler and compass, this principle of constructing the midpoint remains valid for any surface, and this will be the starting point of a generalisation of the forms studied in euclidian space to forms belonging to curved spaces, via the introduction of the immersed segment concept.

But how do we write the midpoint of two points, and any combination of points beyond that ? The reference space is « current » space populated by points that hopefully will be able to be combined according to a linear expression of the type:

$$\sum_i \; ki.pi, \quad \text{with } i = [0,n-1]$$

Combining vectors linearly is not a problem, ... in a vectoral space ; but point space (called affine space) does not have all the properties of a vectoral space, even if it resembles it and, notably, not all the linear point combinations are valid. It is possible, for instance, to define the sum of two vectors independently of the apex, but this cannot be done for two points; on the other hand, the half-sum of two points will produce an invariant point with a change of apex thus constituting a weighted and valid linear combination. Generally speaking, we show that any linear combination of points is valid if it satisfies the condition that :

$\sum_i$ ki = 1, with i = [0,n-1]

Notably correct will be expressions like :

```
p = (p0 + p1)/2
p = 2.p0 - p1
p = (2.p0 + p1)/3
p = (p0 + 3.p1 + 3.p2 + p3)/8
```

which respectively represent the midpoint of two points, the symetrical point of p1 in relation to p0, the point at a third of segment p0p1 and the midpoint of a cubic defined by the four points (p0,p1,p2,p3). You may have noticed that all this is simply another expression of the well known theorems on barycentres...

Having stated this clearly, we can construct the first family of « recursive multilinear forms » constructed with a pair of operators MI() and MIR(), extend it by applying a « diagonalisation », DIAG() operator, and finally propose the fundamental definition of pascalian forms.

11 recursive multilinear forms

Summary of this section :

By applying a pair of operators MI() and MIR() to a set of points we are able to naturally generate the first linear shapes in geometry : right segments, (curved) facets, (curved) cubes, (curved) hypercubes, and beyond, if necessary.

111 a point in R4

The linear combinations envisaged above can be expressed in an affine space of any Rn dimension. For reasons we will go into later, we will operate on points that are always defined in R4, using the so-called homogeneous coordinate, or projective form: <x,y,z,t>, associating the R3 point defined as <x/t,y/t,z/t> with the R4 point. To begin, t=1 will be used by default, forgetting that we are working in R4 space.

In POVRAY/pFlibs syntax, a current point (x,y,z) is defined by stating a local variable P and giving it

3 coordinates x, y, z (and t= 1) between brackets '<' and '>' :

```
#local P = <x,y,z,1>;
```

The macro calls :

```
draw( 0, <0.0,0.0,0.5,1.0>, STANDARD )
draw( 0, <0.0,0.0,0.0,1.0>, point(0.1)  + ma_couleur(<1,1,0,0.5>))
draw( 0, <0.0,0.0,-0.5,1.0>, point(0.1) + ma_texture(GOLD) )
draw( 0, <0.0,0.0,0.0,1.0>, point(0.1)  + ma_texture(MIROIR) )
draw( 0, <-0.5,0.0,0.0,1.0>, point(0.1) + ma_texture(GRANIT) )
draw( 0, <0.5,0.0,0.0,1.0>, point(0.1)  + ma_texture(MARBRE) )
```

will draw six points respectively in the standard form of a red sphere with a radius of 0.05, and in forms that are gold, chrome, granite and translucent plexiglass with black stripes, with a radius of 0.2.

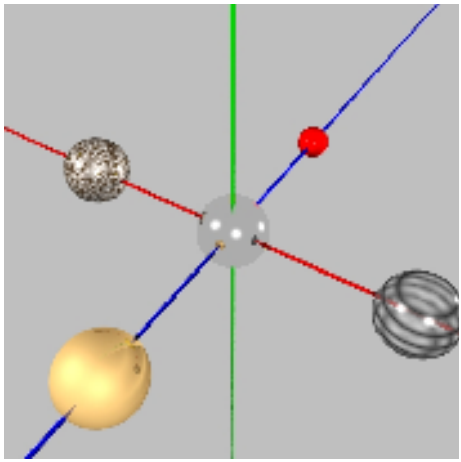


figure 111 : the 3 axes Ox, Oy, Oz (red, green, blue) and some points.

112 a straight segment

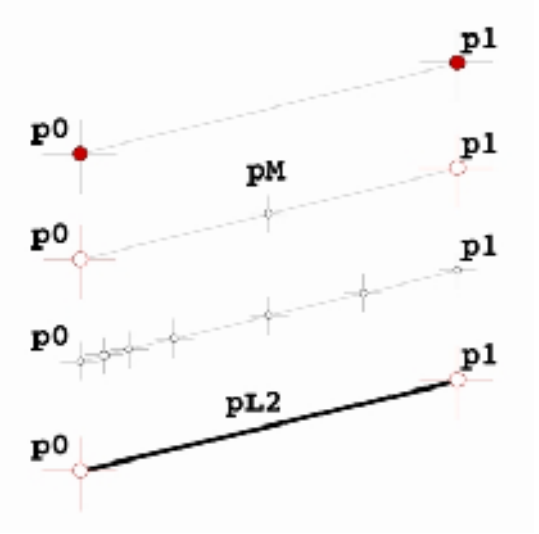


figure 112.1 : from the pair of points to the segment (pL2).

Let us call MI() the operator returning the midpoint of two points p0 and p1 of the space :

```
pm = MI(p0,p1) = (p0+p1)/2
```

Note that the sum of the coefficients is : 1/2 + 1/2 = 1. This third point pm gives rise to two pairs of points (p0,pm) and (pm,p1), and to the irresistable desire to start back in the direction of p0 and p1 :

```
p0m    = MI(p0,pm)          pm1   = MI(pm,p1)
p00m   = MI(p0,p0m)         pmm1  = MI(pm,pm1)
p0mm   = MI(p0m,pm)         pm11  = MI(pm1,p1)
etc…                        etc…
```

Recursive application of operator MI() will produce an infinite and « dense » set of points situated between the two points p0 and p1, a « right segment » that we will designate as pL2 (the reasons for this notation will be given later in the text) ; if we agree to call MIR() the recursive operator thus defined, we will write :

```
pL2 = MIR(p0,p1)
```

This set of points (pL2) is not strictly speaking a right segment such as is usually defined in geometry. While the recursive process does lead to a set of points of the desired number, with a distance

between each point tending to zero, the set obtained is not continuous and is not « really » a right segment. A right segment is usually defined in Rn linking a pair of points (p0, p1) like all points that satisfy bijective linear application :

```
p(t) = (1-t)*p0 + t*p1,
       with t in the interval [0,1] of the set of real numbers (R)
```

This set of points is finite and continuous (like R), and points like p(1/3) or p(k) with k = root(2)/2, can indeed be reached on this right segment; which is not the case for pL2 that is in a bijective relationship to all the partitions of unity (1/2, 1/4, 1/8,...). Such a set is said to be dense, and it really is this property of density that interests us, notably because it brings us back to the indispensable concept of tangent. This will be dealt with in more detail in section 2, and our pL2 will subsequently be assimilated to a « real » right segment and the derived forms (surfaces, volumes,...) to the classic continuous forms in geometry.

In POVRAY/pFlibs syntax, we define and draw a two point array thus :

```
#local pL2 = array[2] { p0, p1 } // array of two points
draw( 1, pL2, STANDARD )         // 1 is the dimension of the segment
```

Note that pL2 IS quite simply THIS two point array. The macro call :

```
#local pL2_subdivisee = pFsubdivision( 1, pL2, 4 )
```

implement operator MIR(), by applying MI() to the pL2 point array 4 times, to produce a series of 2^4+1 = 17 points that will be drawn by the following call :

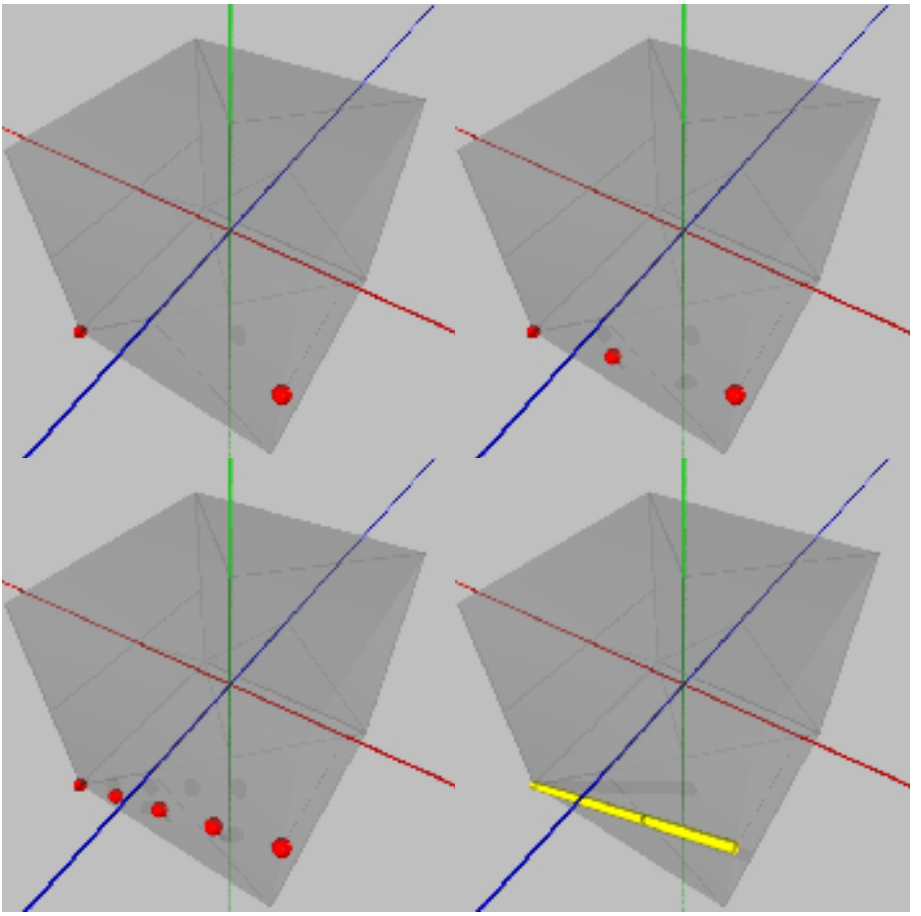
```
draw( 1, pL2_subdivisee, STANDARD )     // 17 small red spheres
```

Subdivision can be incorporated in the drawing call more directly by inserting the finesse() operator :

```
draw( 1, pL2, finesse(4) )              // 17 small red spheres
```

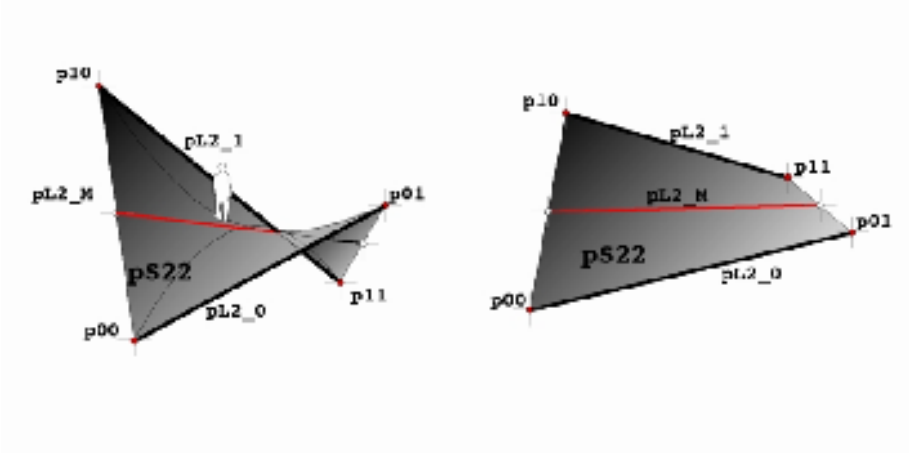
and the spheres can even be replaced by cylinders to obtain a yellow segment with a 0.02 radius by writing :

```
draw( 1, pL2, finesse(4) + courbe( 0.02 ) + couleur(<1,1,0>) )
```



figures 112.2 to 112.5 : a segment (pL2) constructed on two points.

113 a curved facet



figures 113.1 and 113.2 : two representations of a curved facet (pS22).

Given two segments pL2_0 and pL2_1 constructed on points (p00,p01) and (p10,p11). Considering the segment constructed on the respective midpoints of the extremities of segments pL2_0 et pL2_1 :

```
pL2_M = MIR( MI(p00,p10), MI(p01,p11) )
```

and playing on the commutativity oflinear operators MI() and MIR(), leads to understanding the definition of operator MI() thus :

```
pL2_M  = MI( MIR(p00,p10), MIR(p01,p11) )
       = MI( pL2_0, pL2_1 )
```

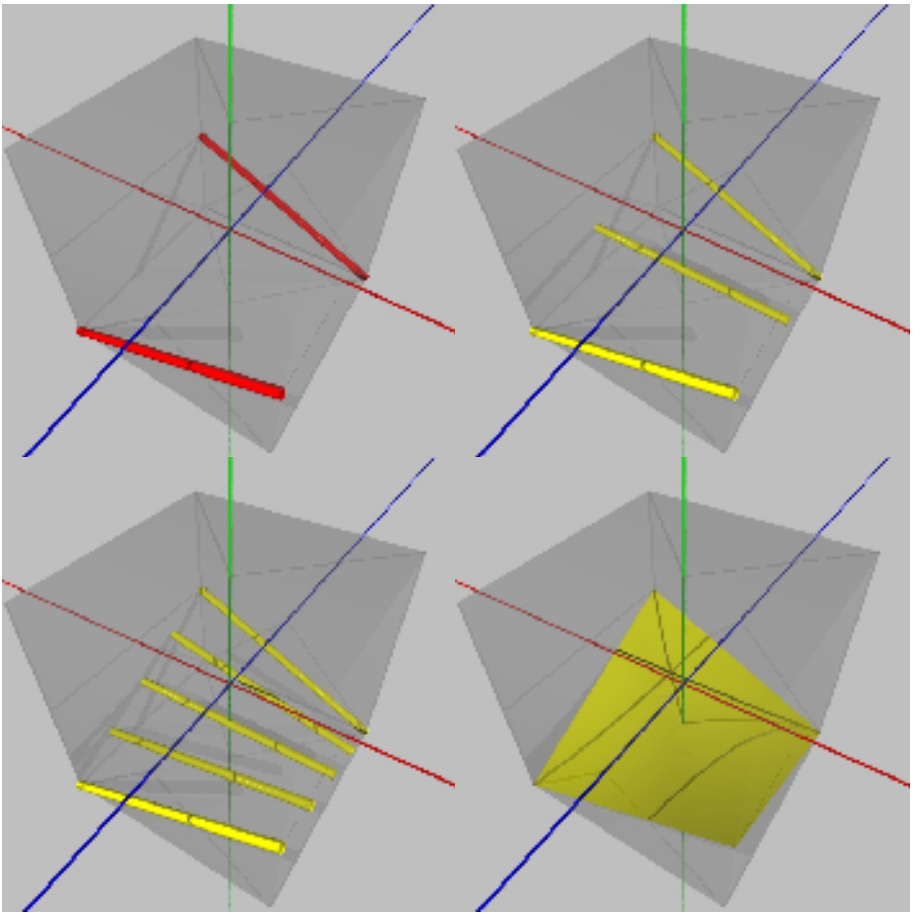
Recursive application of operator MI() to pairs of segments to the left (pL2_0,pL2_M) and right (pL2_M,pL2_1) produces an infinite and dense set of segments forming a portion of the surface, a curved facet that we will designate as pS22 ; extending the application of operator MIR() to the case of two segments, we will write :

```
pS22 = MIR( pL2_0, pL2_1 )
```

In POVRAY/pFlibs syntax, you write :

```
// creation of an array of two segments :
#local pS22 = array[2] { pL2_0, pL2_1 }
// drawing of 2*2 = 4 small red spheres:
draw( 2, pS22, STANDARD ) // 2 is the dimension of the surface
```

```
// recursion of level 2 in both dimensions:
#local spS22 = pFsubdivision( 2, pS22, <2,2> )
// drawing of (4+1)*(4+1) = 25 small red squares:
draw( 2, spS22, STANDARD )
// variant: recursion 2 in both dimensions
// and drawing of a smooth yellow surface:
draw( 2, pS22, finesse(<2,2>)+surface(LISSE)+couleur(<1,1,0>) )
```



figures 113.3 to 113.7 : curved facet (pS22) constructed on two segments (pL2).

114 a curved cube

Following the progression and redefining operator MI() to produce the midpoint facet between two curved facets (pS22_0 et pS22_1) and operator MIR() for its recursive application, we can construct an infinite and dense set of facets, a curves mille–feuilles, a volume that we'll call a « curved cube », noted pV222.

```
pS22_M = MI( pS22_0, pS22_1 )
pV222  = MIR( pS22_0, pS22_1 )
```

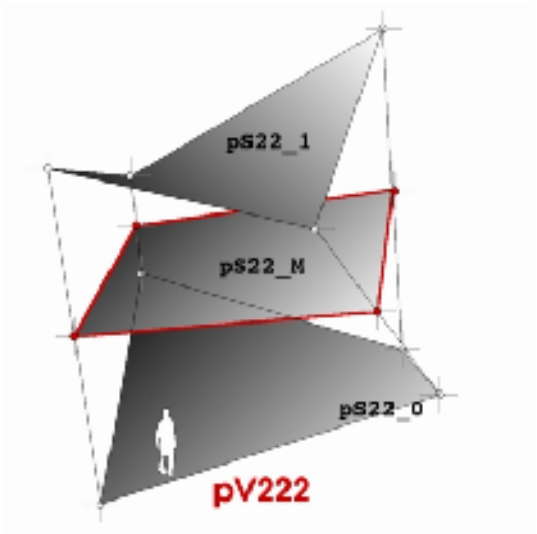
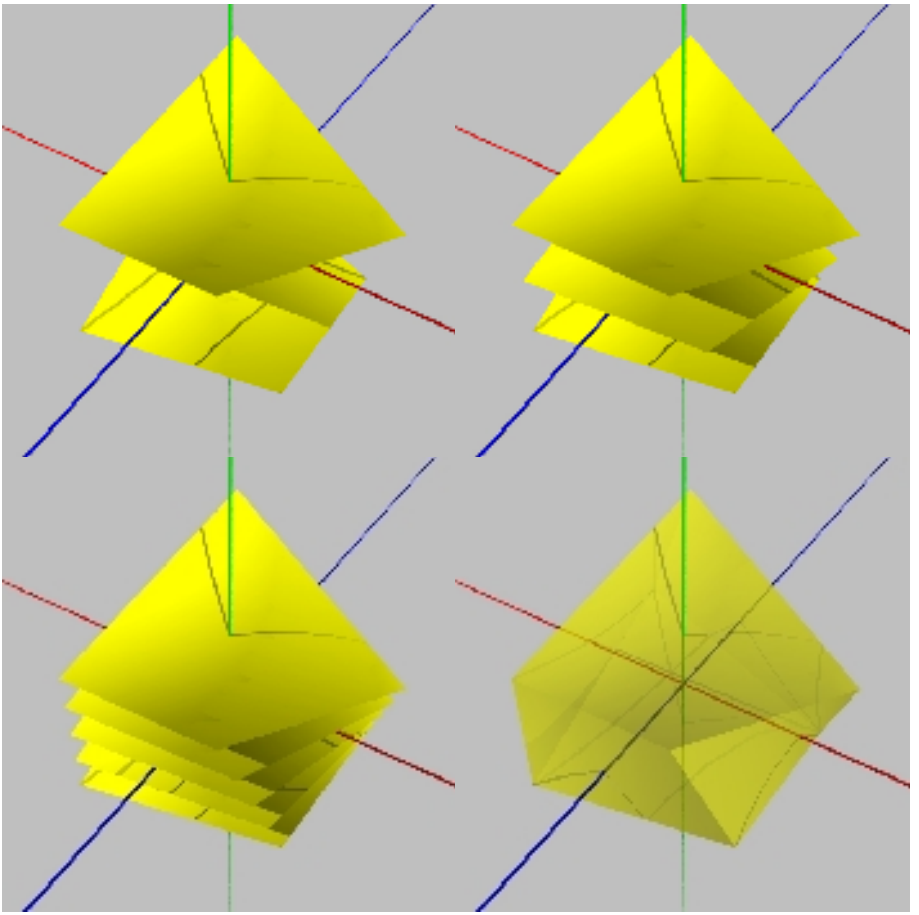


figure 114.1 : midpoint facet of two curved facets (pS22).

In POVRAY/pFlibs, syntax you write :

```
// creation of an array of two surfaces:
#local pV222 = array[2] { pS22_0, pS22_1 }
// drawing of 2*2*2=8 small red spheres:
draw( 3, pV222, STANDARD )
// recursion 2 in the three dimensions
// and drawing of the 6 plane surfaces of a curved cube:
draw( 3, pV222, finesse(<2,2,2>)+enveloppe(LISSE)+couleur(<0,1,1>))
```

to construct and draw the eight apexes of a curved cube and its cyan envelope.



figures 114.2 to 114.5 : curved cube (pV222) constructed on two curved facets (pS22).

Just as a curved facet can be represented in several forms (square matrix of points, segment distribution or contiuous surface), a curved cube can be represented in several aspects : mille-feuilles of curved plane surfaces, tridimensional matrix of points, bundle of segments, or usually in the form of an envelope made up of its six curved plane surfaces. In all cases, a curved cube is above all a tridimensional table of points, a real volume object that can be cut open to extract other objects (surface, curves, points).

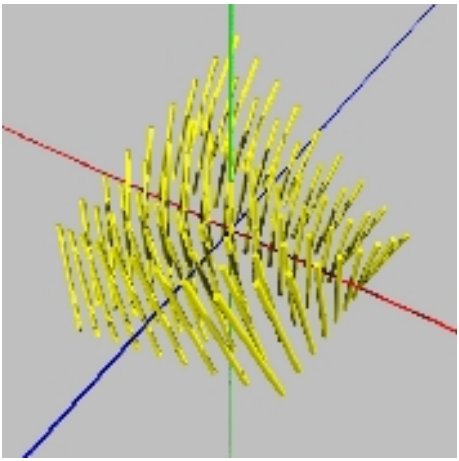


figure 114.6 : fibered representation of a curved cube.

115 a curved hypercube

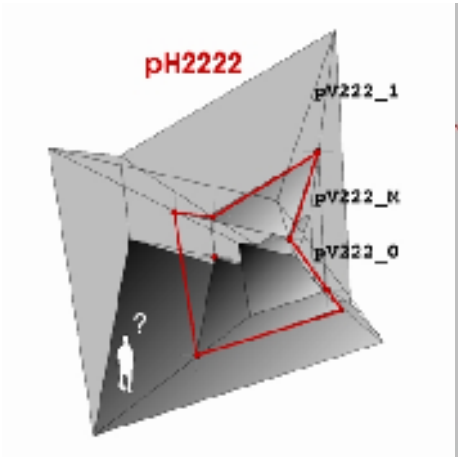
Lastly, by redefining operators MI() to produce the midpoint curved cube between two curved cubes (pV222_0 and pV222_1) and MIR() for its recursive application, we can construct a curved hypercube.

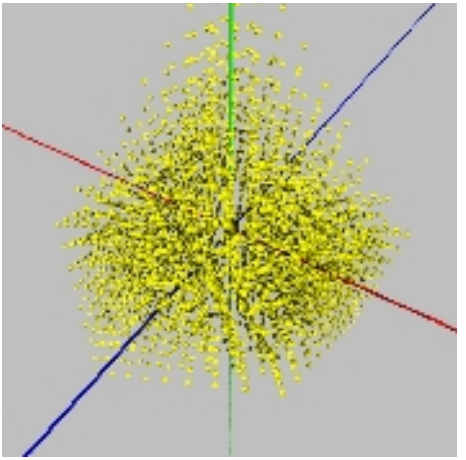
```
pV222_M = MI( pV222_0, pV222_1 )
pH2222  = MIR( pV222_0, pV222_1 )
```

In POVRAY/pFlibs syntax, you write :

```
// creation of an array of two volumes:
#local pH2222 = array[2] { pV222_0, pV222_1 }
// drawing of 2*2*2*2 = 16 small red spheres:
draw( 4, pH2222, STANDARD )
// drawing of (2*2+1)^4 = 625 small yellow spheres:
draw(4,pH2222,finesse(<2,2,2,2>)+point(0.02)+couleur(<1,1,0>))
```

to construct and draw the sixteen red apexes of the hypercube with a filling of small yellow spheres, until we find a more eloquent representation than image 115.2 below ...





figures 115.1 and 115.2 : midpoint cube of two curved cubes (pV222) ; a hypercube « cloud ».

116 first generalisation

The pair of operators MI() and MIR() thus produces a family of linear shapes obtained by a recursive process. As a general rule, we note that operator MI() produces a shape of the same dimension and that operator MIR() produces a form of a bigger dimension. These forms are generally curved (apart from the segment of course), the curved facet is a quadrangular portion of the classic hyperbolic paraboloid, a ruled surface with double negative curvature, and the six sides of a curved cube are curved facets.

These shapes are « full », each point of a shape can be adressed : a curved cube is a volume filled with points rather than just an empty envelope, and the same goes for the hypercube. So we have in fact defined coherent forms in which it will be possible to work and from which we can extract « sub-forms » for instance. An inverse operation, diagonalisation, producing a form whose dimension is smaller but more complex than a linear segment will lead us on to new extensions of operators MI() and MIR().

12 diagonalisation

Summary of this section :

All the preceding forms are linear many times, they are generated by families of straight lines, but this doesn't mean they contain fewer « true » curves presenting a certain interest. The curved facet, for instance, is a saddle-shaped form presenting a certain curvature as soon as the two generating segments cease to be coplanary. By folding a curved facet back on itself, a kind of triangle is made, one side of which is the arc of a parabola. This « folding » of the facet allows us to construct a curve, a form of smaller dimension to that of the facet, but of greater complexity than the right segment.

121 the curved facet and its parabola

Given a curved facet constructed on two segments (pL2_0,pL2_1) defined on two pairs of points (p00,p01) and (p10,p11) :

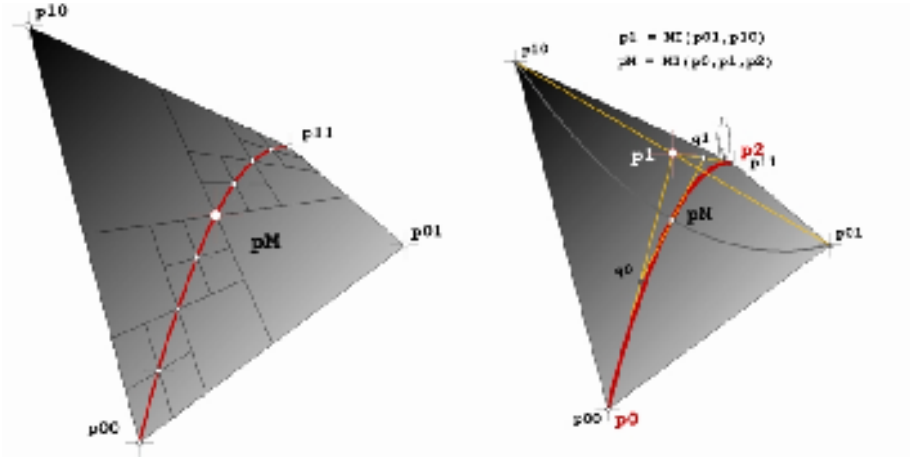
```
pS22   = MIR( pL2_0, pL2_1 )
       = MIR( MIR( p00, p01 ), MIR( p10, p11 ) ),
```

the midpoint of the facet can be defined as :

```
pm     = MI( MI( p00, p01 ), MI( p10, p11 ) )
       = ( (pOO + p01)/2 + (p10 +p11)/2 )/2
       = ( pOO + p01 + p10 + p11 )/4
```

This point determines four sub-facets (p00,pm), (p01,pm), (p10,pm) and (p11,pm). By iterating the operation recursively on the two diagonal sub-facets constructed on pairs (p00,pm) and (pm,p11), an infinite and dense set of points is generated, a curve diagonally linking points p00 and p11 of the curved facet. Let DIAG() be used to name the operator that creates this diagonal line in the facet.

```
diagonal = DIAG( pS22 )
```



figures 121.1 and 121.2 : two views of the diagonal of a curved plane.

This diagonalisation produces a curve of a smaller dimension than that of the facet, leading to seeking a reduced set of « control » points extracted from the four points generating the facet.

By defining the three points :

```
p0     =   p00,
p1     = ( p01+p10 )/2,
p2     =       p11,
```

the midpoint can be rewritten in a form leading to considering an operator MI() applicable to three points :

```
pm     = ( p0 + 2*p1 + p2 )/4
       = MI( p0, p1, p2 )
```

This new operator actually consists of a twofold recursive application of the initial operator MI() leading to point pm from the triplet (p0,p1,p2), using two intermediary midpoints q0 and q1 :

```
   q0   = MI(p0,p1),
   q1   = MI(p1,p2),
   pm   = MI(q0,q1)
```

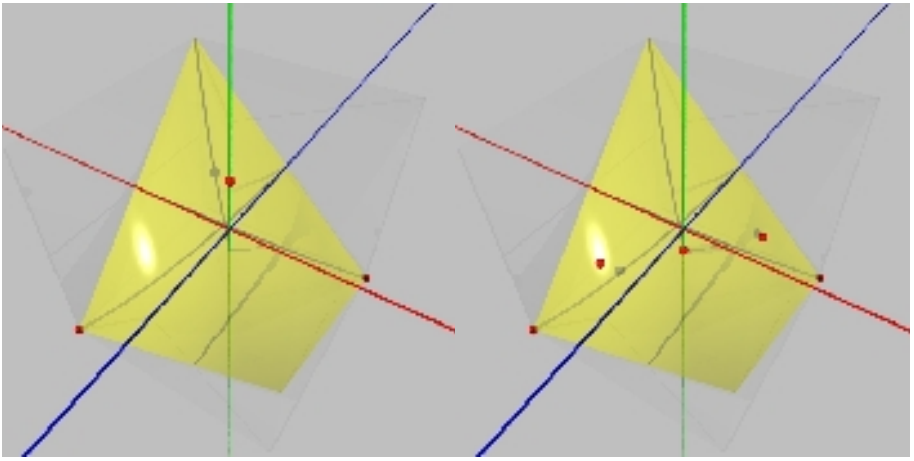
Hence we can consider it as a valid extension of operator MI().

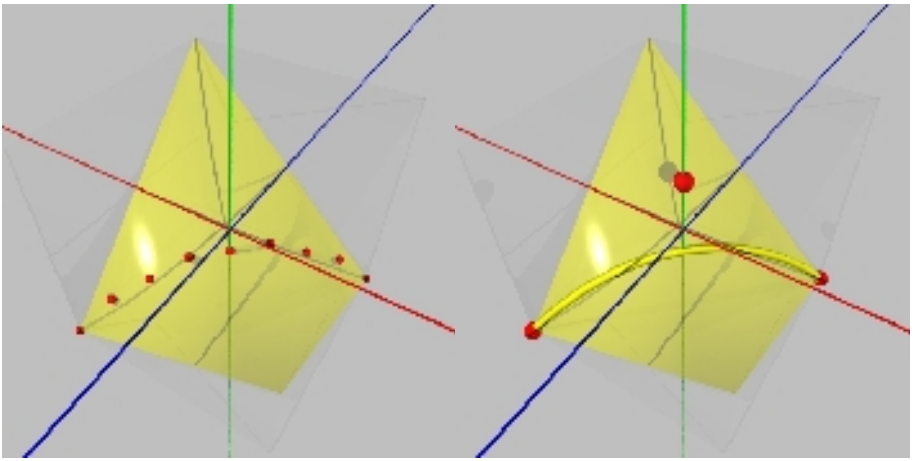
Furthermore, the recursive application of this operator with triplet points (p0,q0,pm) and (pm,q1,p1) reproduces the diagonal curve, that we will now decide to designate as pL3, extending the application of MIR() to the three point case, we write :

```
pL3    = DIAG( pS22 )
       = MIR(p0,p1,p2)
```

This construction brings us back to the basic algorithm proposed by de Casteljau in the case of a curve (parabola) defined by three points ; this is logically linked to the process of generating the recursive multilinear form family, via a kind of contraction/folding/diagonalisation of a curved facet (portion of a hyperbolic paraboloid). This reformulation by applying the extended operators MI() and MIR() allows us to leave the purely rectilinear world of recursive multilinear forms and to really confront that of curved forms. In POVRAY/pFlibs syntax, you write :

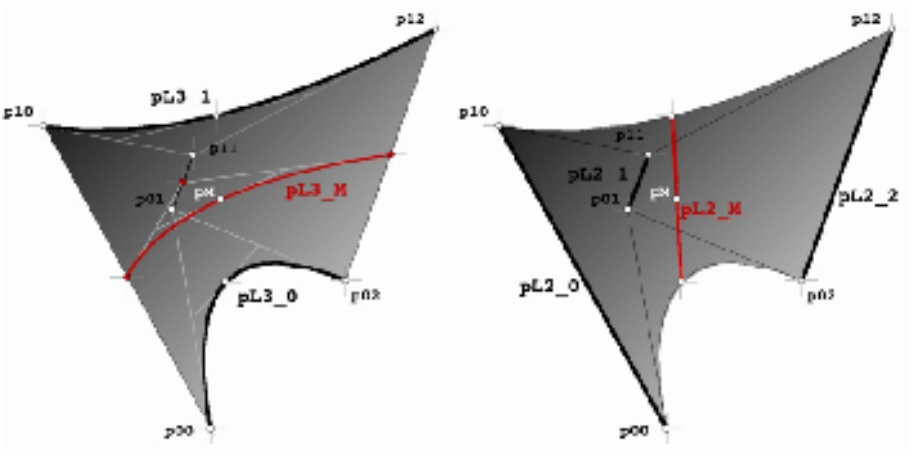
```
// definition of a curved facet and display:
#local pS22 = array[2] { pL2_0, pL2_1 }
draw( 2, pS22, refinement()+surface(SMOOTH)+my_colour(<1,1,1>) )
// extraction of the diagonal control points:
#local p0 =   pS22[0][0];
#local p1 = ( pS22[0][1]+pS22[1][0] ) / 2;
#local p2 =               pS22[1][1];
// definition of the parabola and display:
#local pL3 = array[3] { p0, p1, p2 }
draw( 1, pL3, STANDARD ) // control points
draw( 1, pL3, finesse(5)+courbe(0.02)+ ma_couleur(<1,1,0>) )
```





figures 121.3 to 121.6 : parabola (pL3) constructed on 3 points, diagonal of a curved facet (pS22).

122 the ruled paraboloid and its cubic



figures 122.1 and 122.2 : construction of a ruled paraboloid (pS32) from two parabolas (pL3), and from three segments (pL2).

We know how to construct a parabola. Let two parabolas pL3_0 and pL3_1 be constructed on points

(p00,p01,p02) and (p10,p11,p12). Considering the parabola constructed on the respective midpoints of points defining the two parabolas :

```
pL3_M  = MIR( MI(p00,p10), MI(p01,p11), MI(p02,p12) )
```

we are led to extending the definition of operator MI() thus :

```
pL3_M  = MI(pL3_0,pL3_1)
```

Recursive application of operator MI() to parabola pairs (pL3_0,pL3_M) and (pL3_M,pL3_1) produces an infinite and dense set of parabolas, a ruled paraboloid, that we will designate as pS32 ; extending the application of MIR() to the case of two parabolas, will be written as :

```
pS32   = MIR(pL2_0,pL2_1)  // pS32 or pS23
```

Note that this surface can also be obtained from three segments constructed on (p00,p10), (p01,p11), (p02,p12), leading to the extension of operators MI() and MIR() to the case of three segments :

```
pL2_M  = MI(pL2_0,pL2_1,pL2_2),
pS32   = MIR(pL2_0,pL2_1,pL2_2)
```

From this last point of view and to the extent of our knowledge, this can be considered as a new extension of de Casteljau's algorithm applied to three segments, rather than to three points as is usually presented, ... at least in the case of three points.

The same reasoning that underlies constructing the diagonal of a curved facet applies to a ruled paraboloid to produce a midpoint defined by four points :

```
p0     = p00,
p1     = (2*p01 + p10)/3,
p2     = (p02 + 2*p11)/3,
p3     = p12,
pm     = (p0 + 3*p1 + 3*p2 + p3)/8
       = MI(p0,p1,p2,p3)
```

and a diagonal curve (cubic) :

```
pL4    = DIAG( pS32 )
       = MIR(p0, p1, p2, p3)
```

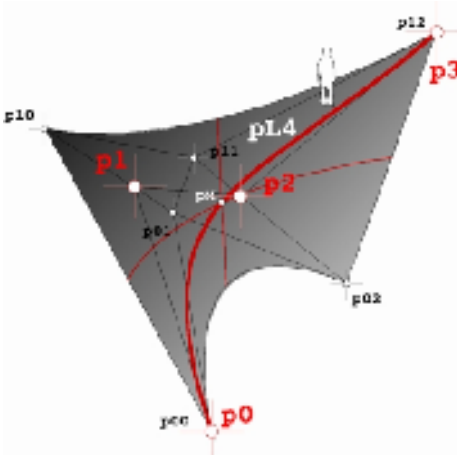


figure 122.3 : ruled paraboloid (pS32) and its diagonal (pL4) constructed on 4 points.

In POVRAY/pFlibs syntax, you write :

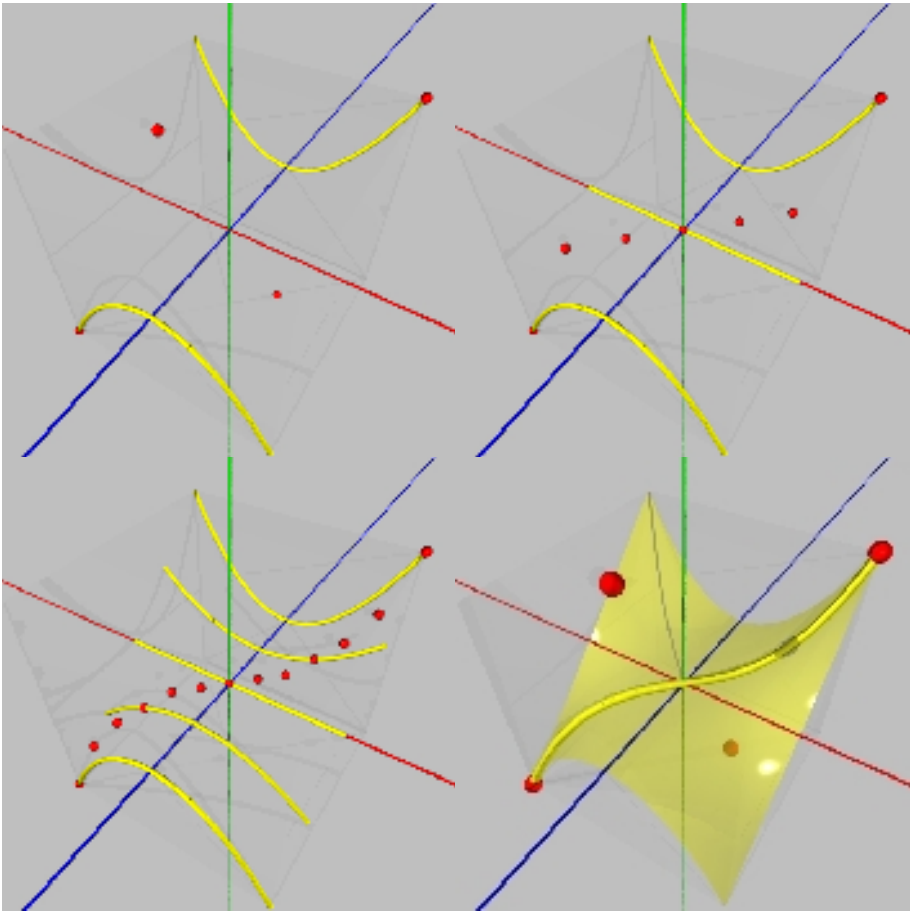
```
/ definition of a ruled paraboloid from
// 3 segments or 2 parabolas, and display:
#local pS32 = array[3] { pL2_0, pL2_1, pL2_2 }
            = array[2] { pL3_0, pL3_1 }
draw( 2, pS32, refinement() + surface(SMOOTH) + my_colour() )
// extraction of diagonal control points:
#local p0   =    pS32[0][0];
#local p1   = (2*pS32[0][1]+  pS32[1][0]) / 3;
#local p2   = (  pS32[0][2]+2*pS32[1][1]) / 3;
#local p3   =                 pS32[1][2];
// definition of the cubic and display:
#local pL4  = array[4] { p0, p1, p2, p3 }
draw( 1, pL4, STANDARD ) // control points
draw( 1, pL4, finesse(5)+courbe(0.02)+ma_couleur(<1,1,0>) )
```

Comment 1 : the cubic is often used in computer graphics to approximately represent the arc of a circle at 90°, -cf figure 122.8-; in part 31 we will see how to get an exact representation of a circle.

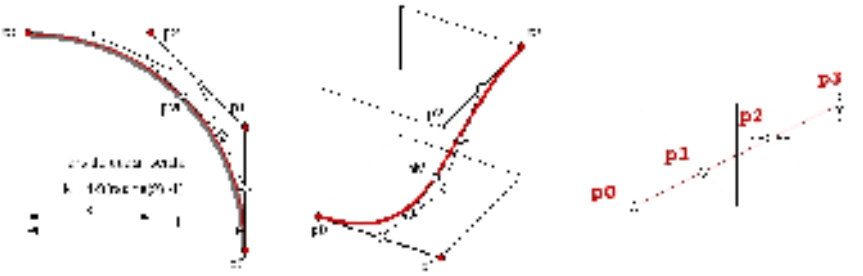
Comment 2 : the cubic is the first curved curve we have encountered so far ; we will see its use in part 34 on Concatenations. Figure 122.9 illustrates the occupation in space of a cubic linking two opposite points of a cube.

Comment 3 : Figure 122.10 represents the diagonal cubic of a paraboloid constructed on a grid orthonormed by its control points ; let us say that this is an orthonormed representation of the surface and

its diagonal. It is easy to visualise positions at 1/3 and 2/3 of points p1 and p2.



figures 122.4 to 122.7 : progressing toward the ruled paraboloid (pS32) and its diagonal (pL4).



figures 122.8 to 122.10 : flat cubic approximating the arc of a circle, and cubic in space ; orthonormed representation of a curved facet (pS32) and positioning of the 4 control points of its diagonal (pL4) .

123 the curved cube and its diagonals

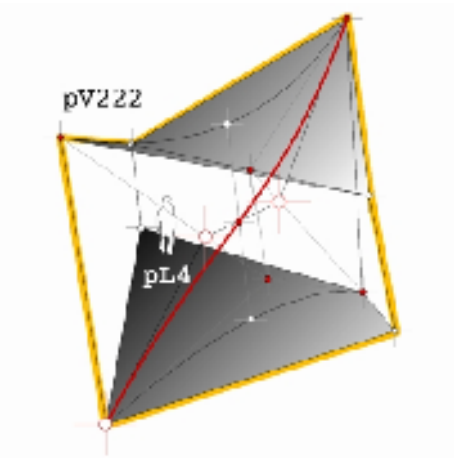


figure 123.1 : curved cube (pV222) and its diagonals, ruled paraboloid (pS32) and cubic (pL4).

The ruled paraboloid (pS32) constructed on the diagonal parabolas of the curved facets generating a cube, can be considered as the diagonal surface of the curved cube ; we saw that a cubic (pL4) could be considered as the diagonal of a ruled paraboloid (pS32), and thus as the « square » diagonal of the cube :

```
DIAG( pV222 )       -> pS32
pL4    = DIAG( pS32 )
```

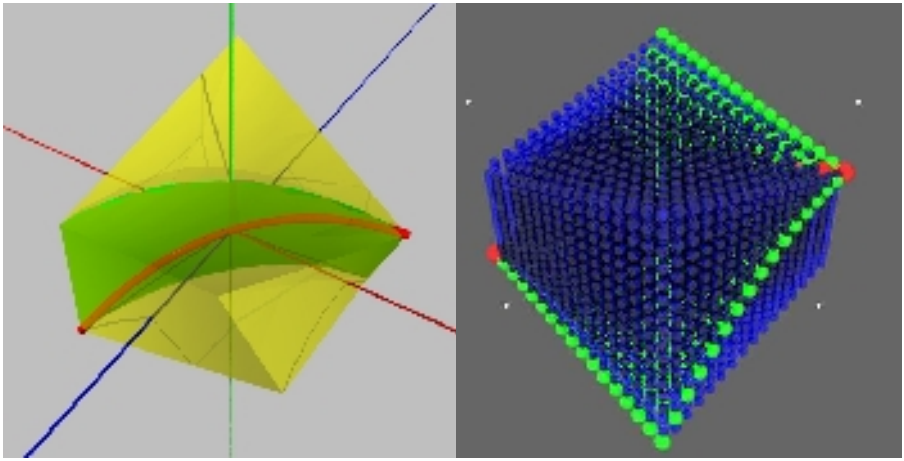
```
      = DIAG( DIAG( pV222 ) )
      = DIAG2( pV222 )
```

Likewise, the third diagonalisation of a curved hypercube pH2222 would lead to a curve controlled by 5 points, a pL5 :

```
DIAG( pH2222 )      -> pV322 // un cube contrôlé par 3 facettes
DIAG( pV322 )       -> pS33  // une surface contrôlée par 3 quads
pL5    = DIAG( pS33 )        // une courbe contrôlée par 5 points
       = DIAG( DIAG( pV322 ) )
       = DIAG( DIAG( DIAG( pH2222 ) ) )
       = DIAG3( pH2222 )
```

In POVRAY/pFlibs syntax, in the case of a cube, you write :

```
// definition of a curved cube and display:
#local pV222 = array[2] { pS22_0, pS22_1 }
draw( 3, pV222, finesse()+envelope(LISSE)+ma_couleur(<1,1,1>) )
// diagonal surface of pV222 and display:
#local pS32 = pFdiagonalisation( 3, pV222 )
draw( 2, pS32, finesse()+surface(LISSE)+ma_couleur(<1,1,0>) )
// diagonal curve of pS32 and display:
#local pL4 = pFdiagonalisation( 2, pS32 )
draw( 1, pL4, STANDARD ) // control points
draw( 1, pL4, finesse(5)+courbe(0.02)+ ma_couleur(<1,0,0>) )
```



figures 123.2 : two representations of the curved cube and its diagonals.

These new forms, notably surface pS33 (that we can decide to call biquadric by analogy with Bézier's bicubic square controlled by 4 cubics and called bicubic) and curve pL5, stem from the diagonalisations of a left hypercube pH2222 ; but they could be constructed more easily using operators MI() and MIR().

124 the biquadric and its diagonal

Two parabolas have allowed us to create a ruled paraboloid, three parabolas lead to a biquadric defined thus :

```
pS33   = MIR( pL3_0,pL3_1,pL3_2 )
       = MIR( MIR(p00,p01,p02),MIR(p10,p11,p12),MIR(p20,p21,p22) )
```

whose midpoint can be written as :

```
pm = MI( MI(p00,p01,p02), MI(p10,p11,p12), MI(p20,p21,p22) )
 = ( (p00+2p01+p02)/4+2(p10+2p11+p12)/4+(p20+2p21+p22)/4 )/4
 = ( p00+4(p01+p10)/2+6(p02+4p11+p20)/6+4(p12+p21)/2+p22 )/16
 = ( q0 + 4q1 + 6q2 + 4q3 + q4 )/16
 = MI( q0,q1,q2,q3,q4 )

with   q0 = p00
       q1 = (p01+p10)/2
       q2 = (p02+4p11+p20)/6
       q3 = (p12+p21)/2
       q4 = p22
```

leading to the construction of a diagonal curve defined by 5 points :

```
pL5    = MIR( q0,q1,q2,q3,q4 )
```

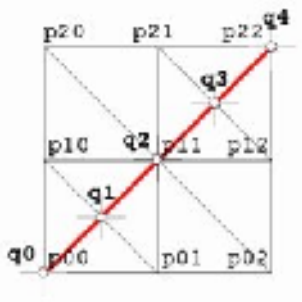
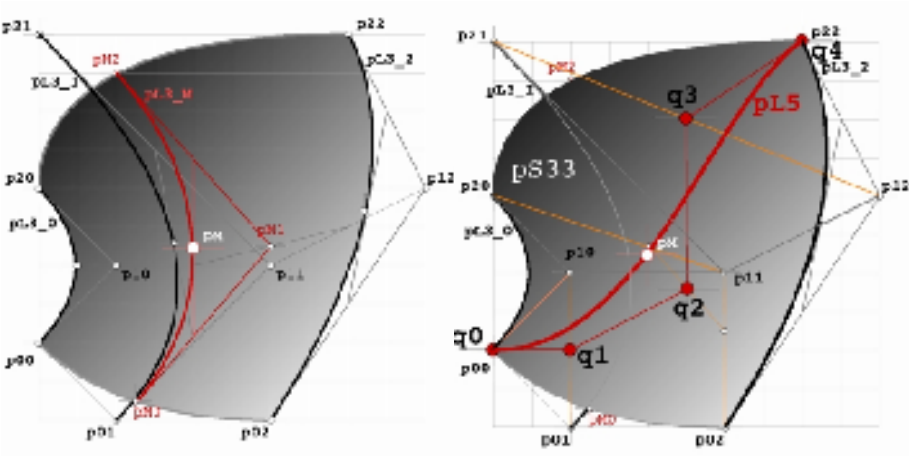


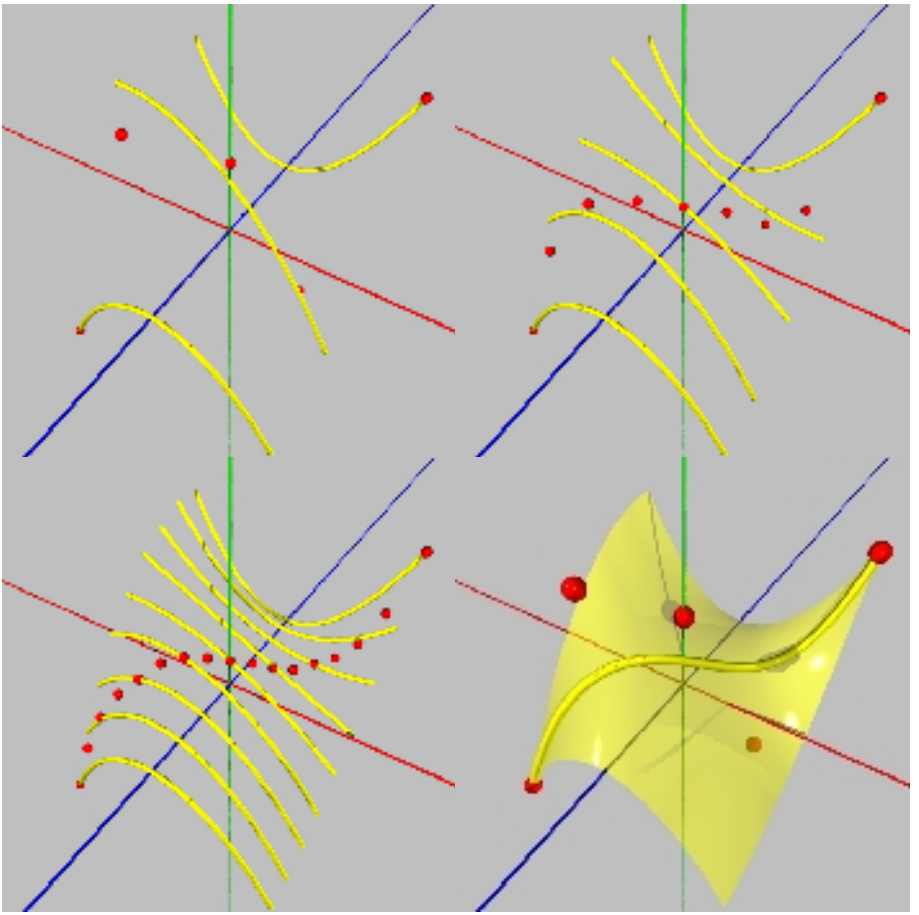
figure 124.1 : orthonormed representation of the biquadric (pS33) and the 5 control points of its diagonal (pL5).



figures 124.2 and 124.3 : biquadric (pS33) with its three generating (control) parabolas (pL3), and diagonal (pL5) of the biquadric with its 5 generating (control) points.

In POVRAY/pFlibs syntax, you write :

```
// definition of biquadric from 3 parabolas
// and display:
#local pS33 = array[3] { pL3_0, pL3_1, pL3_2 }
draw(2,pS33,finesse(<3,3>)+surface(LISSE)+ma_couleur(<1,1,1>))
// extraction of control points of the diagonal:
#local p0  =   pS33[0][0];
#local p1  = ( pS33[0][1]+  pS33[1][0]             ) / 2;
#local p2  = ( pS33[0][2]+4*pS33[1][1]+ pS33[2][0] ) / 6;
#local p3  = (              pS33[1][2]+ pS33[2][1] ) / 2;
#local p4  =                            pS33[2][2];
// definition of pL5 and display:
#local pL5= array[5] { p0, p1, p2, p3, p4 }
draw( 1, pL5, STANDARD ) // control points
draw( 1, pL5, finesse(5)+courbe(0.02)+ma_couleur(<1,0,0>) )
```



figures 124.4 à 124.8 : biquadric (pS33) defined by three parabolas (pL3) and its diagonale pL5.

Note that the biquadric (pS33) is the first, and so the simplest, non ruled curved surface in the family of pForms. The biquadric, and its diagonal (pL5), are particularly important for everything that follows.

13 generalisation : pFormes

Starting from the midpoint of a curved facet we applied a recursive process diagonally and created a parabola ; starting from the midpoint segment of a curved cube we applied the same recursive process diagonally to create a ruled paraboloid ; starting from the midpoint of a ruled paraboloid we likewise created a cubic. We noted the relation between these diagonals and the forms produced by operators MI() and MIR() redefined to apply to any number of forms, a new and generalised writing of the algorithm proposed by de Casteljau in 1959. By continuing to note a curve pL, a surface pS, a volume pV, etc…, followed by indexes to determine the number of generating forms, we have so far constructed the forms :

```
MIR( pP_0, pP_1 )                    -> pL2,segment
MIR( pL2_0, pL2_1 )                  -> pS22,facet
MIR( pS22_0, pS22_1 )                -> pV222,cube
MIR( pV222_0, pV222_1 )              -> pH2222,hypercube
DIAG( pS22 ) = MIR(pP_0,.., pP_2)    -> pL3,parabola
DIAG( pV222 ) = MIR(pL3_0, pL3_1)    -> pS32,ruled paraboloid
DIAG( pS32 )  = MIR(pP_0,.., pP_3)   -> pL4,cubic
```

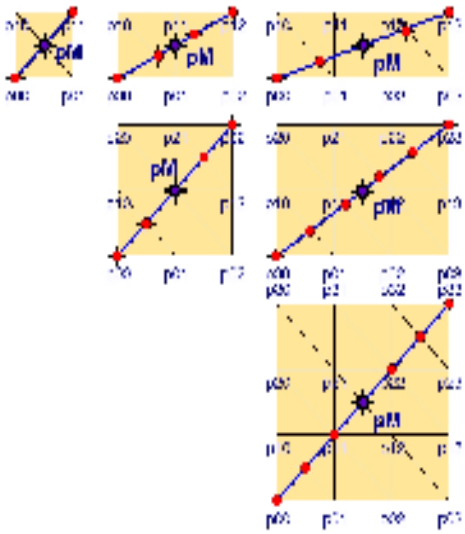


figure 13.1 : the first pSurfaces and their diagonals, from pS22 to pS44.

In a more systematic manner we will now construct the expression of the midpoint of two diagonals of different primitive surfaces, from pS22 (left facet) to pS44 (bicubic), cf figure 13.1 :

```
pS22 :
pM     = MI( MI(p00,p01), MI(p10,p11) )
       = ((p00+p01)/2 + (p10+p11)/2 )/2
       = ( p00 + 2(p01+p10)/2 + p11 )/4
       = ( q0 + 2q1 + q2 )/4
       = MI( q0,q1,q2 )

pS32 :
pM = MI( MI(p00,p01,p02),
         MI(p10,p11,p12) )
   = ((p00+2p01+p02)/4 + (p10+2p11+p12)/4 )/2
   = ( p00 + 3(2p01+p10)/3 + 3(p02+3p11)/3 + p12)/8
   = ( q0 + 3q1 + 3q2 + q3 )/8
   = MI( q0,q1,q2,q3 )

pS33 :
pM = MI( MI(p00,p01,p02),
         MI(p10,p11,p12),
         MI(p20,p21,p22) )
   = ( (p00+2p01+p02)/4+2(p10+2p11+p12)/4+(p20+2p21+p22)/4)/4
   = ( p00+4(p01+p10)/2+6(p02+4p11+p20)/6+4(p12+p21)/2+p22)/16
   = ( q0 + 4q1 + 6q2 + 4q3 + q4)/16
   = MI( q0,q1,q2,q3,q4 )

pS42 :
pM = MI( MI(p00,p01,p02,p03),
         MI(p10,p11,p12,p13) )
   = ( (p00+3p01+3p02+p03)/8 + (p10+3p11+3p12+p13)/8 )/2
   = ( p00+4(3p01+p10)/4+6(3p02+3p11)/6+4(p03+3p12)/4+p13)/16
   = ( q0 + 4q1 + 6q2 + 4q3 + q4 )/16
   = MI( q0,q1,q2,q3,q4 )

pS43 :
pM = MI( MI(p00,p01,p02,p03),
         MI(p10,p11,p12,p13),
         MI(p20,p21,p22,p23) )
   = (  (p00+3p01+3p02+p03)/8 + 2(p10+3p11+3p12+p13)/8
        + (p20+3p21+3p22+p23)/8 )/4
   = (  p00 + 5(3p01+2p10)/5 + 10(3p02+6p11+p20)/10
        + 10(p03+6p12+3p21)/10 + 5(2p13+3p21)/5 + p23)/32
   = ( q0 + 5q1 + 10q2 + 10q3 + 5q4 + q5 )/32
   = MI( q0,q1,q2,q3,q4,q5 )

pS44 :
pM = MI( MI(p00,p01,p02,p03),
```

```
        MI(p10,p11,p12,p13),
        MI(p20,p21,p22,p23),
        MI(p30,p31,p32,p33) )
  = (  (p00+3p01+3p02+p03)/8 + 3(p10+3p11+3p12+p13)/8
       + 3(p20+3p21+3p22+p23)/8+(p30+3p31+3p32+p33)/8 )/8
  = (  p00 + 6(p01+p10)/2 + 15(p02+3p11+p20)/5
           +20(p03+9p12+9p21+p30)/20
           +15(p13+3p22+p31)/5+6(p23+p32)/2+p33)/64
  = ( q0 + 6q1 + 15q2 + 20q3 + 15q4 + 6q5 + q6 )/64
  = MI( q0,q1,q2,q3,q4,q5,q6 )
```

We could likewise study diagonal pSurfaces in pVolumes, and so on…

In general, we can see that :

- operator MI() produces a form of the same dimension,
- operator MIR() produces a form of a bigger dimension,
- operator DIAG() produces a form of a smaller dimension.

Any mid-form can always be written in an explicit form of the type :

```
n = 2 formes: Fm = (           1.F0 + 1.F1           ) /2
n = 3 formes: Fm = (        1.F0 + 2.F1 + 1.F2       ) /4
n = 4 formes: Fm = (    1.F0 + 3.F1 + 3.F2 + 1.F3    ) /8
n = 5 formes: Fm = ( 1.F0 + 4.F1 + 6.F2 + 4.F3 + 1.F4 ) /16
```

expressions where we see the coefficients of Pascal's triangle appearing, leading to the following general expression in the case of n forms :

$$MI(\ Fi\ ) = \sum_i C_{n-1}^{\ i}\ .\ Fi\ /\ 2^{(n-1)}$$

$$\text{with}\quad C_n^{\ i} = n!/(i!.(n-i)!) \text{ and } i \text{ in } [0,n-1]$$

Note the following property, requested for a valid linear combination :

$$\sum_i C_{n-1}^{\ i}\ /\ 2^{(n-1)} = 1$$

This type of coefficient, the particular approach used in this study and some more properties to be seen later lead us to proposing the name « pascalian forms » or « pForms » for this family (with the derived notations : pCurve, pSurface, pVolume, etc.), the letter « p » refering to the « P » for Pascal and the fact that these forms are all dense sets of points, not continuous sets of points leading to forms that are not quite equivalent to the forms in ordinary geometry, even if they can be as close as one might wish. A lot could be said about the question of continuity...

Comment 1 : long before the XVIIth century French philosopher and mathematician Blaise Pascal,

Omar Khayyam, an Iranian mathematician in the XI/XIIth century and Yanghui, an XIIth century mathematician, had studied the properties of this triangle of numbers. So pForms could have been called kForms or yForms, khayyamian or yanghuian, but their pronunciation is harder for an occidental speaker than pascalian forms. So why not « pascalian » ?

Comment 2 : these pascalian forms are not new. Classic literature on the subject refers to Bézier curves and surfaces that were studied from back to front, using a diversity of approaches : Pierre Bézier, an Engineer at Renault, used an algebraic/analytical approach constructed on Bernstein's coefficients ; Paul de Casteljau, an Engineer at Citroën, preferred to study the geometric and recursive aspect. B_splines and NURBS, are important extensions of these. But their presentation often results in a forest of notations and definitions that are often obscure, and out of touch with simple things, with fundamental properties and primitive gestures. By choosing to limit the approach to a handful of elementary geometrical operators applied to a set of points, combined with material representation that can be limited to a simple string to draw segments and find their midpoint, we maintain contact with the simple gestures of freehand drawing, leading quite naturally to an easy approach to forms belonging to more complex spaces, notably immersed forms in the next chapter.

Comment 3 : pascalian forms are a sub-set of the set of valid forms obtained by recursive linear combinations of points, valid because the sums of the coefficients are always unitary. They keep the properties of linear forms and could constitute a good basis for approaching more complex forms, like surfaces defined by implicit equations or equations containing transcendant functions (trigonometric, for instance) or produced by iterative processes like geodesic curves and minimal surfaces as solutions to differential systems of the Laplace type. They could be used as the initial terms of a development in series of a form at one point, (tangent plane to a surface, curved facet or osculating biquadric,…), to analyse the behaviour of diagonals, and beyond that, to envisage a whole range of local geometry.

Having given the general definition of pascalian forms, we can now analyse their main properties as well as their variations and combinations.

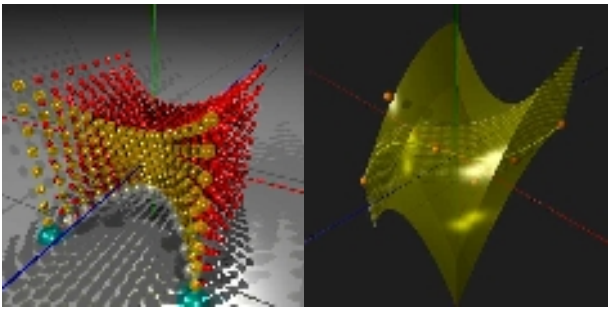


figure 13.2 and 13.3 : a curved cube (pV322) constructed on two ruled paraboloids (pS32) ; transparent yellow pV332 constructed on two pS33, its diagonal surface in yellow marbles (pS42) and yellow diagonal curve (pL6).

2 operations, properties

Summary of this section :

We have to remember that pForms are nothing but Bézier forms approached in a deliberately unitary and simple manner, and inheriting all their properties. We particularly refer to, in the case of pCurves :

- convexity : the pCurve is contained within its control polygon ;
- affine invariance : the transform of a pCurve is the pCurve constructed on control point

transforms ;

- derivation : the « derivative » (we should say hodograph) of a pCurve (pLn) is a pCurve (pLn-1) defined by initial control point differences (the number of points is thus decremented from 1) ; the control points at the extremities immediately given the axes tangent to a pCurve (Serret-Frenet's TNB trihedron) ;
- cutting the control polygon into two concatenates to new subcontrol polygons, allowing effective tracing of the pCurve using a recursive approach, immediate application of de Casteljau's algorithm, including its generalisation to immersed pCurves.

These properties extend to pSurfaces and beyond to all pForms. For more information, you can refer to the bibliography in the appendix.

In what follows, we have chosen to analyse pForm properties by describing the operators implemented in POVRAY/pFlibs, some of which were already used in the previous chapter. The following will be presented :

1. fundamental operators used to subdivide pForms, change the interval of definition, elevate degree, transform in space (translations, rotations, homotheties) and display ;
2. a particular family of operators, the get() operators, to retrieve a subform, a point of a pForm and the local axis ;
3. finally a family of operators associated with an important property of pForms, embedding operators, extending the definition of pForms to curved spaces.

21 fundamental operations

Summary of this section :

The following will be studied : operators for pFsubdivision(), pFelevation(), pFstretch() and the extraction operators pFgetSubForm(), pFgetPoint() and pFgetPijk(). The prefix pF_____() indicates the applicability of the operator to any type of pForm, otherwise the operator can only be envisaged for a subset of pForms, like pCurves, for example.

211 subdivision

Applied to three points, for example, the operator MIR() produces an infinite set of points constituting a parabolic arc. For practical purposes we have defined an operator noted as « pFsubdivision() » starting a recursion for a number of steps that can only be finite and thus producing a finite number of points. Furthermore, to avoid having to define a pFsubdivision() operator for each starting point, these points are grouped in a table sent to the operator as a parameter. This table can contain either the points or other tables of points, making it possible to extend the application of the operator to pForms of a bigger dimension ; as POVRAY is easily able to process four-dimensional vectors, we will stop at 4, for practical purposes, i.e., at pForms of dimension 4. For a given pForm (a table of sub–pForms that can be reduced to points), the call to the pFsubdivision() operator will be written as :

```
#local pForm = pFsubdivision( dim, pForm, recursion )
       with   dim = a whole number between 1 and 4
       and    recursion = a vector containing 4 reals >=0
```

to produce a new pForm, a table containing a subdivision of the initial table, that is « closer » to the pForm theoretically obtained after an infinite number of recursions. We will agree that the initial table and the final pForm are equivalents, refering to a table of three points as a parabola, for instance, in view of the fact that the 3 points lead to a parabola by applying the theoretical operator MIR() and to a polyline as close to the parabola as we wish, by applying the « real » pFsubdivision() operator.

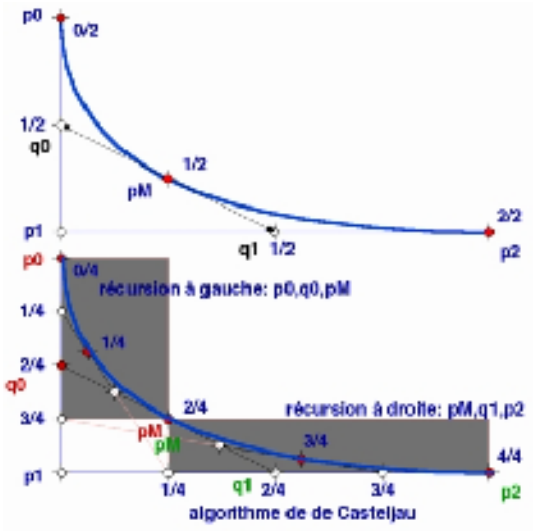


figure 211 : de Casteljau's algorithm, for t=1/2, then t=1/4 and t=3/4..

212 degree elevation

One of the conditions for applying operators MI() and MIR() that has remained in the background up to now, was that the pForms had to be of the same type, the same dimension, of course, but also the same number of definition subforms (the same degree in classic terminology). Looking for the midpoint form of a pL2 (segment) and a pL3 (parabola) seems pointless at first..., unless the segment is considered as a « degenerate » parabola constructed, for example, on three aligned and equidistant points. Note that the contrary is generally impossible ; you can't reduce a parabola to a segment, unless it is degenerate, i.e. constructed on three aligned points. To elevate the degree of a parabola, 4 points are defined according to the points of the parabola :

```
q0     =  p0
q1     = (p0 + 2*p1     )/3
q2     = (     2*p1 + p2)/3
q3     =              p2
the midpoint of a parabola (p0,p1,p2) is written:
pm     = ( p0 + 2*p1 + p2)/4
       = ( q0 + 3*q1 + 3*q2 + q3)/8
```

an expression in which we recognise the cubic midpoint(q0,q1,q2,q3).

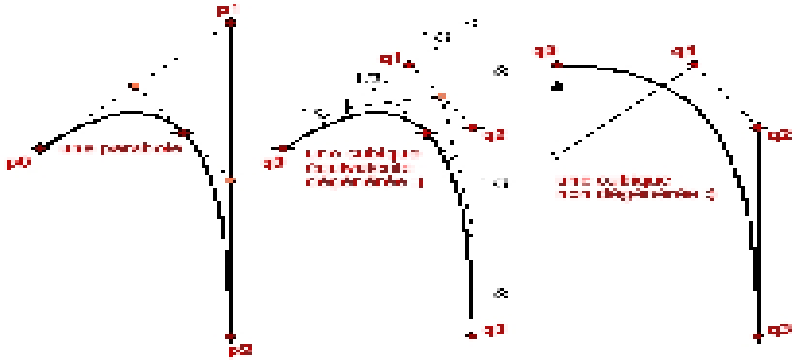


figure 212 : degree elevation of a pL3 (parabola), to become a pL4 (degenerate cubic) then a true cubic.

As a general rule, we are led to defining an operator noted as pFelevation() which, applied to a pCurve pLn will produce the « same » pCurve (pLN) controlled by N>n points, applied to a pSurface pSmn will produce the same pSurface (pSMN) controlled by M>m and N>n points, and so forth for any pForm of any dimension. For the moment, the implementation is operational for pCurves and pSurfaces, and is pending for pForms of any dimension, that are of no particular importance for practical purposes.

The call for the pFelevation() operator will be written thus :

```
#local pForm = pFelevation( dim, pForm, elevation )
      with   dim = a whole number between 1 and 2
      and    elevation = a vector  containing 2 reals >=0
```

213 reparameterization

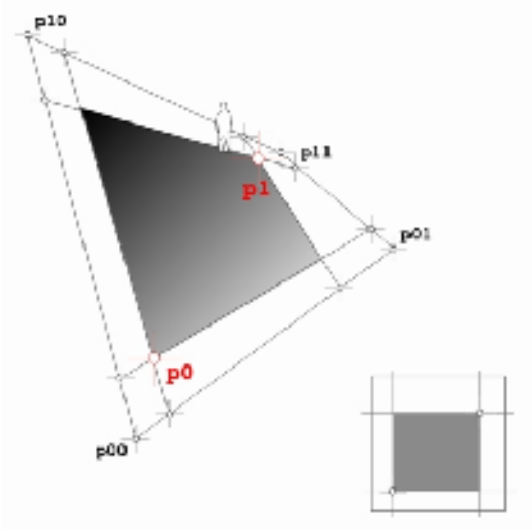


figure 213.1 : curved sub-facet defined by two points p0 and p1.

Let's not forget that pCurves are Bézier curves and that a pL3 (parabolic arc) could, for instance, be written in the following algebraic form :

$$p = (1-u)^2.p0 + 2.(1-u).u.p1 + u^2.p2$$

where p0, p1 and p2 are any three points in space, u a real number in the interval [0,1], and p any point of the parabola lying between p0 and p1. But the algebraic expression produces another point in the parabola for a t value in any interval [a, b], a and b can be infinite.

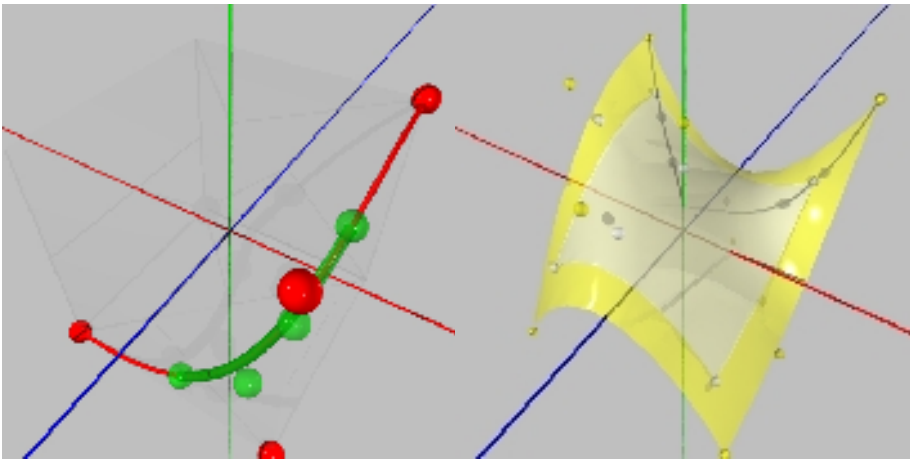
As a pL3 is initially defined by 3 points (p0,p1,p2) between the endpoints p0 and p1, it could be useful to stretch (or restrict) the definition interval to the portions lying between any two points of the parabola, pA and pB, thus leading us to define an operator noted as pFstretch(). Applied to a pCurve, this operator will create a sub-curve between the two specified points as curvilinear abscissa u0 and u1 ; applied to a pSurface, it will create a sub-facet between the two points specified as 2 curvilinear abscissa

and ordinates (u0,v0) and (u1,v1), and so forth up to dimension 4.

The call for the pFstretch() operator is written thus :

```
#local pForm = pFstretch( dim, pForm, pA, pB )
      with   dim = an integer between 1 and 4
      and    pA, pB = two points written in R4 (  )
```

to produce a new pForm, « immersed » in the first between points pA and pB. Other embeddings will be envisaged later, leading to new generalisations of pForms.



figures 213.2 and 213.3 : resetting the parameters of a pL4 (cubic), and a pS33 (biquadric) resulting in the restriction of these pFormes.

214 extractions, tangent axes

Three « get() » operators, are defined to find a sub-form generating the pForm, giving a point of the pForm, and the local axis at this point.

2141 pFgetSubForm()

A pForm is produced by applying operator MIR() to a table of sub-forms. Operator pFgetSubForm() turns the generating sub-form for a given value « u » ; for u comprised in the interval [0,1] the sub-form will be contained in the pForm. The call is written as :

```
#local pForm = pFgetSubForm( dim, pForm, u )
```

```
      with dim dans [1,4] and u in R
```

2142 pFgetPoint()

Recursive application of the preceding operator pFgetSubForm() pushed to the limits will return a point p to R4 ; this is what is produced by operator pFgetPoint() whose call syntax is as follows -cf figure 2143.1 - :

```
#local P = pFgetPoint( dim, pForm, p )
      with dim in [1,4] and p in R4
```

Note that parameter p is a point in R4 ; in the case of a pSurface, for example, we would write :

```
p = pFgetPoint( 2, pSurf, <u,v,0,1> ).
```

2143 pFgetPijk()

The study of local properties at a point of a pForm begins by calculating the local axis at this point. We will examine the case of curves, surfaces and volumes.

21431 the case of a curve

The local axis at a point in a curve (called a Serret–Frenet trihedron) is the set of three unitary vectors constructed on the tangent at the current point, the normal contained in the osculatory plane to the curve at this point and the binormal perpendicular to this plane -cf figure 2143.4 -. In the case of a pCurve these three vectors are easy to calculate at the original control point ; let p0 be this control point, p1 and p2 the two following control points :

1. the vector carried by p0 and p1 is the vector tangent at p0 to the curve ;
2. the plane defined by points (p0,p1,p2) being by construction osculatory to the curve at p0, the vectorial product of vectors p0p1 and p0p2 gives the binormal ;
3. the vectorial product of the binormal and the tangent gives the normal.

To calculate the Serret–Frenet trihedron at a point other than the first point of control, for instance, at the curvilinear abscissa « s » point, calculate the equivalent curve starting from this point, i.e., by shifting the interval of definition from [0,1] to [s,s+1] and by calculating the trihedron at this new point of departure. It should be noted that the Serret–Frenet trihedron is not defined for all points on a segment, and for "true" curves, on the line of the inflection points, and the osculatory plane is undetermined. Moreover, as the curvature inverts at an inflection point, the trihedron shifts 180° producing the abrupt torsion of tubular surfaces constructed on the Serret–Frenet trihedron, dealt with in paragraph 323.

21432 the case of a surface

The local axis at one point is the set of three unitary vectors constructed on two tangents at the current point (tu/|tu|,tv/|tv|), with the normal perpendicular to this plane (vectorial product). As an analogue to what we saw earlier, these three vectors are easy to calculate at the starting control point; let p00 be this control, p01 and p10 the following control points in each direction :

1. the vectors carried by pairs (p00,p01) and (p00,P10) are tangent to the surface at p00, but nonorthogonal to each other ;
2. the vectorial product of these two vectors is normal to the tangent plane at this point.

Calculating the tangent axis at another point will be done in a similar way to that used for the curve by shifting the definition interval -cf figures 2143.1 to 2143.3-.

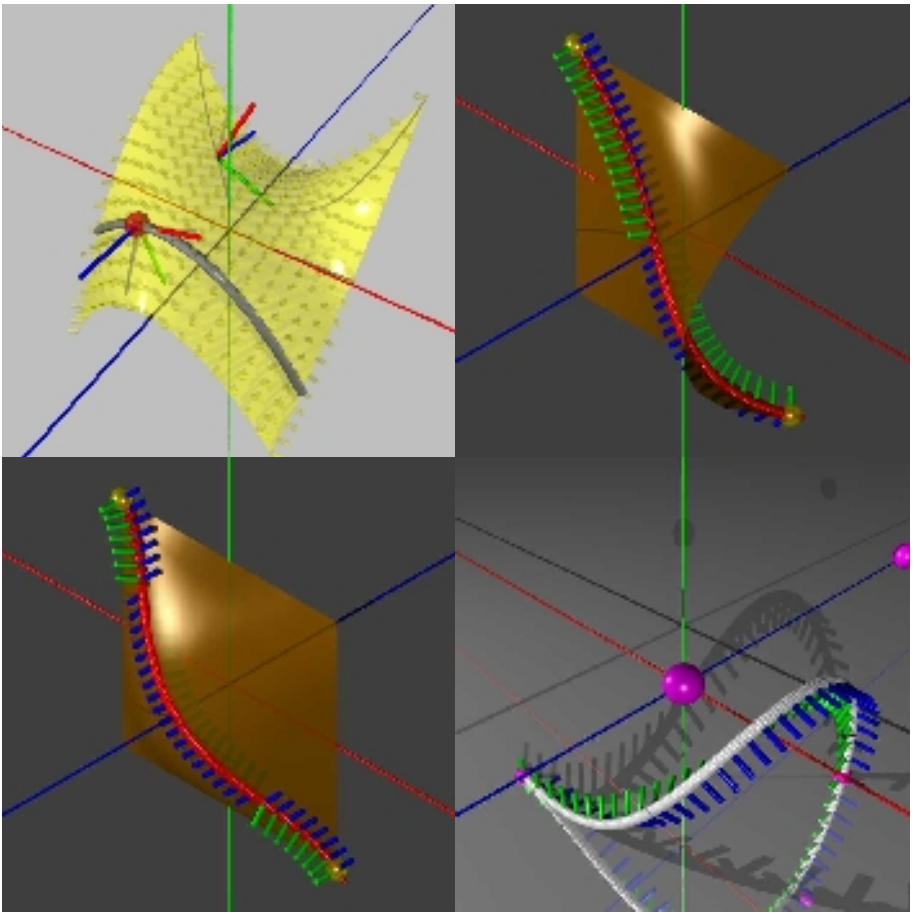
21433 the case of a volume

The local axis at a point is simply constituted by the triplet of unitary vectors (tu/|tu|,tv/|tv|,tw/|tw|) constructed at this point and the following 3 points suivants on axes u,v,w. Note, however, that this axis is not orthogonal.

Operator pFgetPijk() brings back the local axis in the case of pCurves and pSurfaces, call syntax :

```
#local Pijk = pFgetPijk( dim, pForm, p )
      with dim in [1,2] and p in R4
```

Comment: In the case of a curve, the (Serret-Frenet) axis at each point of a curve is orthonormed, in the case of a surface the tree vectors are unitary but the two vectors tangent to the surface are not orthogonal to each other, in the case of a volume, the three vectors are unitary but not orthogonal to each other. Calling on concepts of norm and orthogonality is not really coherent with the principles of any affine geometry considered to be free of metric theory. We will look later (323 Tubular surfaces) at the adverse effects of such an introduction, to the extent that one might wonder about the interest of introducing norm and orthogonality into tangent axes. The question has been raised ...



figures 2143.1 to 2143.4 : operator pFgetPijk() brings the tangent axis to a pSurface or a pCurve.

22 immersions

Summary of this section:

In the first part we saw the importance of the operator DIAG() in the very defining of a pSurface, producing a pL3 (parabola) in a pS22 (curved facet), a pL5 in a pS33 (biquadric), a pS32 (ruled paraboloid) in a pV222 (curved cube), etc… These diagonals will prove to be the basic constituents of these pSurfaces and beyond these, to all pForms, giving rise to the possibility of a whole geometry in curved surfaces. In the process it will be necessary to study the conditions in which a pForm can interpolate an array of points, (which is not the case for the control points, apart from the endpoints). And finally we will see the emergence of some interesting properties for embedding pForms in other pForms, that seem to justify the « pascalian » approach to curved forms in themselves.

221 interpolation

pCurves do not interpolate intermediary control points. We might wish to work with the control points along the pCurve, generating their « real » control points in the background. There is no unique solution, several curves can interpolate a series of points, according to what could be called the respective weight associated with these points, and/or other conditions for the tangents, curves, etc... In this study, we will consider that the interpolated points, called knots, form an evenly distributed series on the abscissa i/N with i=[0,N], and the reader is free to go on from there if he wishes ;) ...

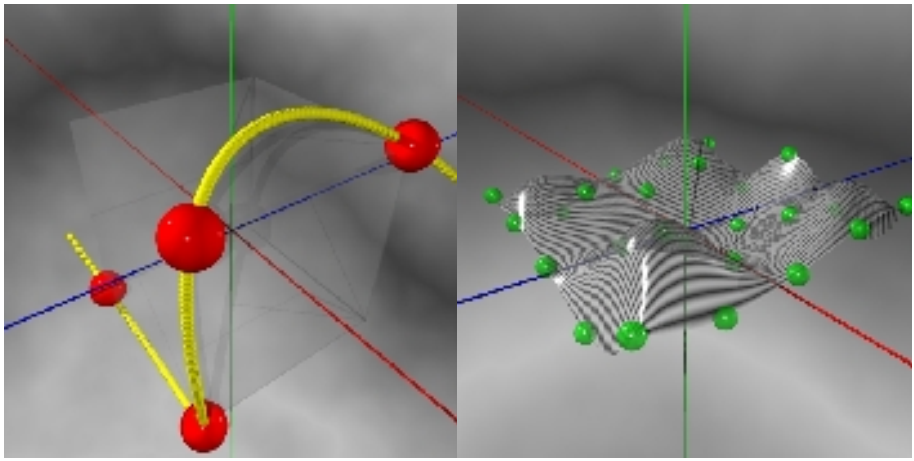
The calculation is made by expressing these points (knots) according to the control points of the desired pCurve and by inverting the linear system thus formed. Below are details of the solutions corresponding to the common cases of the pL3 (parabola), the pL4 (cubic) and the pL5, points b0 to b4 being the knots to interpolate :

```
- case of a parabola pL3:
#local L3 = array[3]
#local L3[0] = b0;
#local L3[1] = (-b0 +4*b1 -b2)/2;
#local L3[2] = b2;
```

```
- case of a cubic pL4: -cf figure 221.1-
#local L4 = array[4]
#local L4[0] = b0;
#local L4[1] = (-5*b0 +18*b1  -9*b2 +2*b3)/6;
#local L4[2] = ( 2*b0  -9*b1 +18*b2 -5*b3)/6;
#local L4[3] = b3;

- caes of a pL5:
#local L5 = array[5]
#local L5[0] = b0;
#local L5[1] = (-13*b0 +48*b1  -36*b2 +16*b3  -3*b4)/12;
#local L5[2] = ( 13*b0 -64*b1 +120*b2 -64*b3 +13*b4)/18;
#local L5[3] = ( -3*b0 +16*b1  -36*b2 +48*b3 -13*b4)/12;
#local L5[4] = b4;
```

Beyond this, it is better to use a general method, and we propose in the POVRAY/pFlibs library an inversion operator based on the Gauss–Jordan algorithm, which proves to be very fast and well adapted to the problem in hand. This operator is only implemented, for the moment, for curves (pLinterpolant()), and for surfaces (pSinterpolant()) with the following call syntax :

```
// curv array of points
#local curve_interpolant = pLinterpolant( curv )
// surf array of points
#local surface_interpolant = pSinterpolant( surf )
```



figures 221.1 and 221.2: pl4 interpolating 4 points, and pS55 interpolating 25 points.

222 diagonalisation

By systematically studying the expression of the midpoint of a pSurface from pS22 to pS44, we have demonstrated the diagonal control points expressed according to the pSurface control points. An « orthonormed » representation of the pSurface - the pSurface becomes a unit square - shows the even distribution of these points on the diagonal inducing a general expression that was used in by an operator pFdiagonalisation() applicable to any pForm of any dimension. The call for this operator is simply this :

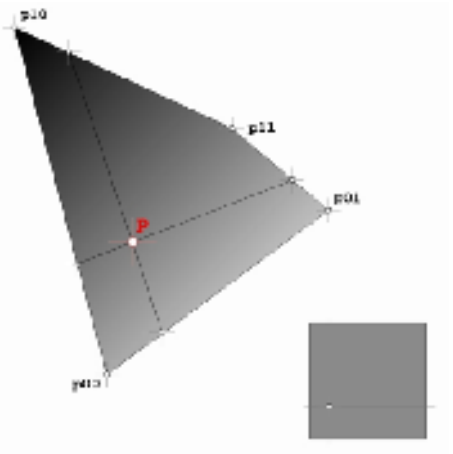
```
#local diag = pFdiagonalisation( dim, pForm ) // with dim>=2
```

and produces a pForm of a smaller dimension, the first example of a pForm immersed in another pForm.

223 immersed pForm

We will now reconsider diagonals from a more general angle, leading to the construction of pForms immersed in other pForms. We will start by studying the curves it is possible to trace in pSurfaces, after brief consideration of the point in a surface.

2231 a point in a pSurface



figures 223: a point in a pS22 (curved facet), and orthonormed representation.

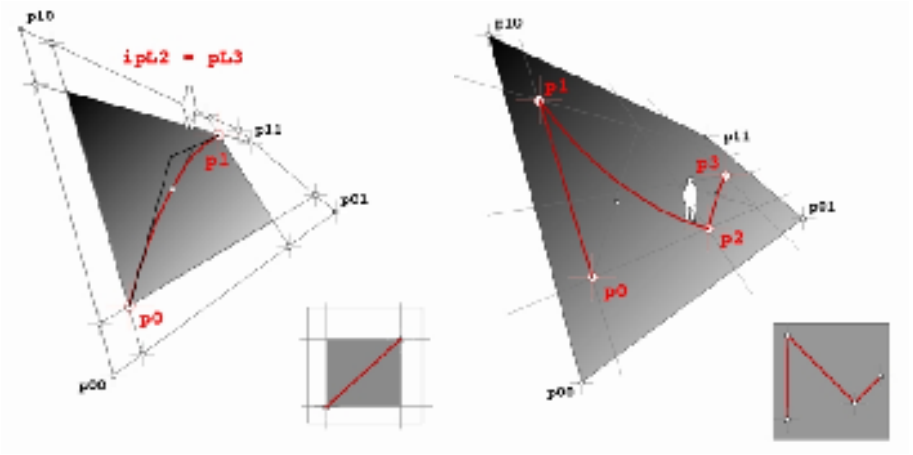
Any point of a pSurface is perfectly defined by the data of two real numbers (curvilinear coordinates) usually comprised in the interval [0,1] ; in fact there is nothing to stop one going beyond this interval, the

surface is perfectly defined in the interval [-infinite, +infinite]. In POVRAY/pFlibs syntax, the operator pFgetPoint() whose call syntax for a pSurface is the following :

```
#local P = pFgetPoint( 2, pSurface, p )
with input p = <u,v,0,1> and output P = <x,y,z,t>
```

waits for a point p defined in the local (curvilinear) axis in the form of <u,v,0,1> and brings a point P to global space in the form <x,y,z,t>, cf 2142. Using this operator, it is possible to immerse any curve, and even any surface, in a pSurface ; a circle, for instance, defined by its canonical parametric equation P(t) = [cos(t), sin(t)] in the curvilinear axis attached to the surface, can thus be transformed point by point into a curved curve plated on the surface. The nature of this punctually defined curved curve is unknown at this stage, and the aim of the subsequent study is to find it out ; we'll start with the curves that are the easiest to draw on a pSurface, curves defined by two points.

2232 two points in a pS22



figures 2232.1 and 2232.2: an ipL2 (iSegment) passes through two points of a pS22, and an iPolyline passes through several.

First of all let's consider the simplest curved surface, the pS22 (curved facet). Any two points p0 and p1 belonging to the pS22 unitarily define an immersed sub–pS22, denoted ipS22, and thus a single diagonal curve belonging to it, a parabola (pL3) linking the two points p0 and p1.

For an « inhabitant » of the space outside the surface, this curve is a parabola defined by 3 points, but for an « inhabitant » of the surface who knows nothing of the space in which the surface is immersed, the curve linking two points p0 and p1 acts as a « classic » segment. The curve is perfectly defined by the data of the two points (the third point does not belong to the surface and the inhabitant does not know of

it) ; it is « isocline » in the sense that it cuts the generative lines following a « slope » that is constant as it is defined (the recursive incrementation of divisions by two is equal in both directions) ; and by defining the metrics on the basis of the number of points defining it, one should even be able to speak in terms of a geodesic curve. For these reasons this segment will be called an « immersed segment » denoted as « iSegment ».

To construct an immersed segment between two points P0 and P1 in a pS22 is like constructing the diagonal of the sub–facet ipS22 constructed on these two points, i.e., a parabola defined by three points (B0, B1, B2) ; as the endpoints are known (B0 = P0 et B2 = P1), the only point to determine is the intermediary point B1. We know the midpoint P1/2 of facet ipS22 constructed on (p0,p1) :

```
#local P1/2 = pFgetPoint( 2,pS22, <1/2,1/2,0,1>);
```

Inverting the expression giving this point as the midpoint of the external parabola :

```
P1/2   = MI( B0, B1, B2 )
       = (B0 + 2*B1 + B2)/4
       = (P0 + 2*B1 + P1)/4
```

we find :

```
B1     = ( -P0 + 4*P1/2 -P1 )/2
```

and the parabola being sought :

```
pL3    = MIR( P0, (-P0 + 4*P1/2 -P1 )/2, P1 )
```

« Internally », forgetting the parabola, we are naturally led to extending the definition of operators MI() and MIR(), and treating this curve as an immersed pascalian curve, denoted ipL2, we can write :

```
P1/2   = MI( P0, P1 )
ipL2   = MIR( P0, P1 )
       = pL3
```

A door is now open to define pascalian forms in curved spaces, at least in pForms. To begin with, this involves going beyond the simple case of the curved facet pS22.

2233 two points in a pS32

In the case of a pS32 (ruled paraboloid), constructing the immersed segment between two points P0 and P1 is like constructing an « external » cubic defined by four points (B0, B1, B2, B3) ; as the endpoints are known (B0 = P0 et B3 = P1), the points to determine are intermediary points B1 and B2. Points P(1/3) and P(2/3) at 1/3 and at 2/3 of the diagonal of the ipS32 constructed on (P0,P1) as points at 1/3 and at 2/3 of the pL4 :

```
P1/3   = (8.B0 + 12.B1 + 6.B2 + B3)/27
P2/3   = (B0 + 6.B1 + 12.B2 + 8.B3)/27
```

and this is inverted to arrive at the two intermediary points :

```
B1     = (-5.P0 + 18.P1/3 -9.P2/3 + 2.P1)/6
B2     = (2.P0 - 9.P1/3 + 18.P2/3 - 5.P1)/6
```

and the cubic being sought.

The midpoint of the cubic being known, here again we can write :

```
P1/2   = MI( P0, P1 )
ipL2   = MIR( P0, P1 )
       = pL4
```

2234 two points in a pSurface

We notice that an ipL2 (immersed segment) is a pL3 in a pS22, a pL4 in a pS32. We continue likewise to a pL5 in a pS42 or a pS33. Generally, an ipL2 (immersed segment) in a pSmn is a pCurve controlled by N = (m+n–1) points. These points are calculated from an even distribution over the immersed segment (ipL2) of N points P(i/(N–1)) with i = [0, N–1]. The pCurve interpolating these points is the solution to the problem, and we can write :

```
P1/2   = MI( P0, P1 )
ipL2   = MIR( P0, P1 )
       = pLN  with N = m+n-1
```

So, in a more general way than using the pFdiagonalisation() operator, we now know how to draw straight « segments » linking any two points in any pSurface, and a number of comments can be made about this important property :

Comment 1 : from this generalised basis for the segment, we can extend to concepts of parallelism, orthogonality, and angle, study the intersection of two iSegments, its divisions, prolongations, etc..., thus defining geometry in pSurfaces.

Comment 2 : an iSegment is not normally a geodesic, it is not the shortest route between two points, the normal at each point is not colinear to the normal at the surface at this point ; but it might be interesting to take iSegments as a basis for the study and construction of the geodesics of a pSurface.

Comment 3 : the idea of considering a parabola as a simple segment immersed in a curved facet could simplify a problem that's difficult to treat in classic euclidian space by reducing it to a simpler problem in curved space defined by the facet. Note that there is an infinite number of facets admitting a parabola as a diagonal : as P0,P1,P2 are given, all the pairs of points q0 and q1 such that P1 is the midpoint, would be suitable.

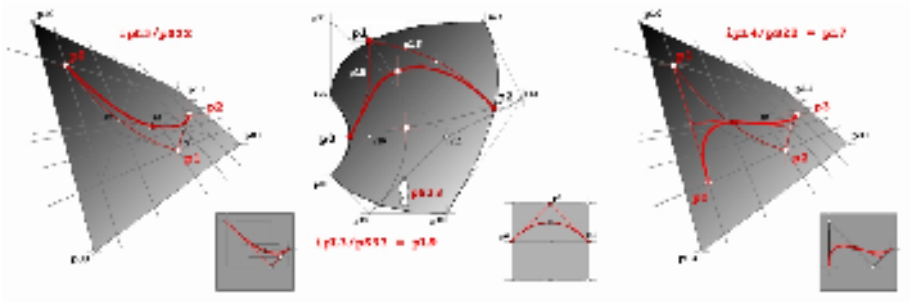
Comment 4 : in the case where the pSurface is taken back to its orthonormed form (unit square), these segments are indeed rectiliear, and any pSurface and its immersed curves could be considered as an « anamorphosis » of the orthonormed form.

Comment 5 : constructing pForms in a space free of any metrics opens the possibility of defining segments immersed in « curved spaces », at least for curved pForms. Remember that in the introduction, we said that in order to find the midpoint of points on a surface, we envisaged « *stretching the string by*

placing it on the curve following the one curved, or geodesic, curve constituting the shortest route between these two points and along which the normal to the string (in the osculator plane) is colinear to the normal to the surface ». Following the geodesic between two points was the condition imagined to ensure the unicity of the path. In the case of a pForm, the geodesic can be replaced by the immersed segment, which is far more practical to construct, using a simple piece of string : a surveyor who had to draw a « straight » line on a piece of ground similar in form to a curved facet, i.e., a parabola in our space, would easily place the midpoint of a stretched rope following a straight line between the two axes opposite the axes to be linked, and from the three control points, would construct the first midpoint of the immersed structure, and so forth.

Beyond the two given points of a pSurface, polylines could be envisaged composed of immersed segments linking the series of points, and between these, triangles, regular polygons tending toward immersed circles, sinusoidal curves, etc., could be traced. It might be more fruitful to construct pCurves, starting from the immersed parabolas, or ipL3.

2235 three points in a pSurface

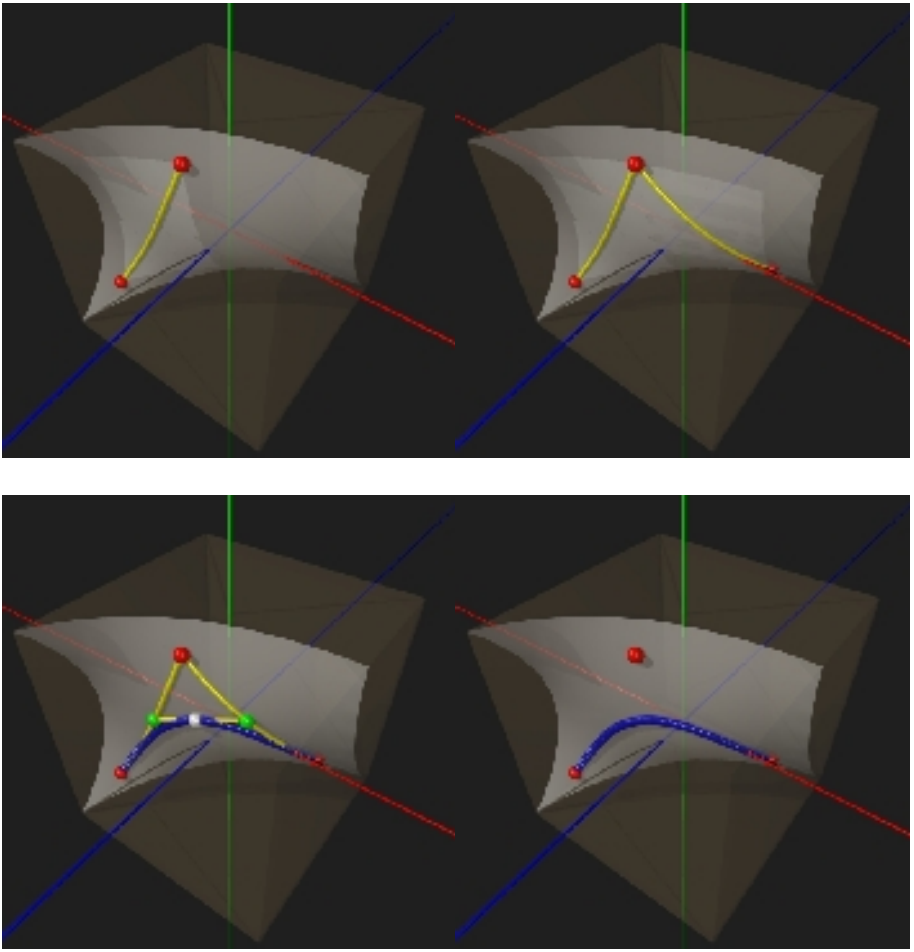


figures 2235.1 to 2235.3 : ipL3 (immersed parabola) in a pS22 and in a pS33 ; ipL4 (immersed cubic) in a pS22.

Similarly, a generalisation of operators MI() and MIR() is envisaged, such that an immersed parabola, or ipL3, could be generated from the three points pertaining to a pSurface. And just as an ipL2 immersed in a pSurface is associated with a pCurve in space, we will try to find the pCurve associated with the ipL3. Given the way in which pascalian forms are constructed and their "linear" properties, we « know » this pCurve exists, it's « part of the family » (see comment later). The construction process for an iSegment called for an even distribution of points "aligned" on the surface, and likewise, the construction process for an ipL3 will work on an even distribution of points along a parabolic path in the surface (think in terms of its orthonormed representation).

A purely inductive reasoning gives the following results : in the case of a pS22, the curve required is a pL5 whose control points are calculated on the basis of an even distribution on the ipL3 of 5 points P(i/4) with i = [0, 4]. In the case of a pS33, we can show that a pL9 is required. And in the general case, an immersed parabola (ipL3) in a pSmn is a pCurve controlled by N = (m+n–2)*2+1 points.

```
P1/2   = MI( P0, P1, P2 )
ipL3   = MIR( P0, P1, P2 )
       = pLN
       with N = (m+n-2)*2+1
```



figures 2235.4 to 2235.8: from 3 points, construction of a blue ipL3 (immersed parabola) and its yellow controlpolygon in a pS33 (biquadric).

2236 generalisation: immersed pForms

Likewise we will find that an immersed cubic (ipL4) defined by four points pertaining to a pS22 is a pCurve of space constructed on 7 points.

In the case of any pCurve immersed in any pSurface we will write :

```
N = ( m + n – 2 ) * (q–1) + 1
      where q is the number of control points
                          of the pCurve defined in the pSurface,
      and (m,n) are the numbers of control points
                          of the pSurface in both directions.
```

Finally in the case of any pCurve immersed in a pForm of any dimension, based on purely inductive reasoning (cf comment), we can « risk » writing :

```
M = ( m1+m2+..+mi – d ) * (q–1) + 1
      where q is the number of control points
            of the curve defined in the pForm,
      d is the dimension of the pForm,
      and (m1, m2, ..., mi) is the number of control points
            of the pForm in the different directions (dimensions).
```

This formula can also be applied to pSurfaces immersed in a pSurface or pVolume, and as a general rule to pForms immersed in pForms ; all that is needed, grace to linear properties of pForms, is to reason separately for each dimension. Here is a summary table, showing for instance that an ipL3 in a pS33 is a pL9.

	pS22	pS32	pS33	pS43	pS44	pV222
ipL2	pL3	pL4	pL5	pL6	pL7	pL4
ipL3	pL5	pL7	pL9	pL11	pL13	pL7
ipL4	pL7	pL10	pL13	pL16	pL19	pL10
ipS22	pS33	pS44	pS55	pS55	pS77	pS44
ipV222	/	/	/	/	/	pV444

Once the number N of control points defining the pCurve is known, an even distribution of N points is calculated along the immersed pCurve, and the pCurve interpolating these points is the solution of the problem.

This result is remarkable, in the sense that it establishes a duality between two representations of the same form, with or without control point interpolation, whether or not immersed in a more complex space. But the basic result is that a pForm immersed in another pForm is equivalent to a pForm in the surrounding space elevated to a higher degree ; herein lies the interest of the immersed form concept, and this can be said for all cases of pForms. All pForms rely on using two operators MI() and MIR() applied

first to any set of points in space, then to generated forms, then to the forms pertaining to these forms, etc…. All these pForms are based on linear combinations with Pascal coefficients. In the next chapter, we will examine new combinations of these pForms in order to extend their range of application to more complex cases, starting with forms based on circles.

Comment : affirming the previous results, whether for correspondence formulas for control points or affirmation «... the pCurve interpolating these points is the solution of the problem » implies a rigorous mathematical demonstration that has not yet been given. pForms are in fact nothing more than multilinear combinations of points, a parabola is simply a degenerate bilinear form, a cubic is a degenerate trilinear form, and so on. It seems reasonable to think that the results stated for the first forms are transmitted to the more complex pForms they generate, and a precise mathematical formulation of this reflection could constitute a rigorous demonstration. The algorithms developed in computer implementation constitute an « experimental machine » producing pForms and testing their properties, and once running, these have validated - at least visually - all the scenarios envisaged. These algorithms may be of demonstrative value in their own right ! And if it turned out that the immersed curves were not immersed outside the points of the pSurfaces they interpolate, this would probably have little effect – at least - on their practical use. Moreover, from a purely intellectual point of view, once the worst is over, a whole field of experimentation opens up, looking for deeper mathematical reasons to invalidate these affirmations.

As for the POVRAY/pFlibs implementation of the operator pFembedding(), apart from operator pFdiagonalisation() that can only produce the iSegments in any pForm, two versions have been developed, the first applicable to all pForms but producing the ipForm without its control points, and the second applicable to pSurfaces producing an ipCurve with its control points.

```
#local iForm = pFimmersion( dim, pForm, idim, ipForm )
      with dim, idim in [2,4], f and imf two pForms
#local ipCourbe = courbe_in_surface( courbe, surf )
```

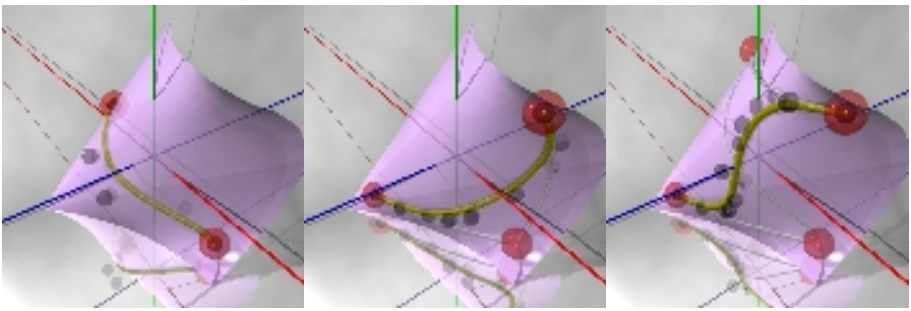
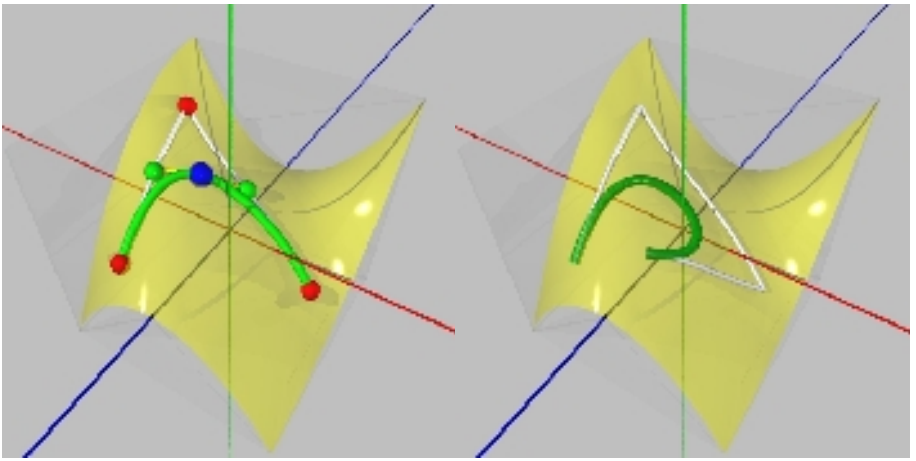


figure 2236.1 to 2236.3 : an ipL2 in a pS33 is a pL5 of the space ; an ipL3 in a pS33 is a pL9 of the space ; an ipL4 in a pS33 is a pL13 of the space.



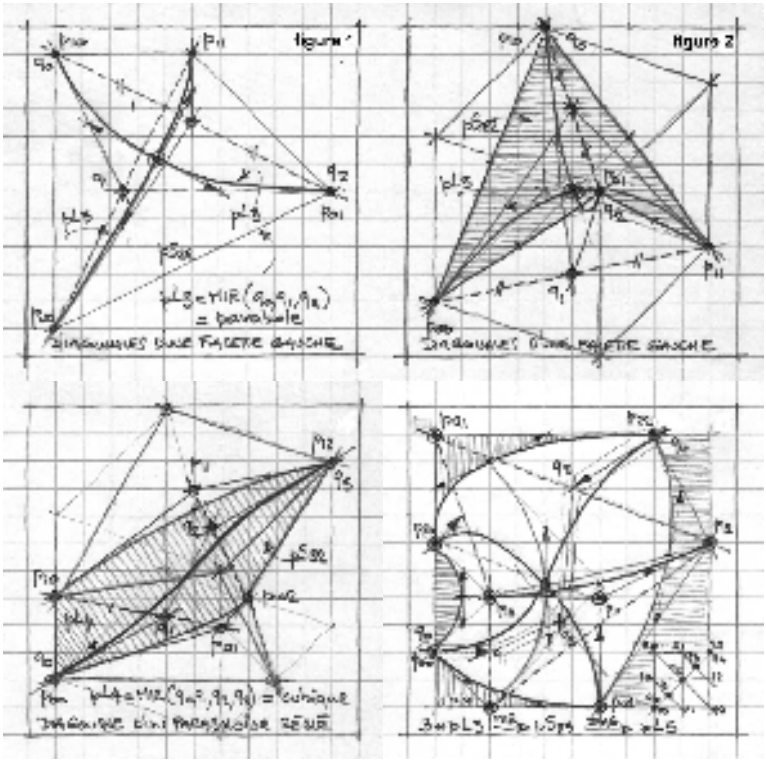
figures 2236.4 and 2236.5: a parabola and a cubic in a biquadric.

23 interface

Summary of this section :

Once pForms have been defined, they can then be manipulated and visualised. They can be drawn by hand - that was the initial aim, after all, to be able to draw them freehand using a string and a few pegs - and some hand drawn constructions on squared paper will be given in figures 23.1 to 23.4.

The pFlibs.inc library can also be used, written in the POVRAY environment and designed for this. We have already used four operators, i.e., pFsubdivision(), pFstretch(), etc... and the indispensable operator draw() to display everything we have been able to imagine, including some we weren't even expecting... The transformations (translation, rotation, homothesis) and the particularities of the display operator will be analysed below.



figures 23.1 to 23.4 : pForms can ALSO be constructed by hand: diagonals and parabolas in a curved facet ; ruled paraboloid and biquadric with their matching diagonals...

231 transformations

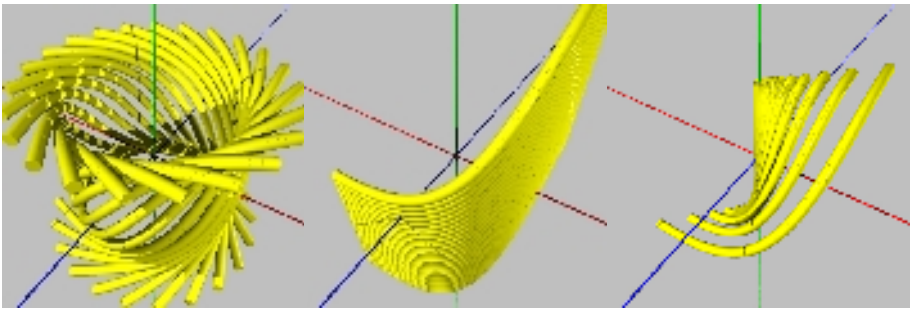
Once a pForm has been defined, it is often necessary to apply to it a translation, a rotation, a scale change (scale, homothesis), a sliding deformation (shear), or a more general affine transformation as is used in the case of the tubing examined later in the text. In regard to the most general transformation, an operator pFtransform() is defined applying a 4x4 matrix to a point in R4 ; this operator is for purely internal use, for instance to construct the operators producing rotating forms, tubes, etc…. The call syntax is the following :

```
#local p1 = pFtransform( mat, p0 )
// the operator rotates p1 transformed from p0
```

The standard operators used are pFtranslate(), pFrotate() and pFscale() which do what they are

expected to do, i.e., transform the pForms in our R3 space by translations, rotations and homothesis. The calls for these operators are written thus :

```
// be careful: the three operators modify pForm:
// translation of a pForm in R3:
pFtranslate( dim, pForm, < tx, ty, tz > )
// rotation on three axes Ox,Oy,Oz:
pFrotate( dim, pForm, < rx, ry, rz > )
// homothesis in relation to three axes Ox,Oy,Oz:
pFscale( dim, pForm, < sx, sy, sz > )
```



figures 231.1 to 231.3: rotations, translations, homotheses applied to a pL4 (cubic).

232 representation

In general, having defined a pForm, then stretched it, transformed and finally subdivided it, you get an irresistible desire to visualise it… But points, curves, surfaces, volumes and, even more, hypervolumes, are not represented in the same way. Furthermore, there is not necessarily only one representation for a given form : a volume can be represented by all the points it is composed of, by boundary surfaces, by a stack of surfaces (mille–feuilles), by a beam of fibres (normals of the layers of the mille–feuilles), by a distribution of tangent axes (tu,tv,n), by characteristic curves such as curvilinear coordinated curves, diagonals, etc…

Two types of representation can be distinguished : a general form valid for any pForm, and specific forms adapted to pCurves, pSurfaces and pVolumes.

2321 basic form

A pForm is basically just a cloud of points, a multidimensional array of points, and its most elementary representation can naturally take the form of a cloud of small spheres. An operator pFdraw() is defined for this, with the following call syntax :

```
pFdraw( dim, pForm, R )  // with dimension dim and radius R
```

POVRAY will display a rather dull cloud of black spheres ; for practical purposes a colour or material will be applied to this group of spheres, and the following call will thus produce the 4 apexes of a curved facet pS22 (or more if the facet has been subdivided) in the form of red spheres with a radius of 0.02.

```
union { pFdraw( 2, pS22, 0.02) une_couleur( < 1,0,0,0 > ) }
```

Note that pFdraw() can be applied to pForms of any dimension, including hypervolumes !

2322 specific form

For curves, surfaces, volumes and even hypervolumes, a more adapted form of operator pFdraw() is proposed ; a new operator draw() (without the pF prefix to underline its specific nature) simplifies and concentrates the writing of the various parameters calling on a number of options as described in the following table :

```
draw( dim, pForme, options ) with:
   1) dim : a real number in the interval [0,4]
   2) pForme : any pForm
   3) options : partial sum of a set of the following choices :
      a) STANDARD : line values by default, red sphere (r=0.02),
      b) finesse( recursion ) : select the level of form subdivision
where recursion = r | < rx,ry > | < rx,ry,rz > | < rx,ry,rz,rt >,
      c) point( R ) : draws spheres of radius R,
      d) courbe( R ) : draws cylinders of radius R,
      e) surface( FACETTES | LISSE | SUPER ) : draws a mesh of
triangles with sharp or smooth edges with a +or- fine interpolation of
colours,
      f) enveloppe( FACETTES | LISSE | SUPER ) : draws the 6 sides of
a volume,
      g) feuilles( FACETTES | LISSE | SUPER ) : draws a volume as a
mille-feuilles,
      h) fibres( rayon ) : draws the fibers of a volume,
      i) normales( rayon, longueur ) : draws the normals to a surface,
      j) repere_tangent( rayon, longueur ) : draws the axes tangent to
a curve or a surface,
      k) couleur( r,g,b,t  ) : selects a colour and opacity,
      l) matiere( choix ) : choose a material in ( GOLD | MIROIR |
GRANIT | MARBRE ).
```

Examples using this operator and the choices available are given throughout this document. Basicaly, a curve, a surface and a volume wil be drawn using the following calls :

```
draw( 1, pCourbe,
```

```
      finesse(5)                  // recursion level 5
      + courbe( 0.02 )            // line radius 0.02
      + ma_couleur(<1,0,0,0>) )  // red

draw( 2, pSurface,
      finesse(<4,4>)              // 4 on U and V
      + surface(LISSE)            // smoothing of triangles
      + ma_couleur(<0,1,0,0>) )  // yellow

draw( 3, pVolume,
      finesse(<2,2,2>)            // 2 on U, V, W
      + enveloppe(LISSE)          // the 6 faces of the cube
      + ma_couleur(<1,1,0,0.5>) )// transparent yellow
```

Other options are available according to needs : hypervolume representation could evolve toward more expressive combinations of layers and fibres, or integrate time management in the form of animations, which POVRAY has no problem handling.

3 compositions, applications

Summary of this section :

Concluding the first approach, chapter 1 allowed us to define pascalian forms generally, and « chapter 2 » presented the fundamental properties, without implying any particular position for the control points. This chapter will enable us to discover the particular arrangements of control points reproducing the classic shapes in geometry (circles, surfaces of revolution,...) operators combining several pForms to producing pipes, Coons surfaces, parallel surfaces, useful concatenations for generating spline curves and surfaces, and enabling us to make a tentative approach to the wonderful continent of curved space geometry...

31 rational forms

Summary of this section :

We already stated in paragraph 111 that the points were defined in R4 in the form referred to as homogenous coordinates (projective form) : <x,y,z,t>, associating the point in R4 with the point in R3 defined as <x/t,y/t,z/t>. Up to now, value 1 has always been taken by default as the fourth coordinate ; now it's time to examine cases in which t is different from 1. In the case where t varies (in principle t can vary from -infinite to +infinite), a given point in R4 will match a set of R3 points defined as <x/t,y/t,z/t>, and a curve defined in R4 will match a set of R3 curves ; note that in this way a surface can be constructed in R3 from the given of a single curve in R4 !

A pCurve is always expressed algebraically in a polynomial form, the current point of a pL3 (parabolic arc) thus being expressed according to the 3 control points :

```
p = (1-u)^2.p0 + 2.(1-u).u.p1 + u^2.p2    // u in [0,1]
```

or explicitly, by listing the 4 coordinate points as (x, y, z ,t) :

```
x = (1-u)^2.x0 + 2.(1-u).u.x1 + u^2.x2
y = (1-u)^2.y0 + 2.(1-u).u.y1 + u^2.y2
z = (1-u)^2.z0 + 2.(1-u).u.z1 + u^2.z2
t = (1-u)^2.t0 + 2.(1-u).u.t1 + u^2.t2
```

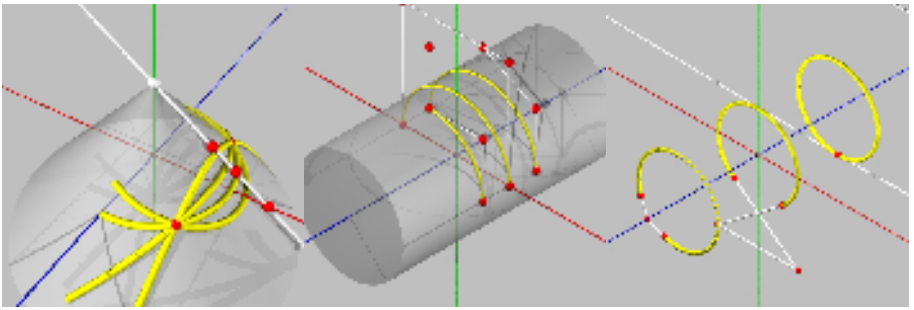
The coordinates of the R3 point defined as <x/t,y/t,z/t> will be the « rational » expressions, i.e., polynomial quotients (ratios) ; so we speak of rational curves, and beyond of rational forms. Pascalian forms, always defined in R4 produce rational forms in R3 in the general case, reducing to polynomial forms in the case where the control points all have a value of 1 for the fourth coordinate.

But what are rational forms for ?

311 conics

If we remain in R3 space, a pCurve can in no way represent the arc of a circle, which is a pity when you consider the importance of this type of curve and the surfaces constructed on circles like cones, cylinders, toruses, spheres and beyond to all the surfaces of revolution !

But the « Theory of Conics » revealed to us by the Greeks, highlights the fundamental relationships between curves that are as different in appearance as a circle, a hyperbola, an ellipsis and a parabola ; all these curves ARE conics, sections of a cone with a circular base on a more or less inclined plane to the axis of the cone. Seen from the apex of the cone, these are all identical and, notably, the base circle can always be considered as as the « conic » projection of the parabolic intersection of the cone by a plane parallel to a generatrix.



figures 3111.1 to 3111.3: the 4 conics as pL3 curves ; 3 arcs of a circle from a pL3, a pL4 and a pL5 ; 3 approaches to a complete circle with a pL3, a pL4 and a pL5.

To grasp this fundamental result, let us remember the analytical expression in R3 of a circle with a radius of 1 centered on the origin in the Oxy plane :

```
P(theta) = [ cos( theta ), sin( theta ), 0 ], theta in [-pi,pi]
```

Changing the variable u = tangent(theta/2) transforms this expression into a rational form :

```
P(u) = [ (1-u^2)/(1+u^2),2u/(1+u^2),0 ], u in ]-infinite,+infinite[
```

and passing into R4 will transform it into a polynomial expression :

```
P(u)   = [ 1-u^2, 2u, 0, 1+u^2 ], u in ]-infinte,+infinite[
```

As the four polynomials are of the second order, two of which are degenerate, each can be expressed in the polynomial form defining a parabola in R4 :

```
P(u)   = (1-u)^2.p0 + 2.(1-u).u.p1 + u^2.p2
       with : p0 = < 1, 0, 0, 1>
              p1 = < 1, 1, 0, 1> * sqrt(2)/2 // note that p1.t ≠ 1
              p2 = < 0, 1, 0, 1>
```

Thus, while it is still true that a pCurve cannot represent the arc of a circle, it can naturally generate one by conic projection, which is in fact the shift from R4 to R3, a simple perspective. We are thus able to apply reasoning to parabolas in R4 knowing that rendering it in R3 will indeed produce circles. We will see that this reasoning remains valid for immersed pForms.

In POVRAY/pFlibs syntax, we will thus define a parabolic arc in this way :

```
#local r = any value;
#local k = 1/sqrt(2);
#local quart_circle = array[3]
{      < r, 0, 0, 1 >,
       < r, r, 0, 1 >*k,
       < 0, r, 0, 1 >
}
```

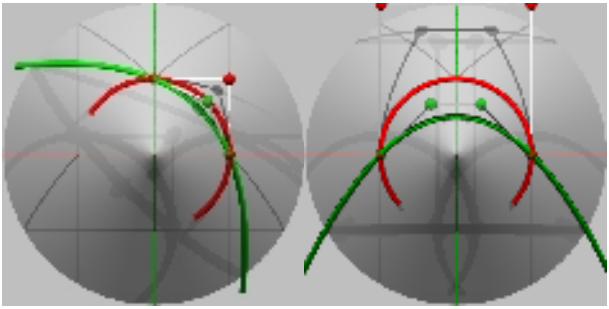
Projecting this parabolic arc from R4 into R3 produces a perfect arc of a circle, centered at the origin, with radius r and an angle of 90°. Other values of k produce ellipses and hyperbolas. Note that the implementation in POVRAY/pFlibs syntax always applies projection R4 -> R3 by default, so this is taken care of. The following call will indeed display the red arc of a circle :

```
draw(1,quart_cercle, finesse(3)+courbe(0.02)+couleur(<1,0,0,0 >))
```

The parabolic arc projects onto a circular arc, but it's possible to imagine other curves of the cone whose projection will also be the arc of a circle, in fact ANY cone curve will do. Here are two handy examples, two pCurves defined by 4 and 5 control points :

```
#local semi_cercle = array[4]
{      <  r,   0, 0, 1 >,
       <  r, 2*r, 0, 1 >/3,
       < -r, 2*r, 0, 1 >/3,
       < -r,   0, 0, 1 >
}

#local k = 1/sqrt(2);
#local semi_cercle = array[5]
{      <  r,     0, 0, 1 >,
       <  r,     r, 0, 1 >*k,
       <  0, 3/2*r, 0, 1 >*2/3,
       < -r,     r, 0, 1 >*k,
       < -r,     0, 0, 1 >
}
```

These two last representations can produce the complete arc of a circle by stretching the definition interval using the operator pFstretch(). In the first case (a pL3), it is necessary to work on the interval [-infinite,+infinite], producing nothing of interest, in the last case (a pL5), an interval [-k,1+k] with k=sqrt(2)/2, is sufficient to produce the full circle with a good distribution of points ; it so happens that this last representation corresponds to the diagonal of a biquadric, a surface constructed using three parabolas, ... still following ?



figures 3111.4 and 3111.5 : a pL3 and apL4 in R4 in green, and the corresponding arcs of a circle in R3 in red.

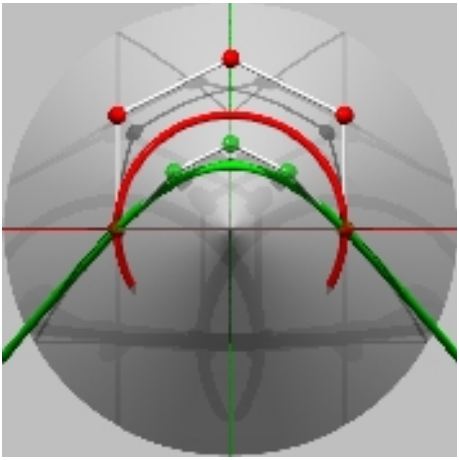


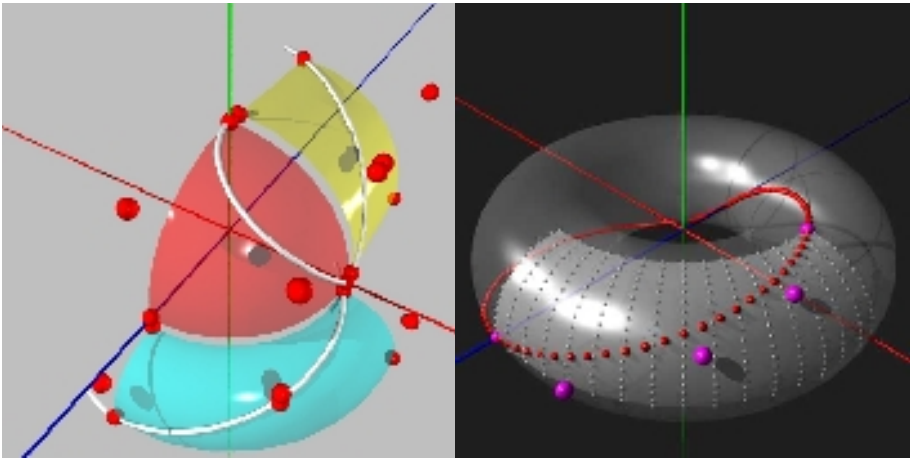
figure 3111.6 : a pL5 in R4 in green, and the corresponding arc of a circle in R3 in red.

312 cones, cylinders, toruses and spheres

Now that the arc of the circle has been incorporated into the pCurve family, the world of pForms of revolution opens up before us ! We're going to build these in two stages : firstly, we will compose the expressions of these surfaces « by hand », and secondly, we'll look for more general composition operators leading to the tube and affine forms. So for the moment, by choosing the suitable points to define a pSurface, it is easy to create fundamental surfaces of revolution like a portion of a cylinder ; torus or sphere. Here are the expressions in POVRAY/pFlibs syntax :

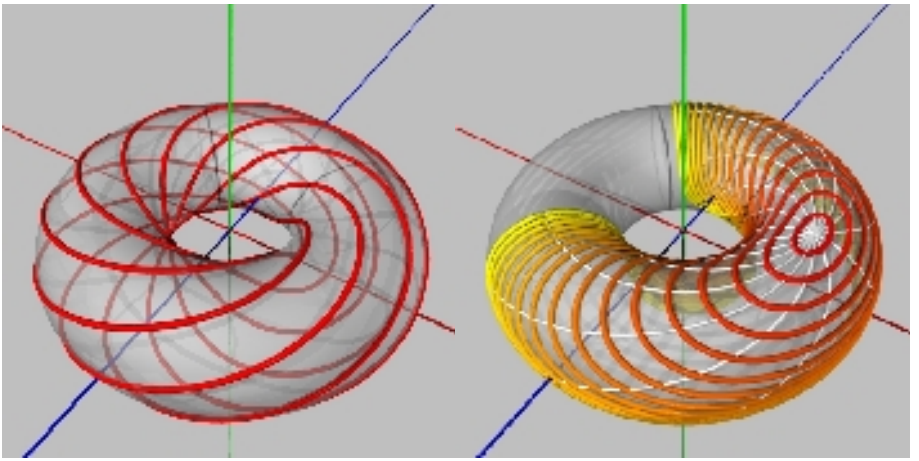
```
// quarter of a cylinder of radius R and height H :
#local k = sqrt(2)/2;
#local pCylinder = array[2]
{     array[3] {
            < R, 0, 0, 1 >,
            < R, R, 0, 1 >*k,
            < 0, R, 0, 1 >
            },
      array[3] {
            < R, 0, H, 1 >,
            < R, R, H, 1 >*k,
            < 0, R, H, 1 >
            }
}
// sixteenth of a torus of radii R1 and R2:
#local R12 = R1+R2;
#local R2 = abs(R2);
#local pTorus = array[3]
{     array[3] {
            < 0,   0, -R12, 1 >,
            < 0,  R2, -R12, 1 >*k,
            < 0,  R2,  -R1, 1 >
            },
      array[3] {
            < R12,  0, -R12, 1 >*k,
            < R12, R2, -R12, 1 >*k*k,
            <  R1, R2, -R1,  1 >*k
            },
      array[3] {
            < R12,  0,  0, 1 >,
            < R12, R2,  0, 1 >*k,
            <  R1, R2,  0, 1 >
            }
}

// eighth of a sphere of radius R :
```

```
#local pSphere = array[3]
{       array[3] {
                < R, 0, 0, 1 >,
                < R, R, 0, 1 >*k,
                < 0, R, 0, 1 >
},
        array[3] {
                < R, 0, -R, 1 >*k,
                < R, R, -R, 1 >*k*k,
                < 0, R,  0, 1 >*k
                },
        array[3] {
                < 0, 0, -R, 1 >,
                < 0, R, -R, 1 >*k,
                < 0, R,  0, 1 >
                }
}
```

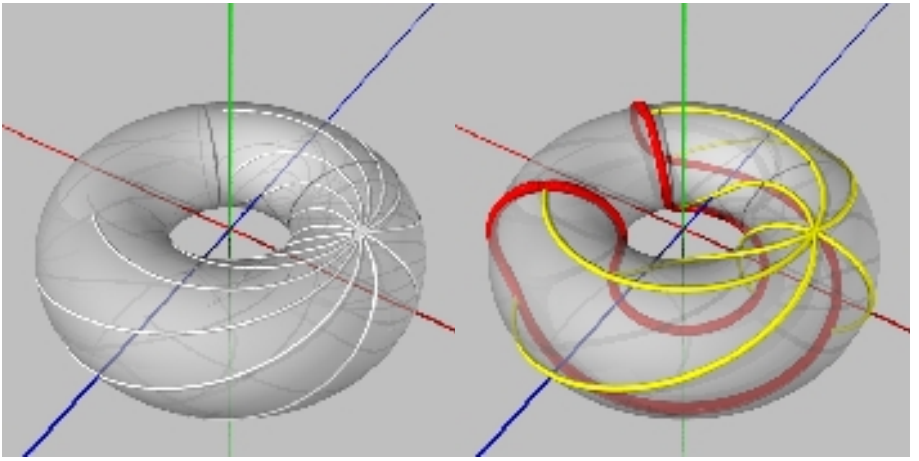


figures 312.1 and 312.2 : 3 pS33 cylinder, torus and sphere and their diagonals; diagonal (pL5) on 1/16th of a torus (pS33) drawn in the interval [-k,1+k], where k=sqrt(2)/2.

Here again it will be possible to use the operator pFstretch() to create complete spheres, cylinders and toruses (of 360°), as we saw for the arc of the circle, using the right pCurves, pL5 being a priori the best adapted to interval [-k,1+k].



figures 312.3 and 312.4 : in a torus, red parallel straight lines, and concentric circles from red to yellow..

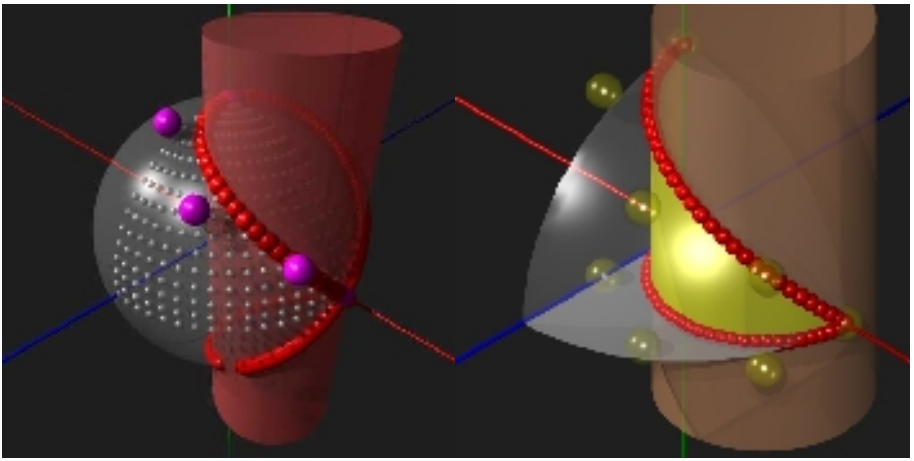


figures 312.5 and 312.6 : in a torus, white radiating segments, and red circle with its yellow radiuses.

313 applications

The pS33 and its diagonal pL5 play an important role in several applications. With the diagonal of a portion of a sphere built on a pS33, it is thus possible to find the curve known as Viviani's window. By embedding a pL5 in a sphere or a torus producing the arc of a circle, we discover an example of point confinement leading to infinity in our « straight » space. And the curve known as the Trefoil Knot (knot with three crossings) appears as a simple straight line in a toric geometry.

3131 Viviani's window



figures 3131.1 and 3131.2: Viviani's window is an immersed segment (pL5) in an 1/8th of a sphere (pS33) drawn in the interval [-k,1+k], where k=sqrt(2)/2.

The portion of the sphere (1/8th) is a pS33 whose diagonal we know is a pL5. The vertical projection of the five control points on the equatorial plane happens to produce the control points of a semi-circle ; thus we find the curved curve called Viviani's window, the intersection of a sphere and a cylinder. The classic expression of the curve is the following :

```
x = R*(1+cos(t)),
y = R*sin(t),
z = 2R*sin(t/2)
```

which is not complicated in itself, but the definition of the tangent trihedron at each point involves far more complex expressions. Considering the curve as a pCurve gives access to 5 control points that help to draw the curve, as well as to all the pForm operators, notably the one that reduces the tangent trihedron at each point. Plus, knowing that this curve is a simple segment immersed in a portion of a

sphere is mentally satisfying !

Here in POVRAY/pFlibs syntax are three methods used to construct Viviani's window :

```
Definition of an eighth of a sphere :

#local R = 1/2;
#local k = sqrt(2)/2;
#local pSphere = array[3]
{     array[3]
      {     < R, 0, 0, 1 >,
            < R, R, 0, 1 >*k,
            < 0, R, 0, 1 >
      },
      array[3]
      {     < R, 0, -R, 1 >*k,
            < R, R, -R, 1 >*k*k,
            < 0, R,  0, 1 >*k
      },
      array[3]
      {     < 0, 0, -R, 1 >,
            < 0, R, -R, 1 >*k,
            < 0, R,  0, 1 >
      }
}
draw( 2, pSphere,
finesse < 3,3 > + surface( 0.01 ) + couleur( < 1,1,1,0.5 >) )

1) construction by the embedding method of the diagonal defined and
subdivided in the surface between points < 0,0 > and < 1,1 >, then
expressed in R3 :

#local diag = array[2] { < 0,0,0,1 >, < 1,1,0,1 > }
#local pDiag = pFsubdivision( 1, diag, <4,0,0> )
pFimmersion( 2, pSphere, 1, pDiag )
draw( 1, pDiag, point(0.02) + couleur(< 1,1,1,0.8 >) )

2) construction by calling the diagonalisation operator :

#local diag = pFdiagonalisation( 2, pSphere )
draw( 1, diag, finesse(3)+point(0.02)+couleur(< 1,0,0,0.9 >) )
draw( 1, diag, point(0.04) + couleur(< 1,1,0,0.8 >) )

3) construction by defining the diagonal in the surface between points
< 0,0 > and < 1,1 > , and calling the general operator
curve_in_surface():
```

```
#local diag = array[2] { < 0,0,0,1 >, < 1,1,0,1 > }
#local diag = courbe_in_surface( diag, pSphere )
draw( 1, diag, finesse(3)+point(0.02)+couleur(< 1,0,0,0 >) )
draw( 1, diag, point(0.04+couleur(< 1,1,0,0.8 >) )
```

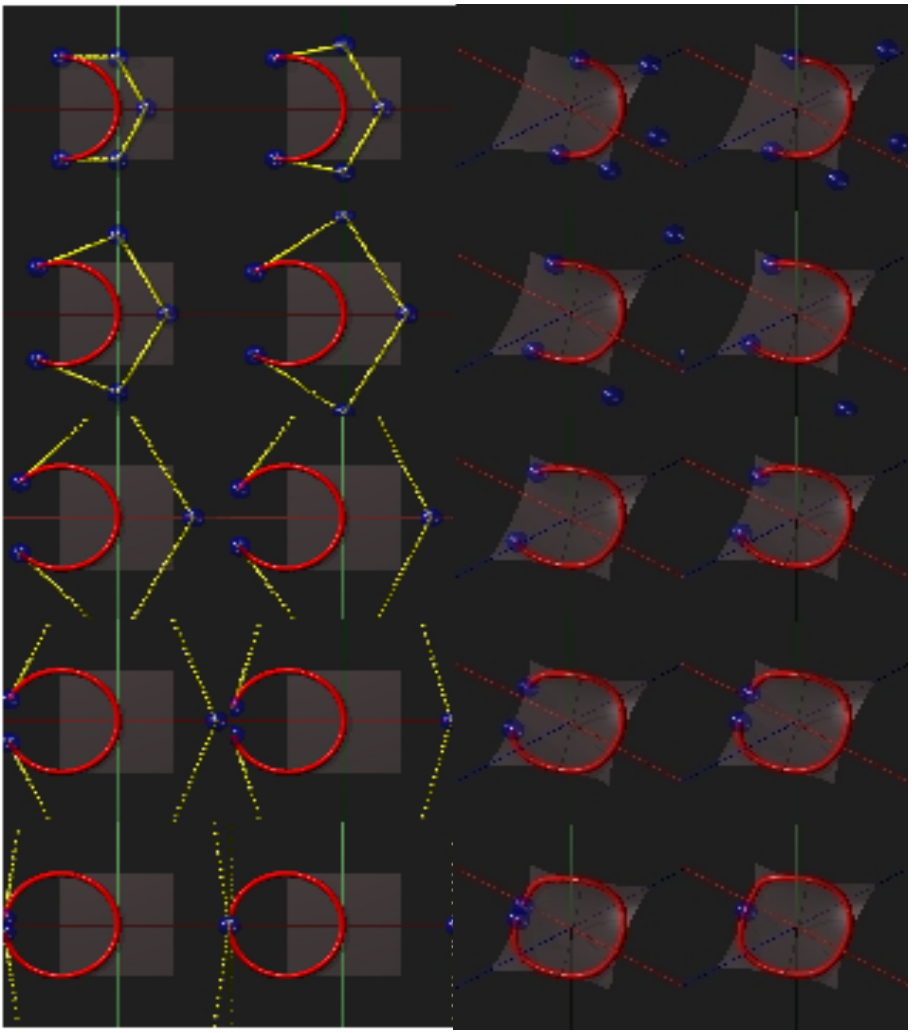
The first method of calculation by immersion is general, it is valid for any ipForm immersed in any pForm, but it will not give the control points in R3. The second method only deals with diagonals, and thus iSegments, but does have the advantage of producing control points. The last method is for now only implemented for pCurves immersed in pSurfaces, but will produce control points for any iCurve. Obviously, all three methods provide the same curve in the present case. Note that by flattening the sphere, we obtain a disc projection whose diagonal is in fact a semi-circle.

3132 immersed circles

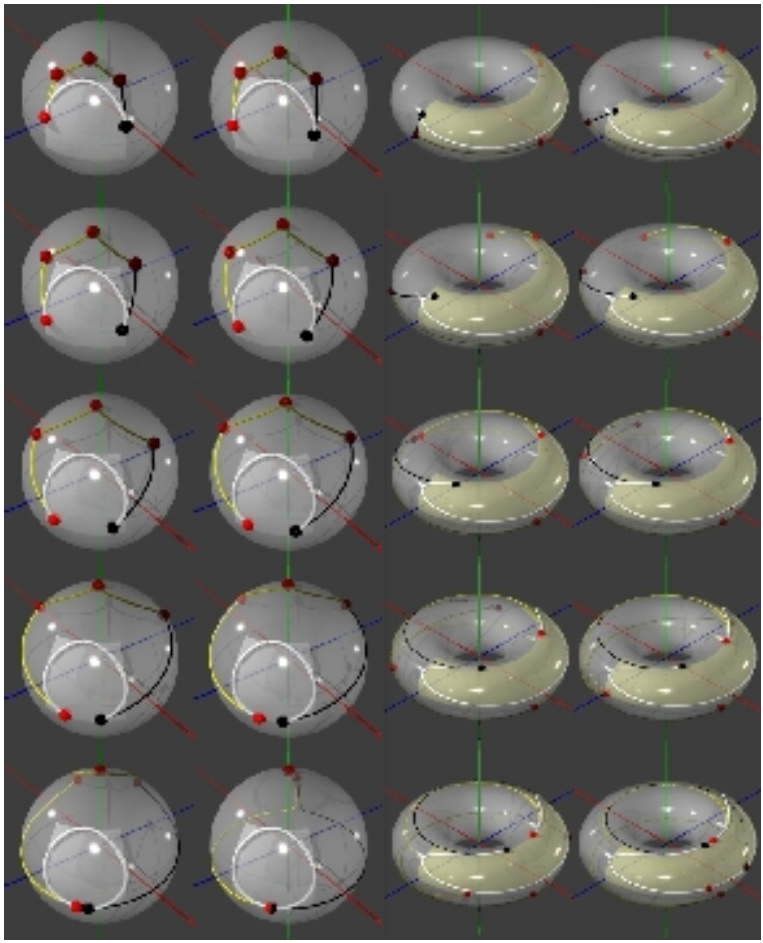
We saw how easy it was to place the 5 points of a pL5 to construct a parametered semi-circle in the standard interval[0,1]. Using operator pFstretch() with interval definition [-k,1+k], where k = sqrt(2)/2 = 0.707, we can produce a complete circle with an acceptable point distribution. The problem is that two of the control points lead to infinity, and it is never comfortable working with infinite points. Figure 3132.1 represents the case in which the arc of the circle is immersed in a flat facet, in fact in the surrounding euclidian space ; in figure 3132.2 the facet is any slightly warped pSurface, which doesn't actually change that much.

By embedding a circle in a finite surface, these points are confined and their behaviour can be more effectively visualised. Embedding the pL5 generating a semi-circle in a pS33 generating a torus, clearly represents the two points that initially lead to infinity when passing from interval [0,1] to interval [-k,1 +k]. Figures 3132.3 and 3132.4 correspond to embedding in a pS55, whose 25 control points are chosen to produce a sphere and a torus, finite and complete surfaces in which the control points of the immersed arc of a circle developing into a complete circle are confined to a finite distance.

Here we get a glimpse the apparent similarity with Riemann's sphere as a representation of projective space, of Poincaré's straight lines in the hyperbolic plane, and even more so of attempts at representing the universe as having an elliptical, parabolic, hyperbolic, and today, a toroidal topology. And also, of the use of immersed pForms to represent cosmological theories. An avenue worth exploring !



figures 3132.1 and 3132.2 : in any plane or curved pSurface, two of the control points of the arc of a circle defined by a pL5 will lead to infinity.

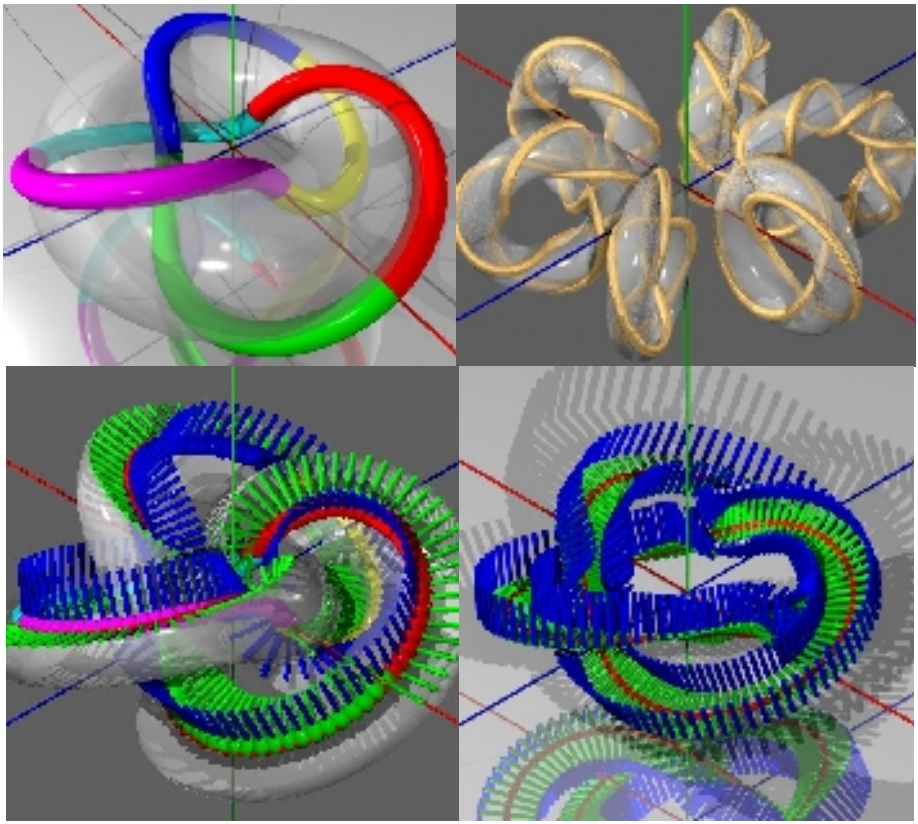


figures 3132.3 and 3132.4 : left, a white semi-circle (pL5) and its control polygon (segments black to yellow) immersed in a sphere, the definition interval is gradually stretched from [0,1] to [k, 1+k] where k=sqrt(2/2), the semi-circle becomes a full circle, the control points remain confined to the surface of the sphere ; right, same example with embedding in a torus.

3133 straight lines that get into knots

The staircases in Montreal are well-known for their rather surprising style. They climb up the front of small two- or three-storey apartment buildings, twisting around to serve every floor, without hindering the windows facing the street, much like MC Escher's impossible stairways. The sculpture shown in figure 3133.1 is the entry to a competition won by Montreal architect, Guillaume Labelle (site: http://labelle.spacekit.ca/), based on the idea of making a staircase in the form of a curve known as the Trefoil Knot (triple loop knot). While the analytical expression of this curve is fairly simple, determining the tangent axes (Serret-Frenet) necessary to work out the steps and handrails calls on expressions that become fairly heavy and do not convey much. Segments immersed in a torus provide a solution that is friendlier to analyse, manipulate and represent the various knots drawn on the torus. The following illustrations show some of the stages that were used to work out the steps and handrails of the staircase produced.

figure 3133.1 : a sculpture in Montreal, a straight line in toric space !!



figures 3133.2 to 3133.5 : the complete curve is the concatenation of 6 diagonals ; 6 examples of knots from segments immersed at different angles (producing continuous or noncontinuous curves) ; applied to the Montreal staircase by representing a distribution of Serret-Frenet trihedrons, then the binormals (sketching the steps) and normals (sketching the handrails). Note that the handrails actually produced are inclined at 45° to be orthogonal to the steps, parallel to the risers.

32 composed forms

Summary of this section :

Up to now the pForms have been constructed « by hand », the control points being positioned « a priori ». This is notably the case for the portions of the cylinder, torus and sphere studied above. We are now going to automate the creation process, giving access to more complex shapes.

321 meshes

In the chapter on approaching pForms, we were able to construct pForms « by hand » : by applying operator MIR() to three pL3 (parabolas), we created a pS33, then pL5, as well as pV222 (curved cubes), etc... It is helpful to define the operators that automatically produce pCurves, pSurfaces and pVolumes controlled by a larger number of points ; we find a create_line() operator producing a pCurve starting as rectilinear and controlled by an arbitrary number of points, a create_facet() operator producing a pSurface that starts flat and finally a create_cube() operator producing a pVolume that starts as perfectly cubic, the three pForms produced being of a given size and centered at the origin parallel to the axes. Here is an example implemented in POVRAY/pFlibs in the case of the cube :

```
#macro create_cube( n1, n2, n3, ttt )  // ttt is the size
  #local f = array[n1]
  #local ff = array[n2]
  #local fff = array[n3]
  #local i=0; #while (i< n1)
    #local j=0; #while (j< n2)
      #local k=0; #while (k< n3)
        #local p = < -0.5+i/(n1-1), -0.5+j/(n2-1), -0.5+k/(n3-1),1 >;
        #local fff[k] = p * ;
      #local k=k+1; #end
      #local ff[j] = fff;
    #local j=j+1; #end
    #local f[i] = ff
  #local i=i+1; #end
  f
#end
```

One application, among others, of a curved cube beyond the very basic pV222 will be presented in the paragraph on deformations.

322 cross surfaces, surfaces of revolution

It is possible to define an operator producing pSurfaces from a « section » curve defined in an Oxz plane, displaced along the Oy axis according to a « profile » curve. In the case where the section curve is the arc of a circle, a surface of revolution is produced with the particular cases of cylinders, toruses and spheres as seen above. Here is an implementation in POVRAY/pFlibs syntax :

```
/*
  CROSS SURFACE :
  input: two pLn planes in Oxy, profile and section
  output: a pSmn
  nota: particular cases: prisms and surfaces of revolution
        curved profile to study
  call: #local surf = cross( c1, c2 )
*/
#macro cross( c1, c2 )
    #local nb1 = taille( c1 );    // profile curve
    #local nb2 = taille( c2 );    // section curve
    #local surf = array[nb1]      // cross surface
    #local i = 0; #while (i< nb1) // for each profile point
        #local profil = c1[i];    // a point of the profile
        #local pp = array[nb2]
        #local j=0; #while (j< nb2)// for each section point
            #local section = c2[j];// a section point
        #local pp[j] = <
              section.x*profil.x,    // base on x
              section.t*profil.y,  // along axis y
              section.y*profil.x,  // base on z
              section.t*profil.t  >  ;
        #local j=j+1; #end
        #local surf[i] = pp
    #local i=i+1; #end
    surf
#end
```

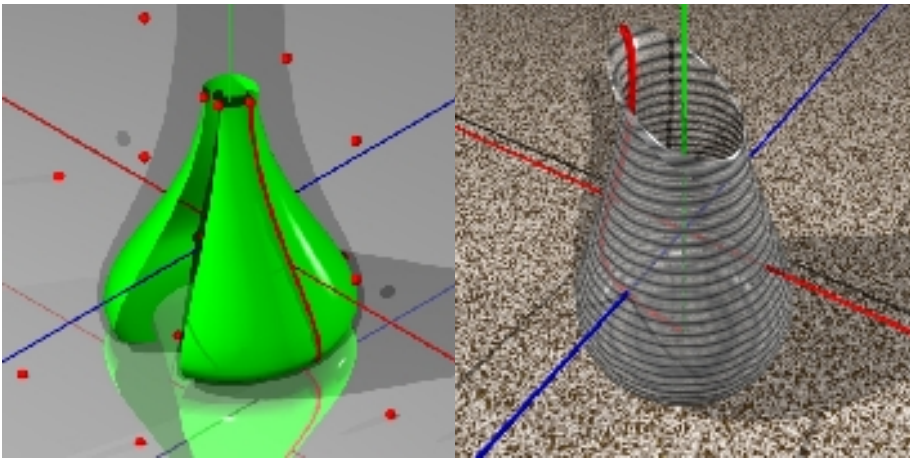


figure 322.1 to 322.2 : a surface of revolution and another deformed at the top.

Note that in the case of a nonrational section and profile (whose points all have a value of t equal to 1), the expression of a current point becomes :

```
#local pp[j] = < section.x*profil.x,   // base on x
                 profil.y,             // along Oy
               section.y*profil.x,     // base on z
               1 >;                    // always
```

an expression where, for coordinates (x,z) of the resulting point we recognize the application of a change of scale in the relation of profile.x to the current point (x,y) of the section, and for the coordinate in y of the current point, application of the value profile.y of the current point of the section ; the section curve is clearly displaced along axis Oy following the profile curve.

Of course, one of the big advantages of pSurfaces is the fact they ARE pSurfaces, i.e., that it's possible to deform them through the intermediary of their pCurves and generating points. This opens the door to creating a whole range of freer shapes based on initially conical sections, like the various sections of the cabin of a plane (an Airbus, for instance), from the nose to the tail, perfect circles, from the pear-shaped cockpit and the widening of the straight line of the wings and the tail-plane - all without the slightest discontinuity, using the same curve in which only the control points vary according to the context. The illustration above shows a more modest example, a pot turned on the wheel, whose top has been deformed by the potter's hand to make a lip for pouring.

323 pipe surfaces

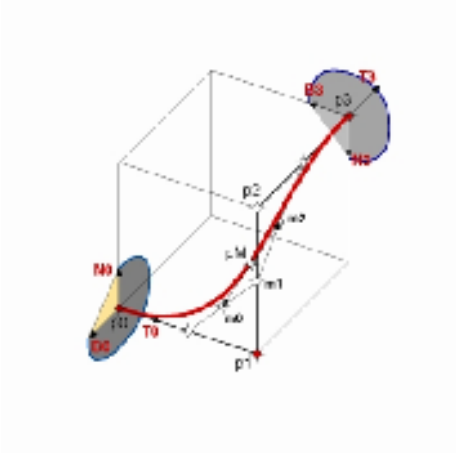


figure 323.1: a pipe surface is the envelope surface of a section curve aligned to the Serret-Frenet trihedron of the curved path it follows.

Taking a curve called « path » and a curve called « section », it is fairly easy to create a pipe surface using the Serret-Frenet trihedron of the path curve as the transformation operator (translation + rotation) of the points on the section curve. The pipe produced can be symbolically written in the form -cf diagram 323.1- :

```
pipe = path + TNB(path) * section
```

where **TNB** is a 3x3 matrix constructed on the tangent, normal and binormal vectors.

But there is a problem here ! Up to now, the combinations studied produced pSurfaces ; a torus constructed using formulas given in in the « surfaces of revolution » section is a set of 9 points distributed in space such that operator MIR() produces a perfectly toric surface. But trying to construct the torus by distributing the 3 control points of the arc of a path circle according to the 3 local axes (Serret-Frenet trihedrons) at each of these points, will not produce the desired result. The 3 intermediate control points are not placed correctly, and the MIR() operator will not produce a torus.

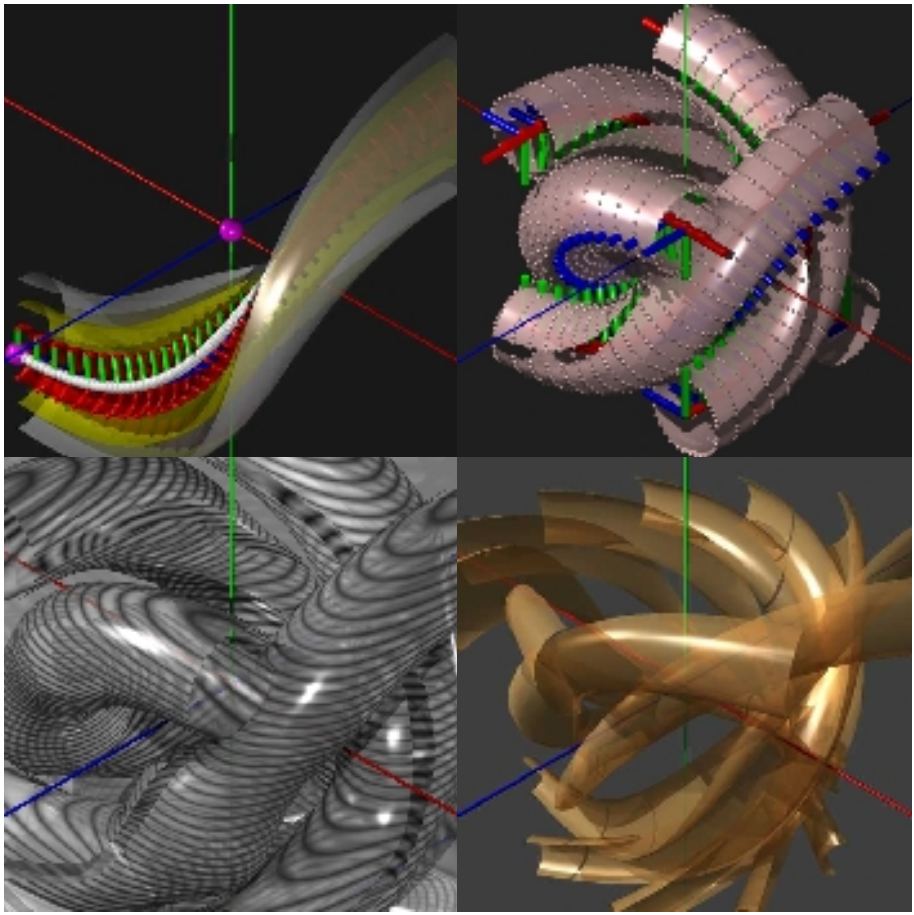
The only solution is to apply the MIR() operator to the path curve before calculating the surface control points. The result is obtained, but the pipe() operator cannot be said to produce a pSurface, as understood up to now.

Here is an implementation in POVRAY/pFlibs syntax, in the form of a pipe() operator :

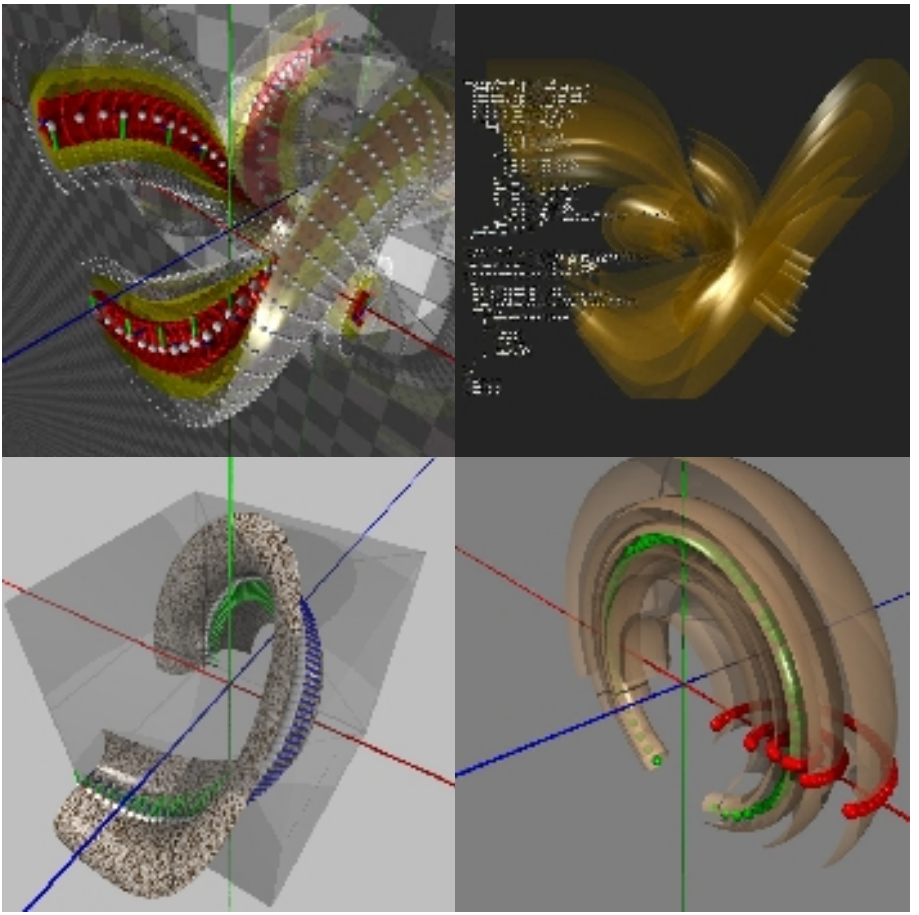
```
/*
  PIPE SURFACE
  input: a path pLm and a section pLn
  recursion values at u and v (pre-subdivision)
  output: a pSmn (that can be subdivided)
  nota: works with rational curves
  call:  #local surf = pipe( path, section, 4, 3 )
*/
#macro pipe(che, sec, r1, r2)
  #local chemin  = pFsubdivision( 1, che, < r1,0,0,0 > )
  #local section = pFsubdivision( 1, sec, < r2,0,0,0 > )
  #local vmax = taille(chemin);
  #local umax = taille(section);
  #local tube = array[vmax]
  #local pp = array[umax]
  #local i = 0; #while (i< vmax)
    #local mat = pFgetPijk( 1, che, i/(vmax-1) )
    #local j = 0; #while (j< umax)
      #local pp[j] = pFtransform( mat, section[j] );
    #local j=j+1; #end
    #local tube[i] = pp
  #local i=i+1; #end
  tube
#end
```

Note that unlike the operator cross() seen previously, this pipe() operator incorporates the recursion parameters used by the pFsubdivision() operators ; we saw how it was necessary, in fact, to apply subdivision to the path curve before composition, and for reasons of call symmetry, we also subdivide the section curve before composition.

Comment : the non commutativity that has appeared between operator MIR() and operator pipe() with the effect of producing two different surfaces, leads to distinguishing two classes of composition operators : those that commute with MIR() and the others. Only the group of pascalian forms armed with commutative operators producing the elements of this group, can be considered as a safe basis for a unitary geometry of curved forms. The others need to be manipulated with maximum precaution, regardless of their practical use, pending a generalisation of pForms that naturally includes pipes.



figures 323.1 to 323.4 : some examples of duplicated pipes in rotation.



figures 323.5 to 323.8 : a few aliens!

324 affine surfaces

Surfaces resulting from the product (profile x section) and pipe surfaces can be considered as specific cases of surfaces expressed in the following general case :

```
surface = AFFINE * curve
```

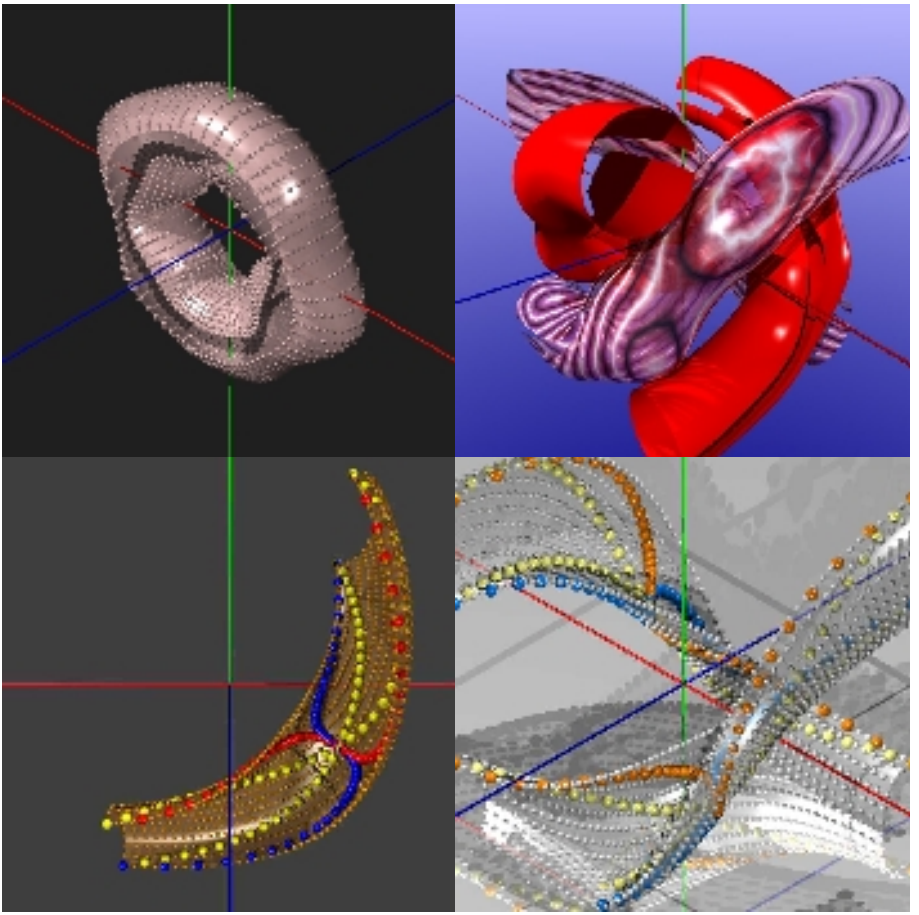
in which AFFINE is a 4x4 matrix that can be read like this :

```
          |  Nx*Kx Bx    Tx    dx  |
AFFINE =  |  Ny    By*Ky Ty    dy  |
          |  Nz    Bz    Tz*Kz dz  |
          |  Sx    Sy    Sz    dt  |

with   N:     normal vector
       B:     binormal vector
       T:     tangent vector
       d:     translation vector
       K:     scalar vector
       S:     deformation vector
```

the first four vectors being linked to a path curve, vector k giving modulation and vector S tangential deformation (shear), and any other interpretation of the 16 coefficients of the 4x4 matrix that can be envisaged a priori.

Here in POVRAY/pFlibs syntax, is an implementation of an extension of operator pipe(), waving_pipe() applying a sinusoidal wave to the produced pipe surface, with a given amplitude and frequency :

```
/*
WAVING PIPE SURFACE
#local surf = waving_pipe( path, section,< 4, 3 >,< 0.5, 5 > )
*/
#macro waving_pipe(che, sec, r, ondula )
  #local chemin  = pFsubdivision( 1, che, < r.x,0,0,0> )
  #local section = pFsubdivision( 1, sec, < r.y,0,0,0> )
  #local vmax = taille(chemin);
  #local umax = taille(section);
  #local tube = array[vmax]
  #local pp = array[umax]
  #local i = 0; #while (i< vmax)
    #local uu = i/(vmax-1);
    #local mat = pLgetFrenet( che, uu )
    #local coeff = 1 + ondula.x*sin(2*pi*uu*ondula.y );
    #local temp = mat
    #local temp[0][0] = mat[0][0]* coeff;
    #local temp[1][1] = mat[1][1]* coeff;
    #local temp[2][2] = mat[2][2]* coeff;
    #local j = 0; #while (j< umax)
      #local pp[j] = pFtransform( temp, section[j] );
    #local j=j+1; #end
    #local tube[i] = pp
  #local i=i+1; #end
  tube
#end
```



figures 324.1 to 324.4 : portion of a torus of sinusoidally variable radius ; other variable pipes with lines drawn underneath.

Comment 1 : what was said in section 323 on pipe surfaces equally applies to this section : operator waving_pipe() is not an operator producing a pSurface and must be used with precaution.

Comment 2 : this in no way detracts from the general character of operator AFFINE, a linear operator that requires no use of metrics, and uses a normalisation and orthogonalisation operation. This will need a closer look...

325 parallel forms

Pipe surfaces with a circular section are an example of a form obtained by looking for a set of points lying at a constant distance from a given curve ; the surface can be considered to be produced by displacing the centre of a sphere of a constant radius along the path curve. A similar problem arises in looking for surfaces lying at a constant distance from a given surface, or from parallel surfaces.

Implementation in POVRAY/pFlibs syntax is the following :

```
/*
  PARALLEL SURFACES
  create a surface lying at distance dd of a surface
  other possible cases with modulation of dd
*/
#macro parallel_surface( surf, dd )
  #local m = taille( surf );
  #local n = taille( surf[0] );
  #local S = surf
  #local i = 0; #while (i< n)
    #local j=0; #while (j< m)
      #local mat = pFgetPijk( 2,surf,< j/(n-1),i/(m-1),0,1 > )
      #local nn = < mat[0][0], mat[1][0], mat[2][0] >;
      #local pp = S[j][i];
      #local pt = pp.t;
      #local qq = < pp.x/pp.t, pp.y/pp.t, pp.z/pp.t >;
      #local qq = qq + nn*dd;
      #local S[j][i] = < qq.x*pt, qq.y*pt, qq.z*pt, pt >;
    #local j=j+1; #end
  #local i=i+1; #end
  S
#end
```

Comment : as was seen with the pipe, constructing parallel surfaces involves the local axis obtained by calling up operator pFgetPijk(). Hence a parallel surface is not strictly speaking a pSurface.

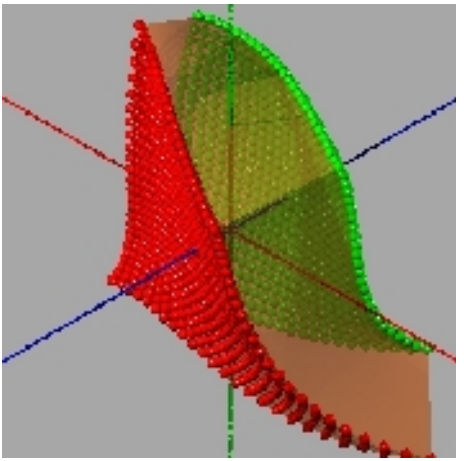
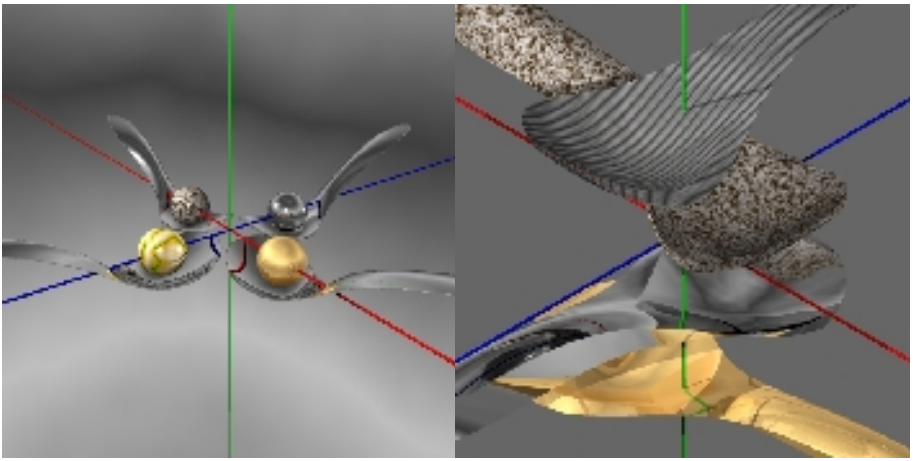


figure 325.1: two parallel pSurfaces containing a pVolume.



figures 325.2 to 325.3: volumes constructed on parallel surfaces, to produce thick spoons in granite, wood, chrome and striped plexiglas !

326 developed surfaces

Due to the properties of the derivatives of pCurves (which are pCurves of degree-1), the extremity of the tangent vector (non normalised) at a point travelling through a pLn displaces on a pLn. The surface generated by this tangent vector is thus a pS2n, a ruled surface that we know is developable, i.e., it can be rolled out flat without folds or tearing (a cone is a developable surface, but a sphere is not). Implementation in POVRAY/pFlibs is left to the reader, as an exercise ;) -cf diagram 326-.

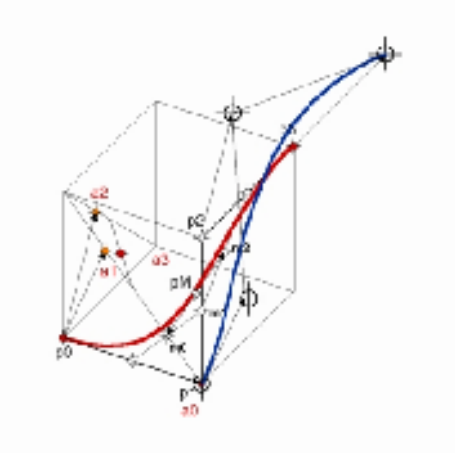


figure 326 : a pS32 (ruled cubicoid) constructed on two pL4 (cubics) (red and blue).

Comment : the study of pipe surfaces and parallel surfaces has shown what happens when normalisation is introduced to operator pFgetPijk(), the result does not conform to the coherent whole of non metric pSurface geometry. A study of developed surfaces illustrates the interest of using a non normalised tangent vector, and we can imagine that a pFgetPijk() operator that did not normalise the three tangent, normal and binormal vectors would produce the desired results, entering the family of pSurfaces. This needs to be looked at closer !

33 special linear combinations

Summary of this section :

If any linear combination of points whose sum of coefficients is equal to 1 produces a « valid » point (invariant with a change of axis), the same goes for the linear combinations of pForms in general, produced in the final analysis by valid combinations of points :

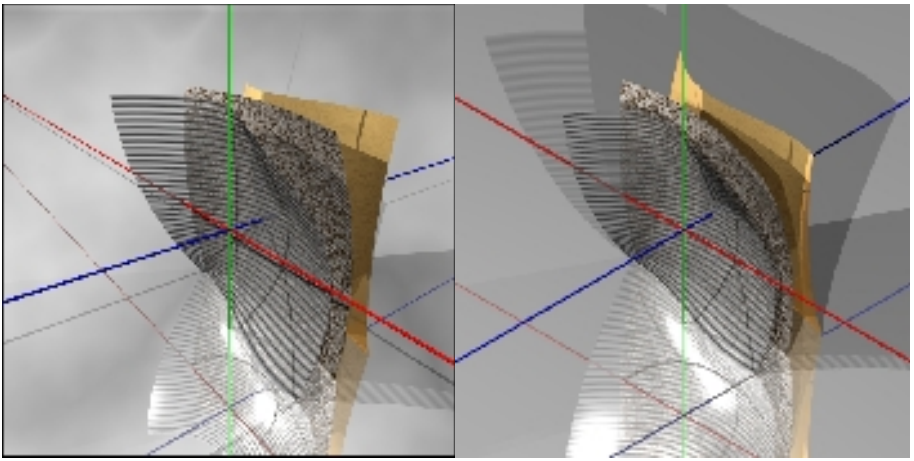
$$pF = \sum_i ki.pFi \qquad \text{with } \sum_i ki = 1.$$

We already know the midpoint of two or N forms, the forms produced using operator MIR() and those obtained via the commutative composition operators, all linear combination pascalian forms. We are now going to look at other, non pascalian linear combinations, and examine to what extent they could enter this range.

331 symmetrical forms

Given two points P1 and P2, expression « P = 2*P1 - P2 » produces point P that is the symmetrical form of P2 in relation to P1. We could replace the points with any pForm, and for instance play with the combination « S = 2*S1 - S2 » to produce the symmetrical form of S2 in relation to form S1; in the specific case where S1 is a plane, we obtain plane symmetry. Here is an implementation in POVRAY/pFibs syntax in the case of surfaces :

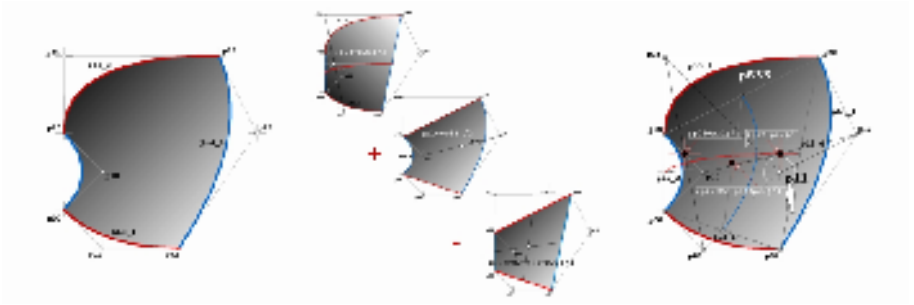
```
/*
SYMMETRICAL SURFACES
create a surface that is symmetrical to another surface
*/
#macro symmetrical_surface( s1, s2 ) // surf = 2*s1 - s2
  #local M = taille( s1 );
  #local N = taille( s2[0] );
  #local surf = array[M]
  #local i=0; #while (i< M)
    #local pp = array[N]
    #local j=0; #while (j< N)
      #local pp[j] = 2*s1[i][j] - s2[i][j];
    #local j=j+1; #end
    #local surf[i] = pp
  #local i=i+1; #end
  surf
#end
```



figures 331.1 and 331.2 : the gold leaf and the striped transparent leaf are symmetrical in relation to the granite leaf.

Any surface combination can be treated in the same way, and we will see a simple and powerful application of this in Coons surfaces.

332 coons surfaces



figures 332.1 to 332.3 : a Coons surface on 4 pL3 is a linear combination of 3 pSurfaces ; pS1+pS2-pS3, is a pSurface.

One particularly useful type of combination is that discovered by Coons. The following combination of three surfaces :

```
S = S0 + S1 - S2     ( note : 1+1-1 = 1, so it's OK )
```

is used in defining squares with the property of interpolating any 4 concurrent curves two by two, forming a continuous limit (homeomorphic to a complete circle).

Restricting ourselves to the case of pCurves, let's consider 4 of these (pLn1_1, pLn2_2, pLn3_3, pLn4_4) concurrent two by two to points P00, P01, P10, P11. Using insertion to equalise the number of control points of the 4 pCurves to a common value n (n1 = n2 = n3 = n4 = n), the combination can be rewritten in the form :

```
S =  MIR( pLn_1, pLn_3 )         // S0
  +  MIR( pLn_2, pLn_4 )         // S1
  -  MIR( P00, P01, P10, P11 )   // S2
```

S0 is a ruled surface constructed on pLn1_1, pLn3_3, S1 a ruled surface constructed on pLn2_2, pLn4_4, and S2 a curved facet constructed on the four meeting points. As the number of control points is identical for S0, S1 et S2, this expression can be applied to the midpoint :

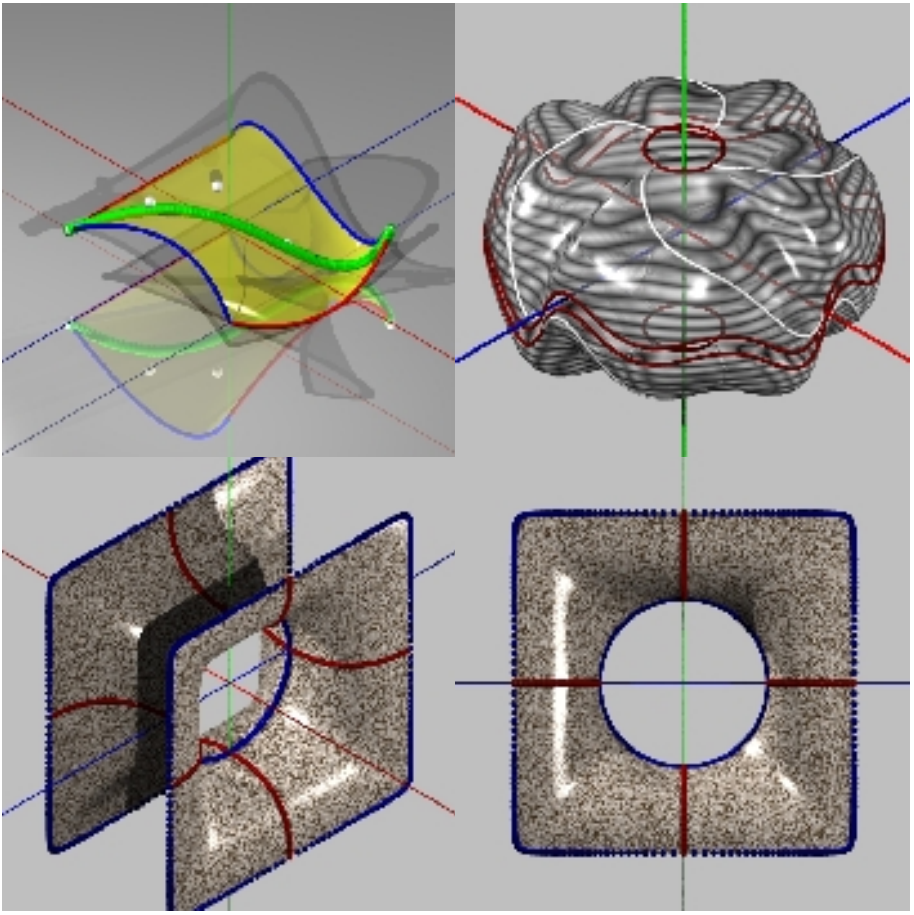
```
Pm = MI( pLn_1, pLn_3 )
   + MI( pLn_2, pLn_4 )
   - MI( P00, P01, P10, P11 )
```

point of departure to launch a recursion and obtain a pSurface. The result is that a Coons surface constructed on pCurves is a pSurface (but we already knew that).

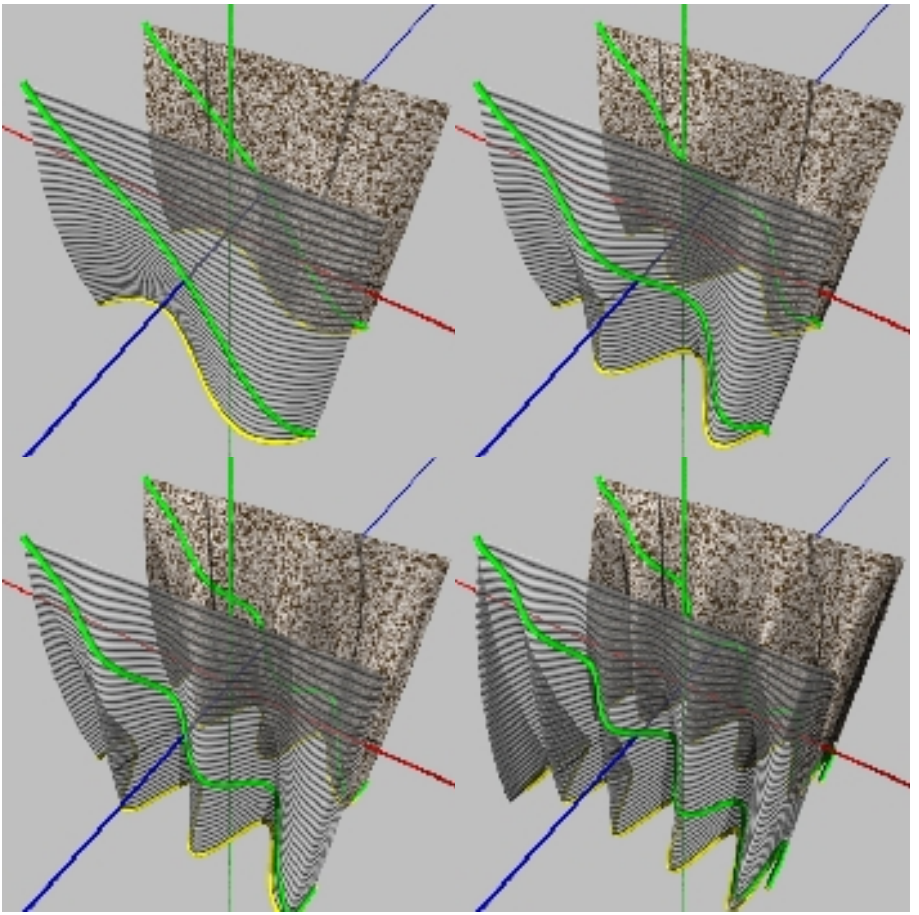
Other, more complex, linear combinations could be dealt with, notably Gordon surfaces, interpolating a series of curves (think for instance of generating a Joystick). Implementation in POVRAY/pFlibs syntax is given in the appendix (the code is like that for symmetrical surfaces presented above), here's an example of a call producing a Coons square and its diagonal from 4 pCurves of different degrees (pL3, pL4, pL2, pL3) :

```
#local L1 = array[3] {< -1,0,-1,1 >,< 1/2,1,-1,1 >,< 1,0,-1,1 >}
#local L2 = array[4] {< -1,0,1,1 >,< -1/2,1,1,1 >,< 1/2,-1,1,1 >,
<1,0, 1,1 >}
#local L3 = array[2] {< -1,0,-1,1 >,< -1,0,1,1 >}
#local L4 = array[3] {< 1,0,-1,1 >,< 1,-1,1/2,1 >,< 1,0, 1,1 >}
#local coons = create_coons( L1, L2, L3, L4 )
#local diag = pFdiagonalisation( 2, coons )

draw( 2, coons,
 finesse(< 3,3 >)+surface(LISSE)+ma_couleur( < 1,1,0,0.5 > ) )
draw( 1, L1, finesse(4)+courbe(0.02)+ma_couleur( < 0,0,1 > ) )
draw( 1, L2, finesse(4)+courbe(0.02)+ma_couleur( < 0,0,1 > ) )
draw( 1, L3, finesse(4)+courbe(0.02)+ma_couleur( < 1,0,0 > ) )
draw( 1, L4, finesse(4)+courbe(0.02)+ma_couleur( < 1,0,0 > ) )
draw( 1, diag, point( 0.05 )+ma_couleur( < 1,1,1 > )
draw( 1, diag, finesse(4)+point(0.05)+ma_couleur(< 0,1,0 > ) )
```

The following figures show the extent to which a simple linear combination (S = S1 + S2 – S3) can turn out to be full of possibilities, opening the road to other combinations between pForms, starting with pVolumes.



figures 332.4 to 332.7 : Coons pSurface and its diagonal, a shell comprised of two symmetrical surfaces each constructed on a small red circle and a big wavy circle, and a minimal pseudo-surface constructed on 8 Coons, AXO and seen from the front.



figures 332.8 à 332.11 : four Coons pSurfaces constructed on a segment and pCurves waving at increasing frequency, curtains in the wind !

34 concatenations, splines

Summary of this sectionn :

- 341 non interpolating splines
- 342 interpolating splines
- 343 NURBS

When more than three or four control points are required constructions can be a problem. So imagine concatenating several pCurves controlled by the same reduced number of points by defining connecting rules to ensure the different continuity conditions for the tangents, curvatures and torsion. Several approaches are possible, depending on whether or not we want to interpolate the control points.

341 non interpolating splines

The best known approach is the « B_spline » approach that we will outline briefly.

A « window » of the size of 2, 3, or 4 points is slid over N given points Qi, incrementing by one point in each case.

1) With a 2 point window (Qi,Qi+1) two points are defined as follows :

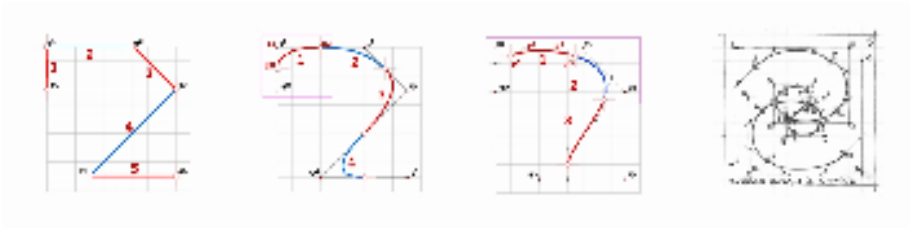
```
p0 = Qi
p1 = Qi+1
```

used as the control points of a pL2 (segment) per window, we easily obtain the following segments forming the base polygon, a B-spline of degree 1.

2) With a 3 point window (Qi,Qi+1,Qi+2) three points are defined (in i/2 for i=[0,2]) as follows :

```
p0 = 1/2(Qi + Qi+1 )
p1 = 1/2(   2.Qi+1 )
p2 = 1/2(     Qi+1 + Qi+2)
```

used as the control points of a pL3 (parabola) per window. In constructing, the successive curves connect up, with collinear tangents of the same module. The flatness of the parabolas implies discontinuities of torsion which is null inside the parabolas but can be infinite at the nodal points. This construction is thus restricted to flat drawing, used intensively in CAD software under the name of quadratic splines.



figures 341.1 to 341.4 : linear and quadric splines ; cubic splines, 6 and 16 points.

3) With a 4 point window (Qi,Qi+1,Qi+2,Qi+3) we define four points (in i/3 for i=[03]) as follows :

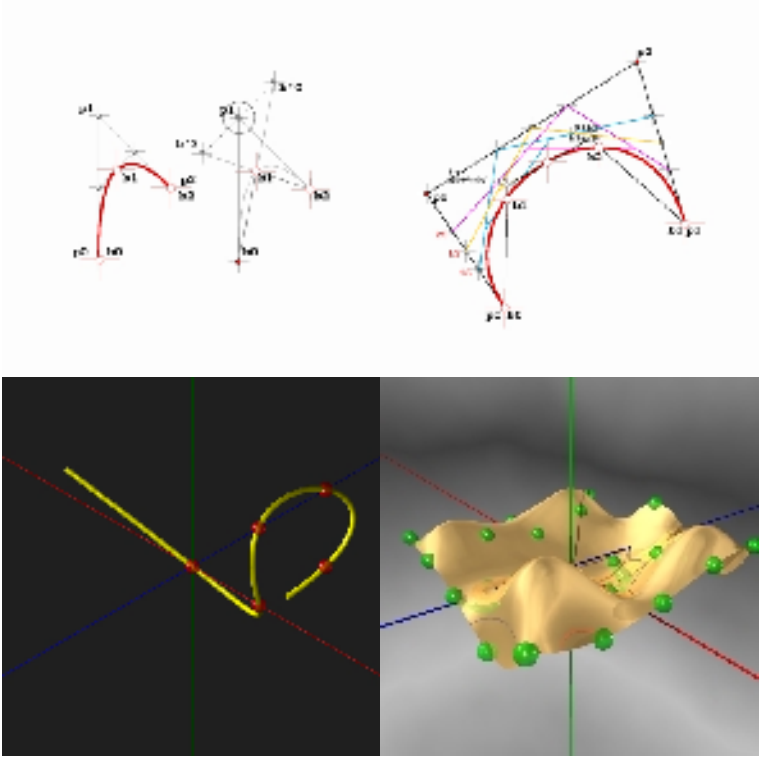
```
p0 = 1/6(Qi + 4.Qi+1 +   Qi+2        )
p1 = 1/6(     4.Qi+1 + 2.Qi+2        )
p2 = 1/6(     2.Qi+1 + 4.Qi+2        )
p3 = 1/6(       Qi+1 + 4.Qi+2 + Qi+3)
```

used as control points for a pL4 (cubic) by window. In constructing, the successive cubics connect up, with collinear tangents of the same module, and collinear osculatory planes that ensure torsion continuity, distributing it in fact along each curve. The latter property makes it a good tool for piloting complex curved curves in space, the trajectory of a camera, a robot arm, etc... and CAD software has turned it into the all-purpose tool.

Comment 1 : in fact, the simplified B_splines construction presented here correspond to what are referred to as uniform splines ; in the quadric case, for instance, we chose to position the nodal points at the midpoint of the control points, but it would be both possible and interesting to look at dissymmetrical and different positionings, generating what is called non-uniform splines, beta-splines, etc., abundantly dealt with in the literature on the subject.

Comment 2 : it's important to note that these curves do not interpolate the given points, except for the linear case (where we simply obtain the control polygon) ; we can contrive to create phantom points at the start and at the end to « draw » the curve toward the first and last points, we could « duplicate » certain intermediary points to give them more weight and attach the curve to them, but continuity could extend to provoking the emergence of angular points (which would moreover be an interesting property) ; we could also reconsider the problem and construct proper interpolating splines.

342 interpolating splines



figures 342.1 to 342.4 : construction of interpolating pL3 and pL4 ; a yellow pCurve interpolating 5 points, and a pSurface interpolating 25 points.

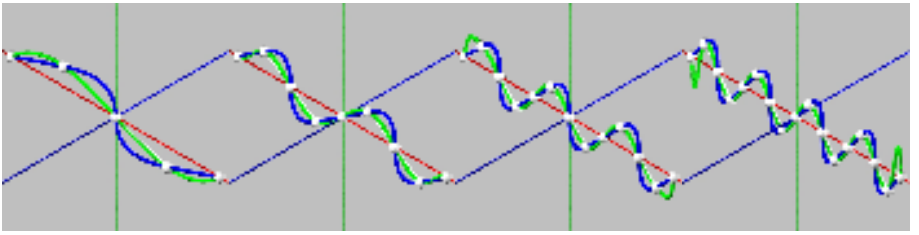
Construction of a concatenation of 1, 2 and 3 degree curves interpolating the given points presents no real problem in principle, but for our purposes the demonstration will be limited to presenting the quadric case. Let us take N **bi** points with i=[0,N-1] to interpolate using parabolas. By taking a point **b_1** defining **b0** as a vector tangent to the start of the curve, we can begin constructing the first parabola, noting that its second control point is point **b_1** symmetrical to **b0** :

```
p0 = b0
p1 = 2*b1 - b_1
p2 = b2
```

and so forth for the other parabolas, **b_1** being point **p1** of the preceding parabola.

Look how simple this operation is, no inversion of the linear system is required to find the solution, as was the case in section 221. The reader can tackle the cubic case as an exercise :-). The following programme produces a perfect circle using the quadric method (a parabola easily generates the 90° arc of a circle). It's interesting to see how, by displacing point **b_1** parallel to axis OZ, the curve stays on the cylinder of axis OZ and radius 1/2.

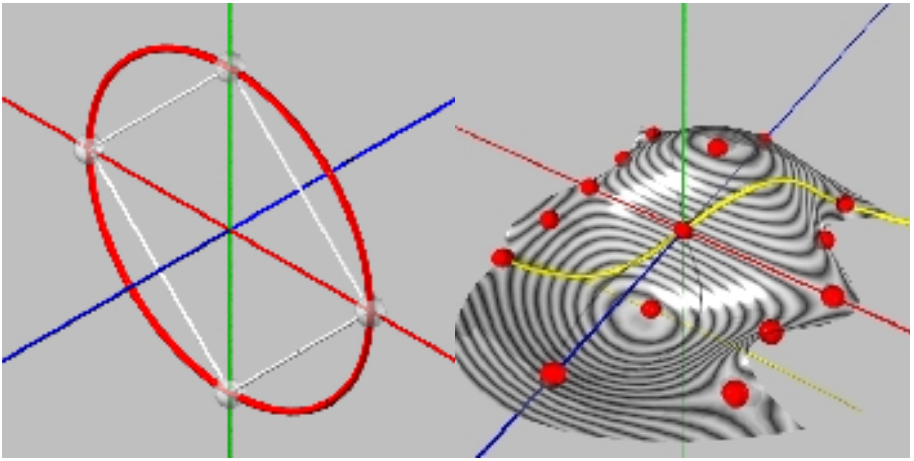
```
#macro spline_quadric( b0, b )
        #local k = sqrt(2)/2;   // -> circle
        #local n = _size( b );
        #local spline = array[n-1]
        #local q = array[3]
        #local q[1] = b0;
        #local i=0; #while (i< n-1)
                #local q[0] = bb[i];
                #local q[1] = 2*bb[i]-q[1];
                #local q[2] = bb[i+1];
                #local spline[i] = q
                #local spline[i][1] = spline[i][1]*k;
        #local i=i+1; #end
        spline
#end
#local b_1 = < 1/2,-1/2,0,1 >;  // initial point
draw_point( b_1, point( 0.05, < 1,0,0,0.7 > ) )
#local bb = array[5] { < 1/2,0,0,1 >, < 0,1/2,0,1 >, < -1/2,0,0,1 >, <
0,-1/2,0,1 >, < 1/2,0,0,1 > }
draw_courbe( bb, 0, point( 0.04, < 1,0,0 > ) )
#local circ = spline_quadric( b_1, bb )
#local i=0; #while (i< _size(circ))
        draw_courbe(circ[i],3,point(0.02,< 1,1,0 > ))
#local i=i+1; #end
```



figures 342.5 to 342.8 : blue interpolating quadric splines and green interpolating pCurves.

343 NURBS

So far we have only discussed concatenations based on an even distribution of points ; but it is also possible to envisage applying different (asymmetrical) behaviour at the straight line of nodal points, or multiplying points in the same place to construct non uniform splines, using the same reasoning. So we can construct interpolating or non interpolating splines that are or are not uniform. Remembering that the work is carried out in R4 space, we have thus produced NURBS, « Non Uniform Rational B-splines », keeping things simple, which is no mean feat ... This family of shapes considered a « must » in CAD, a kind of mathematical monster that is beyond direct understanding, has for a long time been rather mistreated by CAD softwares. Apart from software like Rhino which is based on them, NURBS are normally only used in constructing circles and surfaces of revolution. All we need to remember here is simply that NURBS can be seen as concatenations of pForms interpolating or non interpolating, uniform or non uniform, whose control points are obtained from base points through simple « sliding » linear transformations. The question that often arises as to whether or not Bézier curves are specific cases of NURBS is no more important basically than knowing whether a segment is a specific case of a circle or whether the atom comes before the molecule. In the pascalian approach, they're members of the same family !



figures 343.1 and 343.2 : a NURBS circle, quadric spline interpolating 4 points ; a NURBS surface interpolating 3 splines controlled by 5 points, in a slightly increased definition interval, and its yellow diagonal; the marble texture shows up the level lines.

35 deformation operators

In theory, any pForm can be manipulated at will to the smallest detail using the sub-forms that generate it, the deepest of which are the control points themselves. In practice, this is not always easy to do : how do you apply deformations like stretching/compression, bending, contorting or waving an object with any kind of coherence ? Imagine how hard it would be to position the control points, by hand, of the twenty-odd bicubics making up a teapot - to use the favourite example given by infographic programmers - into which boiling tea is poured until swelling point, a painful torsion resulting in plastic fusion that finally completely flattens it on the table... So an operator is needed that is capable of applying such deformations globally.

Two approaches at least can be envisaged : the first consists in applying a mathematical function (for example a sinusoid for waving) to all the points in the pForm, the drawback being that the resulting form is not a pForm, so we can no longer apply the aforementioned operators to it. The second consists in « embedding » the pForm in another pForm, the result of which is a pForm, as we know. Thus we can embed a pCurve in a pSurface (a pS32 for example) and displace the control points of the pSurface to deform the pCurve.

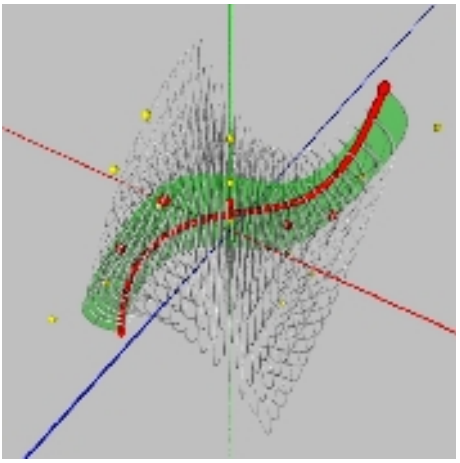


figure 35.1 : a flat facet and its diagonal immersed in a deformed cube (pV332).

A complex pVolume can also be immersed in another, simpler pVolume that is easier to deform. This provides a method for creating anamorphosis.

Figure 35.1 shows the deformation of a flat facet and its diagonal immersed in a cube (pV332) constructed on two biquadrics (pS33); we have 3x3x2 = 18 possible control points to deform this facet and its diagonal, and any other immersed pForm (curve, surface, volume).

Figure 35.2 shows another example of deformation : a portion of a torus in granite (pS33) and its white diagonal (pL5) are immersed in a cube (pV223) controlled by 3 facets (pS22) and represented by a grid of 12 red points ; the median facet is sinusoidally displaced horizontally, and the torus portion and its diagonal are carried along in the deformation of the cube.

Comment 1 : One important point to remember is the fact that before being curved, a pForm can be perfectly « straight »! A pS22 can totally represent a perfect classic square and a pV222 can represent a simple orthonormed cube. The particularity of pforms is that no metrics are presupposed in their definition, which is why there's no problem carrying them off into curved space.

Comment 2 : another point we shouldn't forget is that a complex form in our space can be considered as a slightly simpler form in more complex space ; the white diagonal of the torus portion carried off in the deformed cube in figure 35.2 is a simple segment immersed in the torus portion, the torus portion is a truly classic torus with constant radius carried off in the cube. Question : what, in our space, is the number of control points of the pCurve diagonal of the deformed torus ?

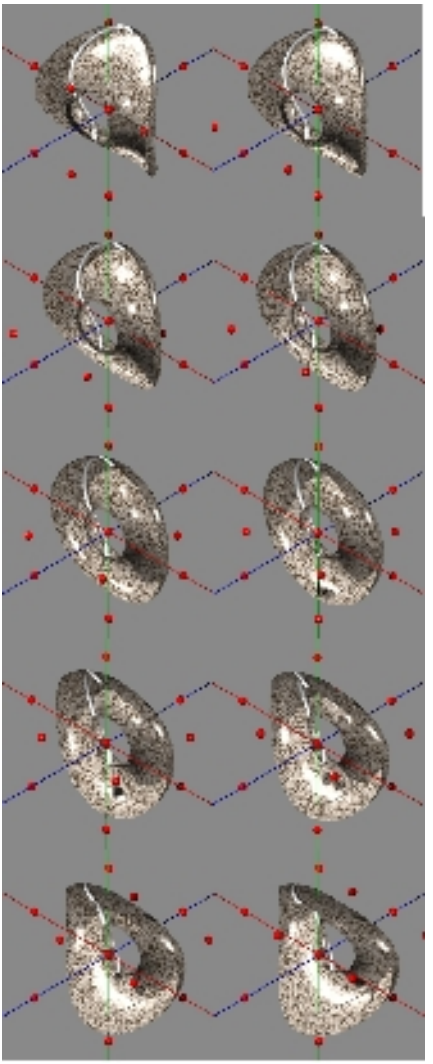
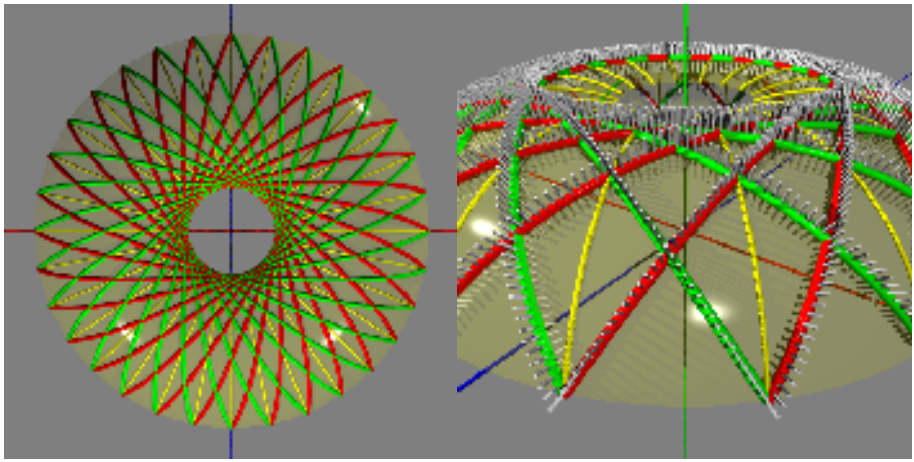


figure 35.2 : a torus (pS33) and its diagonal (pL5) immersed in a cube (pV322) deformed by horizontal sinusoidal translation of its median control plane.

36 geometry in pForms

We have already seen that, from the concept of a segment immersed in a pSurface, we could stretch the concepts of parallelism, orthogonality, and angle, study the intersection of two eSegments, its divisions, prolongations, etc..., thus defining a geometry in pSurfaces. We are not limited to segments immersed in pSurfaces, and the reasoning is valid for all pForms, starting with pVolumes. The present essay constitutes nothing but a first approach to this, beyond which, all the necessary precision and coherence is required to construct a non metric pGeometry of curved forms, and such is the task ahead of us...

While the segments immersed in pSurfaces are not usually geodesics, it is sometimes necessary to calculate geodesics on pSurfaces, to deal with « metric » in pSurfaces. This was done in participation in the design of a roof for the circular swimming pool in St Quentin in Yvelines for the firm of engineers of Michael Flach, specialised in wooden structures. The structural principle is particularly elegant : planks aligned along geodesic lines are stacked alternatively to form arcs of great inertia perpendicular to the surface ; crossing them naturally forms knots, the whole becoming a mesh onto which the roof plates are laid.



figures 36.1 and 36.2 : study, crossing geodesics on a portion of torus.

The problem with this technique « is » to apply a thin plank to a curved surface - not always an easy task : in the case of a hyperboloid of revolution, the two families of rectilinear generatrixes can be followed, in the case of a sphere we would follow the arcs of the big circle (which is the case for meridians but not for parallels), in the case of a torus and even more so, of a more complex form, there is only one solution : following the geodesic lines. So it is the differential system presented in section 0232 that has to be used, in a classic iterative approach resulting in an array of points defined as [x,y,z]. For the

implementation carried out in POVRAY/pFlibs, you write :

```
#local geo = geodesic( surf, P, A, N, dt )
where P=starting point, A=firing angle, N= nb points, dt= pitch
```

figures 36.3 and 36.4 : construction, toroidal roof composed of geodesic arcs in nailed planks allowing easy crossing at the nodes.

37 other operations on pForms

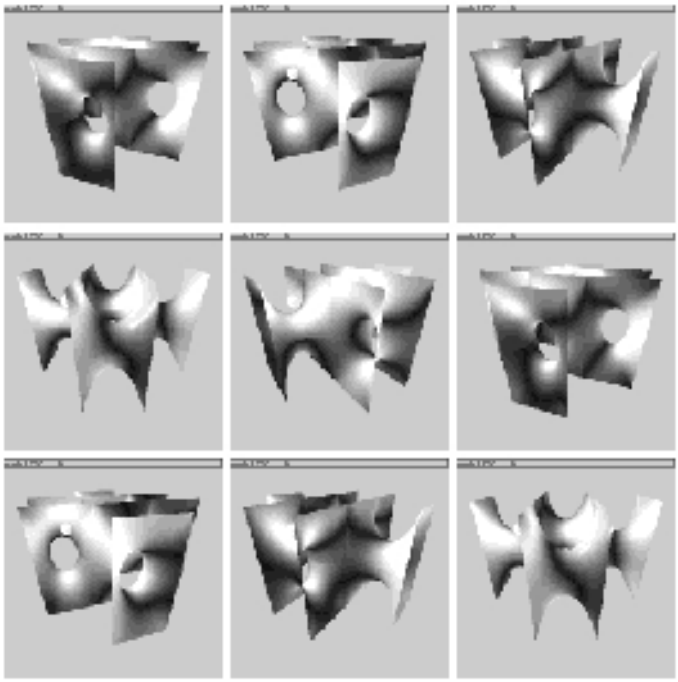


figure 37: periodic minimal surface.

Other forms could be studied such as the products of pForm operators, boolean operators on pForms (intersection, difference, union), and why not, quasi-minimal surfaces that satisfy Laplace's equation, whose expression of current point finite differences :

```
P[i,j] = ( P[i-1,j] + P[i+1,j] + P[i,j-1] + P[i,j+1] ) / 4
```

recalls the definition of a midpoint of a curved facet. A well-chosen pSurface could perhaps, with its ipSegments and other ipCurves, prove to be a good guide in the study of quasi-minimal surfaces. Figure 37 was calculated using to a standard method (iterative resolution of Laplace's equation) ; to be compared with Figure 332.5 using the Coons method to produce a pSurface.

Comment 1 : operators MI() and MIR() are central to generating pascalian forms, and it is possible to imagine fruitful variants. The Cantor set (1883) is a fractal set with points defined as attracting the family of two homeotheses of factor 1/3 and centres 0 and 1. The 1/3 factor is not chosen randomly: if we

chose 1/2 we would obtain a good pCurve (pL2) which is not at all fractal... One idea would be a THIRD() operator rendering two points to 1/3 and to 2/3, linked with a THIRD_R() operator to generate some very interesting fractal pForms. Another idea would be to find a definition generating generalised Peano pForms, for instance pForms with a topological dimension 1 and fractal dimension 2, diagonals of who knows what pMonster. And why not a RAND() operator rendering a random point between two points, very useful in fact, to generate an equal distribution (Gaussian) throughout the form ?

Comment 2 : treatises on geometry have a duty to study surfaces in their widest generality, able to be expressed easily, in continuous functions that are N times differentiable. They contain a point by point analysis of linear properties, what occurs in the tangent plane, and the second order properties, curvature, torsion, what occurs in osculatory quadrics (elliptical, parabolic, hyperbolic), and they usually stop there, as a kind of Taylor development series of surfaces limited to the second order. The concept of metrics has to be mentioned in passing, and one can generalise in a very abstract way and speak of Riemannian varieties, but beyond that lies a great void with no signposts, for ordinary mortals at least ! By limiting the study of surfaces to the pSurface family, things get a lot easier : no more development in series a priori, (pSurfaces are polynomials of finite degree) and beyond the reference surfaces of the order of 2, like the ellipsoid, paraboloid, hyperboloid, we find some interesting beings that are easy to manipulate (outside any metrics) starting with the biquadric pS33 and its diagonal pL5. The pS33 is the first real double curvature surface that is fairly malleable with its 9 control points to serve as a model for numerous surfaces like the sphere and the torus. As for the pL3, we have seen from this essay how important it is, both in representing the complete circle and because it happens to be the immersed segment of the pS33, i.e., the key component of its whole geometry.

But all these avenues were explored long ago by generations of talented mathematicians who staked them out in cogent theories, so there is nothing new here ; my only hope is that this different if not original rereading, based on elementary gestures will enable, as it did for me, non mathematicians to see things more clearly, gain a better grasp of curved forms and open the door to other, more exotic geometries.

conclusion

Beneath these vague pipe forms lie simple statements, beneath the complex forms of the Sagrada Familia lies the art of forms, mastered and transmissible, creative art to be shared !

It is important to remember that without needing to go back to complex analytical formulation, the two fundamental operators MI() and MIR() have allowed us to generate a family of forms, the pForms ; we have studied the relationships between pForms immersed in other pForms (ipForms) and pForms in space, we have imagined them in four-dimensional space and projected conically in our own to produce rational forms, concatenated to produce splines and combined as affine sums and products to produce more complex forms, finding most of them in the well known forms of geometry.

Pascalian forms and their accompanying operators thus constitute a tool to design and manipulate an important family of geometrical shapes. Let us finally note the following points :

1. **Simplicity of approach**
   All the avenues revisited in this approach have long been explored by talented mathematicians who staked them out in cogent theories, and so there is nothing very new here ; but the hope is that this rereading based on a unitary approach and elementary gestures will be of as much help to someone having to draw curved forms freehand, as to the user of today's powerful computer tools, to see a little more clearly, to better master the hand and the machine, and even to explore and discover new forms. This is precisely one of the prime advantages of a simple approach, opening the door to new combinations and embeddings leading to geometries whose classic study is often a mathematical challenge, or at least requires a level that is beyond the reach of ordinary mortals. I'm thinking here of projective geometry, Riemann's sphere, Poincarré's circles, attempts to represent post relativist cosmological theories. Geometry (according to

Klein) is the study of invariant figures/properties in a given transformation, thus leading us to envisage different geometries that fit into one another, euclidian, affine, projective, geometries,... by successively removing the « constraints », metrics, angles, points to infinity, the removal of each constraint freeing geometry to move toward greater abstraction and richer content. In what « corner » of this progression could we find a geometry based on pForms ?

2. **Just one tool, a cord**
   Let us remember that the initial gesture defined in operator MI() is the construction of a midpoint, and such a construction can be done « by hand » in - very straight- euclidian space with a simple cord that was first stretched between two posts, then folded back on itself to mark the midpoint. Under certain conditions seen in this essay, this gesture can still be used for pSurfaces : you can in fact break them down more or less accurately into a « patchwork » of pSurfaces, drawing « by hand » with (in principle) a simple cord, immersed pCurves, pCurves assembled in a car body, pLines in the tailoring of a garment, pCurves to assemble stretched surfaces before or after optimisation, etc...

3. **Anamorphosis**
   This approach can also be used in the case of anamorphosis ; the deformation of a square surface leaving the four sides rectilinear and coplanary will generate a facet pS22 ; an immersed segment will become a pL3 (parabola), a parabola will become a pL5, the arc of a circle will become the conical projection of a parabola etc... We can also envisage the coherent deformation of a set of pForms by embedding them in a curved cube V222, V333, or again, in torsion, folding, stretching operations, etc... Whatever the case me be, we have at our disposal a vectorial anamorphosis tool to explore and to exploit.

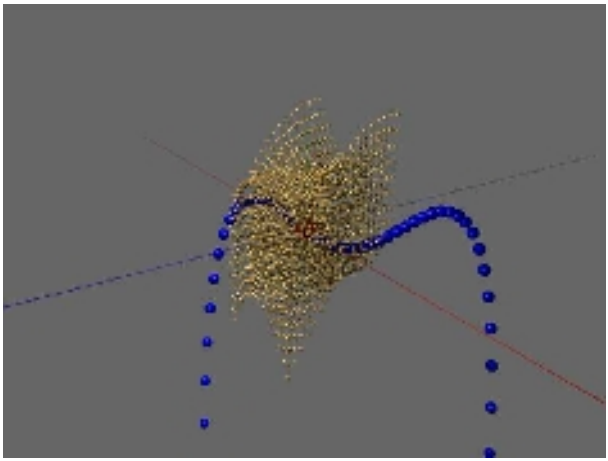
4. **Rules of composition**
   This approach provides a glimpse of a system of rules that can be applied to the « harmonious composition » of pForms, extending the well established rules of rectilinear geometrical forms ; the « harmonious » disposition of the control points of a pForm would thus ensure the harmonious character of the form, and three points situated at the apexes of a golden rectangle (1,618) would be assumed to generate a « harmonious » parabola. This could provide a good analytical tool for curved architecture : it would be interesting to look into the network of control points for the Sagrada Famillia, the Guggenheim Museum in Bilbao, Laetitia Casta's hips or Ford's KA ?

5. **3D Spreadsheet**
   The spreadsheet concept is ideally simple : a table of cells to enter data and visualise the results and the possibility of establishing all kinds of relationships between the cells. By entering the number « 2 » in cell A1, number « 3 » in cell A2 and the formula « = A1 + A2 » in cell A3, we obtain « 5 » in this cell. Now imagine that by entering three « pL3 » respectively, in cells A1, A2 and A3, writing in cell A4 the formula « = MIR(A1, A2, A3) » and in cell A5 the formula « =DIAG(A4) », and calling up the graphic representation of these two cells, having carefully chosen the 9 control points of the three initial pL3 we could, for instance, see a portion of a sphere and its diagonal as a portion of Viviani's window. Pascalian forms armed with their internal operators could thus fill a kind of 3D spreadsheet, ... which still remains to be written.
   .

6. **Computer implementation**
   Lastly, the gestural approach developed (cord, posts, midpoint, ...) turns out to be easily translated into terms that are immediately understood by the computer ; a computer has more trouble processing high degree algebraic expressions than applying the recursive application of dividing by two, and this is about all it is asked to do in this approach. Implementation in POVRAY language (that can be downloaded from the site : http://www.povray.org) has resulted in a complete software tool (POVRAY+pFlibs.inc) allowing the construction of all pForms (presented in this work and in others) with a means of « visually » controlling their validity. Further information about this implementation can be found on the following page : http://amartyfree.free.fr/alain.marty/index.php?page=pformes .

Leafing through two recent works on projective geometry and on NURBS, the former flying high in Bourbaki type mathematical concepts, and the latter full of heavy indexed expressions of increasing complexity, so virtually inaccessible as far as I am concerned, I was reminded of the long hours spent at university studying Laplace's equation in every possible system of coordinates, and how little I actually retained. Then I thought about this manipulation of Laplace's formula on a simple spreadsheet (cf 0233 sliver of soap) presented to me during an Apple symposium « Think different! », and the effect that discovery had on me : I had just understood Laplace's equation, the obvious simplicity of its local behaviour (a simple arithmetical mean, zero acceleration) and the fundamental importance of the conditions at the limits that alone model the essentials of the resulting form. So I thought that if, in spite of its faults, the approach proposed in this essay on pascalian forms could have the same effect on the reader, I'd be a happy man.

Alain Marty, July 2004, Villeneuve de la Raho, marty.alain@free.fr

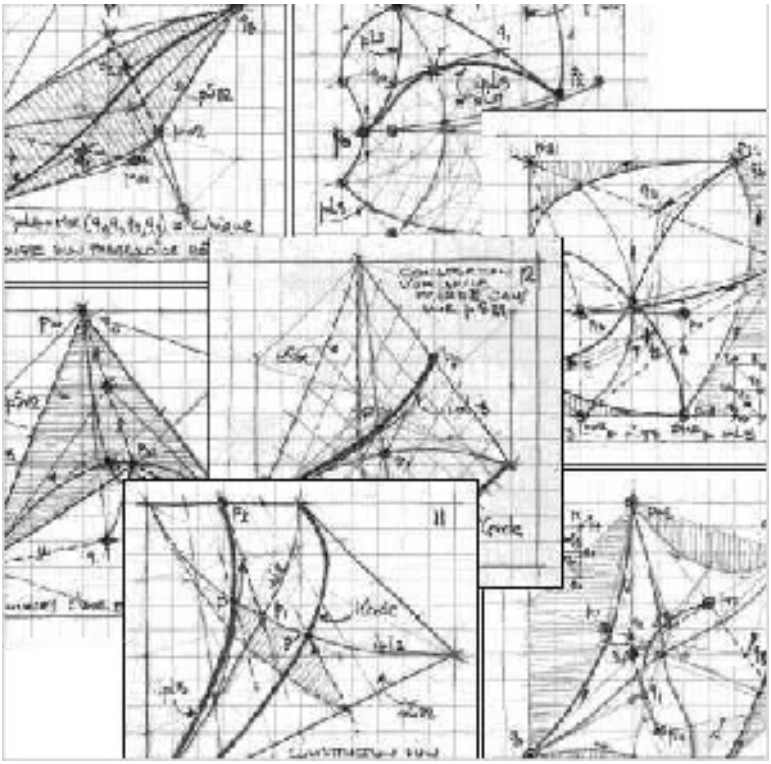


A deformed curved cube (pV332) represented in the interval u,v,w = [0,1] and its diagonal stretched in the interval [-infinite,+infinite] ; note that the effect of deforming a curved cube does not stop at the boundaries of its representation, it concerns the whole of space.

references

There is abundant literature on the subject. The short list below refers only to the works that really accompanied this study, serving as a true guide and model.

- 1] GAMOV G (1936), Mr Tompkins in Wonderland, republished by Dunod in 1965: a real story of love and relativist and quantic physics.
- [2] DE CASTELJAU, Paul (1959), Courbes et surfaces à pôles, Citroën, Paris: unpublished at the time, because of Citroën's industrial secrecy; Pierre Bézier had more luck with Renault, and his name remains for pole curves and surfaces.
- [3] ROGERS DF ADAMS JA Mc Graw Hill edition (1990) Mathematical Elements for Computer Graphics: a summary of computer graphics.
- [4 UPSTILL S Addison-Wesley edition (1990) The Renderman Companion: a high quality software tool used by the firm PIXAR for its animation films.
- [5] SNYDER JM Academic Press edition (1992) Generative Modeling for Computer Graphics & CAD: a well-constructed unitary approach to geometrical forms.
- [6] FARIN G Masson edition (1992) Courbes et surfaces pour la CGAO: the basis for really understanding Bézier curves and what follows on.
- [7] BERNARD WERBER Edition de poche (1994) Les Fourmis (Ants): think different!
- [8] MICHIO KAKU Oxford University Press edition (1994) Hyperspace, Through The 10th Dimension: more and more dimensions for an increasingly simple universe.
- [9] LES PIEGL WAYNE TILLER Springer edition (1996) The NURBS Book: a summary of NURBS.
- [10] FARIN G AK Peters Ltd edition (1999) NURBS, from Projective Geometry to Practical Use: certainly the way to go, but so hard to understand; and what if pForms were a simpler expression of this?
- [11] POVRAY, (1999) POV-Help, POV-Team, accompanying the Open Source software, available at: http://www.povray.org ;
- ... and some more useful books whose references will be found in those listed before, books on pure and applied mathematics, on classical geometry and differential geometry, on tensor calculus, on riemanian varieties, on relativity, mecanics and elasticity, on computer systems, ..., this important set of tools you are brought to use in such kind of exploration ...



Basic pForms can be drawn by hand on squared paper.

implementation

All the images of pascalian forms used were produced in the POVRAY open source environment (http://www.povray.org). The fundamental macros as well as those at the basis of the images rendered that are incorporated in this work were grouped into two files, pFlibs.inc and pFbook.inc.

A typical example is presented below with the full listing of the two files that follow on.

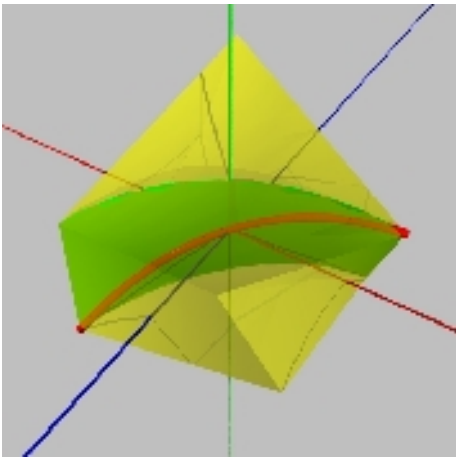
This listing can be downloaded from this site :

http://amartyfree.free.fr/alain.marty/index.php?page=pformes

avoiding a lot of keyboard entering with the inevitable typing errors ; pending a manual/tutorial (that still needs to be written), reading the interface macro statements (draw(), pFdiagonalisation(), etc...) could provide some necessary indications on their use.

typical example

Typical example of a program using the POVRAY/pFlibs libraries and producing the figure below :



Transparent yellow curved cube (pV222), its green diagonal surface (pS23) and the red diagonal of this surface (pL4)

```
// 1) POVRAY alone does not recognize pForms, so
// it is necessary to include the file pFlibs.inc:
#include "pFlibs.inc"

//////////////////////////////////////////////////////////////

// 2) CONSTANT SETTINGS:
// background colour of the image:
background { color rgb 3/4 }     // grey 75%
// unit cube transparent centered on the origin:
// box { -1/2, 1/2 pigment { color rgbt 0.9 } scale 1.0001}
// three light sources:
light_source { <  2,2,-2 >   color rgb 1 }
light_source { < -2,2,-2 >   color rgb 1 }
light_source { <  2,0, 2 >   color rgb 1 }
// camera, either axonometric, or perspective:
// view_axonometric( AXO, 7/8 )
view_perspective( < 0.7,1.3,-1.1 >*6/6 )

//////////////////////////////////////////////////////////////

// 3) SCENE :

// a) construction of pForms :
#local pL2_0 = array[2] {<-0.5,-0.5,-0.5,1.0>,<0.5,0.0,-0.5,1.0>}
#local pL2_1 = array[2] {<-0.5, 0.0, 0.5,1.0>,<0.5,-0.5,0.5,1.0>}
#local pS22  = array[2] { pL2_0, pL2_1 }
#local pS22_0 = pS22
#local pS22_1 = pS22
pFtranslate( 2, pS22_1, <0, 0.5,0> )
pFrotate( 2, pS22_1, <0,-15,0> )
#local pV222 = array[2] { pS22_0, pS22_1 }
#local pS23 = pFdiagonalisation( 3, pV222 )
#local pL4 = pFdiagonalisation( 2, pS23 )

// b) drawing of pForms :
draw( 3, pV222, finesse(<1,3,3,0>)
      + enveloppe(LISSE) + ma_couleur(<1,1,0,0.5>) )
draw( 2, pS23, finesse(<4,4>)
      + surface(LISSE) + ma_couleur(<0,1,0,0>) )
draw( 1, pL4, finesse(5)
      + courbe( 0.02 ) + ma_couleur(<1,0,0,0>) )
```

pFlibs.inc file

The macros of the fundamental pFlibs.inc file are listed in the following pages.

```
// file « pFlibs.inc »
// under GPL open source licence
// http://marty.alain.free.fr
// version of 01/06/2004

/*----------- MACROS GENERALES DANS R4 ------------*/

/*
    TRANSFORMATION GENERALE AFFINE:
        pp.x      |  00 01 02 03 |    |  p.x
        pp.y   =  |  10 11 12 13 |  * |  p.y
        pp.z      |  20 21 22 23 |    |  p.z
        pp.t      |  30 31 32 33 |    |  p.t
*/
#macro pFtransform( mat, p )
< mat[0][0]*p.x + mat[0][1]*p.y + mat[0][2]*p.z + mat[0][3]*p.t,
  mat[1][0]*p.x + mat[1][1]*p.y + mat[1][2]*p.z + mat[1][3]*p.t,
  mat[2][0]*p.x + mat[2][1]*p.y + mat[2][2]*p.z + mat[2][3]*p.t,
  mat[3][0]*p.x + mat[3][1]*p.y + mat[3][2]*p.z + mat[3][3]*p.t >
#end

#macro pFtranslate( dim, f, dt ) // translation d'une pForme
    #if (dim=0)
        #local tt = f.t;  // translation dans R3 < dt.x,dt.y,dt.z >
        #local pp = < f.x/tt, f.y/tt, f.z/tt > + dt;
        #local f = < pp.x*tt, pp.y*tt, pp.z*tt, tt >;
    #else
        #local n = taille( f );
        #local i=0; #while (i< n)
            pFtranslate( dim-1, f[i], dt )
        #local i=i+1; #end
    #end
#end

#macro pFrotate( dim, f, dr )    // rotation d'une pForme
    #if (dim=0)
        #local tt = f.t;  //  rotation dans R3 < dr.x,dt.y,dt.z >
        #local pp = < f.x/tt, f.y/tt, f.z/tt >  ;
        #local rad = pi/180;
        #if (dr.x != 0)
```

```
            #local pp = <
            pp.x,
            cos(rad*dr.x)*pp.y - sin(rad*dr.x)*pp.z,
            sin(rad*dr.x)*pp.y + cos(rad*dr.x)*pp.z >;
        #end
        #if (dr.y != 0)
            #local pp = <
            cos(rad*dr.y)*pp.x -sin(rad*dr.y)*pp.z,
            pp.y,
            sin(rad*dr.y)*pp.x + cos(rad*dr.y)*pp.z >;
        #end
        #if (dr.z != 0)
            #local pp = <
            cos(rad*dr.z)*pp.x - sin(rad*dr.z)*pp.y,
            sin(rad*dr.z)*pp.x + cos(rad*dr.z)*pp.y,
            pp.z >  ;
        #end
        #local f = < pp.x*tt, pp.y*tt, pp.z*tt, tt >;
    #else
        #local n = taille( f );
        #local i=0; #while (i< n)
            pFrotate( dim-1, f[i], dr )
        #local i=i+1; #end
    #end
#end

#macro pFscale( dim, f, ds )    // homothetie d'une pForme
    #if (dim=0)
        #local tt = f.t;  //  homothetie dans R3 < ds.x,ds.y,ds.z >
        #local pp = < f.x/tt*ds.x, f.y/tt*ds.y, f.z/tt*ds.z > ;
        #local f = < pp.x*tt, pp.y*tt, pp.z*tt, tt >  ;
    #else
        #local n = taille( f );
        #local i=0; #while (i< n)
            pFscale( dim-1, f[i], ds )
        #local i=i+1; #end
    #end
#end

/*
    MACROS INTERNES
*/

#macro _interpol( dim, f0, f1, uu )    // interpole deux formes
    #if (dim=0)
```

```
        (1-uu)*f0 + uu*f1;
    #else
        #local nb = taille( f0 );
        #local fm = array[nb]
        #local i=0; #while (i< nb)
        #local fm[i] = _interpol( dim-1, f0[i], f1[i], uu )
        #local i=i+1; #end
        fm
    #end
#end

#macro _shift( dim, n, sens, uu )
    // calcule la sous_forme droite ou gauche
    #if (n >  0)
        #local i=0; #while (i< n)
            #if (sens)            // droite
             #declare _d[i] = _interpol( dim-1, _d[i], _d[i+1], uu )
            #else                 // gauche
             #local nb = taille( _g );
             #local k = nb-1-i;
             #declare _g[k] = _interpol( dim-1, _g[k], _g[k-1], uu )
            #end
        #local i=i+1; #end
        _shift( dim, n-1, sens, uu )
    #end
#end

#macro _stretch( dim, f, p0, p1 )  // cale f entre p0.x et p1.x
    #local _g = f
    _shift( dim, taille(f)-1, false, 1-p1.x ) // travail sur g
    #local _d = _g
    _shift( dim, taille(f)-1, true, p0.x/p1.x )// travail sur d
    _d                              // mêmes dims que f
#end

#macro _sous_subdi( dim, p, r ) // charge out et incrémente index
    #if (dim=0)
        #declare out[index] = p;
    #else
        #local pp = pFsubdivision( dim,p,< r.y, r.z, r.t, 0 > )
        #declare out[index] = pp
    #end
    #declare index=index+1;
#end
```

```
#macro _subdi( dim, f, r, uu ) // corps récursif de subdivision()
    #if (r.x=0)
        #local i=0; #while (i< nb_in-1)
            _sous_subdi( dim-1, f[i], r )
        #local i=i+1; #end
    #else
        #local _g = f
        #local _d = f
        _shift(dim, nb_in-1,true,uu)
        _shift(dim, nb_in-1,false,uu)
        _subdi( dim, _g, < r.x-1, r.y, r.z, r.t >  , uu )
        _subdi( dim, _d, < r.x-1, r.y, r.z, r.t >  , uu )
    #end
#end

/*
    3 MACROS FONDAMENTALES
*/

#macro pFsubdivision( dim, in, r )
    // retourne la forme subdivisee
    #local nb_in = taille(in);        // récursivement au 1/2
    #local nb_out = (nb_in-1)*pow(2, r.x)+1;    //
    #local out = array[nb_out]
    #local index = 0;
    _subdi(dim, in, r, 1/2)      //_subdi(dim,in,r,1/3) sierpinsky
    _sous_subdi(dim-1, in[nb_in-1], r)
    out
#end

#macro pFstretch( dim, f, p0, p1 )
    #if (dim=1)
        _stretch( 1, f, p0, p1 )
    #else
        #local ff = _stretch( dim, f, p0, p1 )
        // ff prend f recalé entre p0.x et p1.x
        #local n = taille(ff);    // nb de sous_formes
        #local i=0; #while (i< n) // pour chaque sous_forme:
            #local ff[i] = pFstretch( dim-1, ff[i],
                    < p0.y,p0.z,0,1 >  , < p1.y,p1.z,0,1 >   )
        #local i=i+1; #end
        ff    // retourne ff recalée
    #end
#end
```

```
#macro pFgetSubForm( dim, f, uu )  // retourne la sous_forme
    #local _d = f    // generatrice en uu (uu est un REEL !!)
    _shift(dim, taille(f)-1, true, uu)
    _d[0]    // tableau unidim
#end

/*
    IMMERSION
1) get_point() retourne les coordonnées globales d'un point défini en
coordonnées locales (p) dans la pForme f
2) immersion() transforme une ipForme définie en coordonnées locales
dans une pForme, en un tableau de points définis en coordonnées
globales. Ce tableau ne définit pas une pForme (une post-subdivision
n'est pas possible)
3) voir plus loin les macros interpolation plus générales retournant
les points de contrôle dans le cas de pCourbes dans une pSurface
*/

#macro pFgetPoint( dim, f, p )
    #local qt = p.t;
    #local q = < p.x/qt, p.y/qt, p.z/qt,1 >  ;
    #switch (dim)
        #case (1)
            #local pp = pFgetSubForm( 1, f, q.x );
        #break
        #case (2)
            #local ff = pFgetSubForm( 2, f, q.y )
            #local pp = pFgetSubForm( 1, ff, q.x );
        #break
        #case (3)
            #local gg = pFgetSubForm( 3, f, q.z )
            #local ff = pFgetSubForm( 2, gg, q.y )
            #local pp = pFgetSubForm( 1, ff, q.x );
        #break
        #case (4)
            #local hh = pFgetSubForm( 4, f, q.t )
            #local gg = pFgetSubForm( 3, hh, q.z )
            #local ff = pFgetSubForm( 2, gg, q.y )
            #local pp = pFgetSubForm( 1, ff, q.x );
        #break
        #else
            #render "...nothing out there !!! "
    #end
    < pp.x*qt, pp.y*qt, pp.z*qt, pp.t*qt >  ;
#end
```

```
#macro pFimmersion( dim, f, idim, imf )
    // ATTENTION: imf est modifiée
    #if (idim=0)        // et n'est plus une pForme
        #local imf = pFgetPoint( dim, f, imf )
        // NO post-subdivision
    #else
        #local n = taille( imf );
        #local i=0; #while (i< n)
            pFimmersion( dim, f, idim-1, imf[i] )
        #local i=i+1; #end
    #end
#end

/*
    DIAGONALISATION D'UNE FORME
*/

#macro _diag_N2( M, s0, s1 )
    // diagonale d'une surface de type [N,2]
    #local pp = array[M+1]
    #local pp[0] = s0[0];
    #local pp[M] = s1[M-1];
    #local i=1;     #while (i< M)
        #local uu = i/M;
        #local pp[i] = (1-uu)*s0[i] + uu*s1[i-1];
    #local i=i+1;   #end
    pp
#end

#macro _diag_surf( surf )
    // diagonale d'une surface de type [M,N]
    #local M = taille( surf[0] );
    #local N = taille( surf );
    #if (N >  2)
        #local ss = array[N-1]
        #local j=0;         #while (j< N-1)
        #local ss[j] = _diag_N2( M, surf[j], surf[j+1] )
        #local j=j+1;    #end
        _diag_surf( ss )
    #else
        _diag_N2( M, surf[0], surf[1] )
    #end
#end
```

```
#macro pFdiagonalisation( dim, f )
    // diagonale d'une pForme quelconque
    #if (dim=2)
        _diag_surf( f )
    #else
        #local M = taille( f );    // nb de sous-formes
        #local ddd = array[M]
        #local i=0;    #while (i< M)
            #local ddd[i] = pFdiagonalisation(dim-1,f[i])
        #local i=i+1;    #end
        ddd
    #end
#end

/*
    ELEVATION DU DEGRE D'UNE FORME
*/

#macro pLup( f, dd ) // en attendant une version up_forme qq...
    #local ff = f
    #local i=0; #while (i< dd)
        #local N = taille(ff);
        #local fff = array[N+1]
        #local fff[0] = ff[0];
        #local fff[N] = ff[N-1];
        #local j=1; #while (j< N)
            #local fff[j] = j/N*ff[j-1] + (1-j/N)*ff[j];
        #local j=j+1;    #end
        #local ff = fff
    #local i=i+1;    #end
    ff
#end

#macro pSup( f, dd ) // en attendant une version up_forme qq...
    #local M = taille( f );      // nb de courbes
    #local i=0; #while (i< M)    // pour chaque courbe:
        #local f[i] = pLup( f[i], dd.u )    // élever degré
    #local i=i+1; #end
    #local N = taille( f[0] );   // nb de points d'une courbe
    #local ff = array[N]   // création d'une surface ortho
    #local i=0; #while (i< N)    // pour chaque point:
        #local M = taille( f );  // nb de courbes
        #local fff = array[M]    // création d'une courbe ortho
        #local j=0; #while (j< M)
            #local fff[j] = f[j][i];
```

```
        #local j=j+1; #end
        #local ff[i] = pLup( fff, dd.v )    // élever degré
    #local i=i+1; #end
    ff
#end

/*
    AFFICHAGE D'UNE FORME DANS LE CAS GENERAL
        cas des points, courbes, surfaces et volumes
        traité plus loin sous différentes formes
*/

#macro pFdraw( dim, f, rayon )
    // dessine les points de la pForme dans R3
    #if (dim=0)
        sphere { < f.x/f.t, f.y/f.t, f.z/f.t  >  , rayon }
    #else
        #local n = taille( f );
        #local i=0; #while (i< n)
            pFdraw( dim-1, f[i], rayon )
        #local i=i+1; #end
    #end
#end

/*------------- COURBES INTERPOLANTES ET
            COURBES IMMERGEES DANS UNE SURFACE ------------*/

//    MATRICE DE POINTS UNIFORMEMENT DISTRIBUES SUR UNE pCOURBE

#macro fact( n )
    #if (n=0)
        1;
    #else
        n*fact(n-1)
    #end
#end

#macro calcul_mat( n )                  // matrice de n points
       //#local n = n-1;                // uniformement distribues
       #local mat = array[n+1][n+1]     // sur une pLn
       #local fn = fact( n )
       #local CC = array[n+1]
       #local j=0; #while (j<n+1)       // precalul coeffs binomiaux
             #local fnj = fact( n-j )
             #local fj  = fact( j )
```

```
            #local CC[j] = fn/(fnj*fj);
      #local j=j+1; #end
      #local EPS = 0.0000001;   // problem with zero in MegaPov
      #local i=0; #while (i<=n)        // pour chaque ligne
            #local j=0; #while (j<=n) // pour chaque colonne
                  #local temp = CC[j];
                  #local temp = temp*pow(EPS+1-i/n,n-j);
                  #local temp = temp*pow(EPS+i/n,j);
                  #local mat[i][j] = temp;
            #local j=j+1; #end
      #local i=i+1; #end
      mat
#end

#macro ctrliCurvInSurf( surf, curv )
    // nb de points de controle
    #local nb1 = taille( surf );       // de la courbe de R4
    #local nb2 = taille( surf[0] );    // correspondant à la courbe
    #local nb = taille( curv );        // immergée dans une surface
    ((nb1+nb2-2)*(nb-1)+1)
#end

/*
    pCOURBE et pSURFACE INTERPOLANTES
    en attente d'une formule générale pour les pFormes
*/

#macro pLinterpolante( curv )
    #local n = taille( curv );
    #local vx = array[n]
    #local vy = array[n]
    #local vz = array[n]
    #local vt = array[n]
    #local i=0; #while (i< n)
        #local vx[i] = curv[i].x;
        #local vy[i] = curv[i].y;
        #local vz[i] = curv[i].z;
        #local vt[i] = curv[i].t;
    #local i=i+1; #end
    #local mat = calcul_mat( n-1 )
    #local tam = inverse_mat( mat )
    #local _vx = produit_mat( tam, vx )
    #local _vy = produit_mat( tam, vy )
    #local _vz = produit_mat( tam, vz )
    #local _vt = produit_mat( tam, vt )
```

```
    #local icurv = array[n]
    #local i=0; #while (i< n)
        #local icurv[i] = < _vx[i], _vy[i], _vz[i], _vt[i] >;
    #local i=i+1; #end
    icurv
#end

#macro pSinterpolante( surf )
    #local m = taille(surf);
    #local n = taille(surf[0]);
    #local ipLm = array[m]
    #local i=0; #while (i< m)
        #local ipLm[i] = pLinterpolante( surf[i] )
    #local i=i+1; #end
    #local temp = array[m]
    #local isurf = array[n]
    #local i = 0; #while (i< n)
        #local j=0; #while (j< m)
            #local temp[j] = ipLm[j][i];
        #local j=j+1; #end
        #local isurf[i] = pLinterpolante( temp )
    #local i=i+1; #end
    isurf
#end

//    pCOURBE IMMERGEE DANS UNE pSURFACE

#macro courbe_in_surface( courbe_immergee, surf )
    #local nb = ctrliCurvInSurf( surf, courbe_immergee );
    #local b = array[nb]
    #local i=0; #while (i< nb)
    #local pp = pFgetSubForm( 1,courbe_immergee,i/(nb-1) );
        #local b[i] = pFgetPoint( 2, surf, pp )
    #local i=i+1; #end
    #local ss = pLinterpolante( b )
    ss
#end


/* ----------- SPLINES INTERPOLANTES -----------------*/
// retourne des TABLEAUX de quadriques ou de cubiques
//    SPLINE INTERPOLANTE QUADRIQUE:

#macro spline_interpolante_quadrique( b_1, b, cercle )
    #local n = taille( b );
```

```
    #local spline = array[n-1]
    #local q = array[3]
    #local q[1] = b_1;
    #local i=0; #while (i< n-1)
        #local q[0] = b[i];
        #local q[1] = 2*b[i] - b_1;
        #local q[2] = b[i+1];
        #local spline[i] = q
        #local b_1 = q[1];
        #if (cercle)
            #local spline[i][1] = spline[i][1]*sqrt(2)/2;
        #end
    #local i=i+1; #end
    spline
#end

//    SPLINE INTERPOLANTE CUBIQUE:
#macro spline_interpolante_cubique( b_2, b_1, b )
    #local n = taille( b );
    #local spline = array[n-1]
    #local q = array[4]
    #local i=0; #while (i< n-1)
        #local q[0] = b[i];
        #local q[1] = 2*b[i] -b_1;
        #local q[2] =  4*b[i] -4*b_1 +b_2;
        #local q[3] = b[i+1];
        #local spline[i] = q
        #local b_2 = q[1];
        #local b_1 = q[2];
    #local i=i+1; #end
    spline
#end

/*
       CREATION DE COURBES ET SURFACES PRIMITIVES (dans R4)
*/

/*
     MAILLAGES BIDIM et TRIDM GAUCHES:
input:     le nombre de points de controle sur u,v,w et la ttt
output:    une pSmn ou un pVmnp orthogonaux
             a points de controle equirepartis
*/
```

```
#macro creer_ligne( n1, ttt )
    #local f = array[n1]
    #local i=0; #while (i< n1)
        #local p = < -0.5+ i/(n1-1), 0.0, 0.0, 1 > ;
        #local f[i] = p * < ttt,ttt,ttt,1 > ;
    #local i=i+1; #end
    f
#end

#macro creer_facette( n1, n2, ttt )
    #local f = array[n1]
    #local ff = array[n2]
    #local i=0; #while (i< n1)
        #local j=0; #while (j< n2)
        #local p = < -0.5+ i/(n1-1), -0.5 + j/(n2-1), 0.0, 1 >;
        #local ff[j] = p * < ttt,ttt,ttt,1 > ;
        #local j=j+1; #end
        #local f[i] = ff
    #local i=i+1; #end
    f
#end

#macro creer_cube( n1, n2, n3, ttt )
    #local f = array[n1]
    #local ff = array[n2]
    #local fff = array[n3]
    #local i=0; #while (i< n1)
        #local j=0; #while (j< n2)
            #local k=0; #while (k< n3)
                    #local p = < -0.5+i/(n1-1),
                                 -0.5+j/(n2-1),
                                 -0.5+k/(n3-1), 1 >  ;
            #local fff[k] = p * < ttt,ttt,ttt,1 >  ;
            #local k=k+1; #end
            #local ff[j] = fff
        #local j=j+1; #end
        #local f[i] = ff
    #local i=i+1; #end
    f
#end

#macro pFmaillage( n, nn, ttt )
    #switch (n)
        #case (1)creer_ligne( nn, ttt )    #break;
        #case (2)creer_facette( nn.x, nn.y, ttt )    #break;
```

```
        #case (3)creer_cube(nn.x, nn.y, nn.z, ttt)   #break;
        #case (4)    #render "... not yet !"
    #end
#end

#macro surface_aleatoire( surf, see, delta )
    #local ran = seed( see );
    #local m = taille(surf);
    #local n = taille(surf[0]);
    #local i = 0; #while (i< n)
        #local j=0; #while (j< m)
            #local rrr = delta*rand( ran );
            pFtranslate( 0, surf[j][i], < 0,0,rrr >   )
        #local j=j+1; #end
    #local i=i+1; #end
    surf
#end

/*
     COURBES ARC DE CERCLE:
input:       le rayon du cercle
output:      une pL3 quart de cercle
             une pL4 demi-cercle
             une pL5 demi_cercle, extensible au cercle complet
*/

#macro quart_cercle_3( r, coeff )
    // coeff: attirance du point médian
    #local k = sqrt(2)/2;    // -> ellipse, cercle, hyperbole
    #local p0 = < r, 0, 0, 1 >  ;
    #local p1 = < r, r, 0, 1 >  *k*coeff;
    #local p2 = < 0, r, 0, 1 >  ;
    array[3]     { p0, p1, p2 }
#end

#macro demi_cercle_4( r )
    #local p0 = <  r, 0,  0, 1 >  ;
    #local p1 = <  r, 2*r,0, 1 >  /3;
    #local p2 = < -r, 2*r,0, 1 >  /3;
    #local p3 = < -r, 0,  0, 1 >  ;
    array[4]     { p0, p1, p2, p3 }
#end

#macro demi_cercle_5( r )
    #local k = sqrt(2)/2;
```

```
    #local p0 = <  r, 0,      0, 1 >  ;
    #local p1 = <  r, r,      0, 1 >  *k;
    #local p2 = <  0, 3/2*r,0, 1 >  *2/3;
    #local p3 = < -r, r,      0, 1 >  *k;
    #local p4 = < -r, 0,      0, 1 >  ;
    array[5]     { p0, p1, p2, p3, p4 }
#end

#macro arc_cercle( n, r )
    #switch(n)
        #case (3)    quart_cercle_3( r, 1 ) #break
        #case (4)    demi_cercle_4( r ) #break
        #case (5)    demi_cercle_5( r ) #break
    #end
#end

#macro creer_cylindre( R, H )
    #local k = sqrt(2)/2;
    #local temp = array[2]
    #local temp[0] = array[3]
        {   < R, 0, 0, 1 >,
            < R, R, 0, 1 >*k,
            < 0, R, 0, 1 >
        },
    #local temp[1] = array[3]
        {   < R, 0, H, 1 >,
            < R, R, H, 1 >*k,
            < 0, R, H, 1 >
        }
    temp
#end

#macro creer_tore( R1, R2 )
    #local k = sqrt(2)/2;
    #local R12 = R1+R2;
    #local R2 = abs(R2);
    #local temp = array[3]
    #local temp[0] =   array[3]
        {   < 0,   0, -R12, 1 >,
            < 0,  R2, -R12, 1 >*k,
            < 0,  R2,  -R1, 1 >
        }
    #local temp[1] =   array[3]
        {   < R12,  0, -R12, 1 >*k,
            < R12, R2, -R12, 1 >*k*k,
```

```
            <  R1, R2, -R1,  1 >*k
        }
    #local temp[2] =   array[3]
        {   < R12,  0,  0, 1 >,
            < R12, R2,  0, 1 >*k,
            <  R1, R2,  0, 1 >
        }
    temp
#end

/*
     SURFACE PRODUIT:
input:       deux pLn planes dans Oxy, profil et section
output:      une pSmn
nota:        cas particuliers: prismes et surfaces de révolution
             profil gauche à etudier
*/

#macro cross( c1, c2 )
    #local nb1 = taille( c1 );    // courbes profil
    #local nb2 = taille( c2 );    // courbe section
    #local surf = array[nb1]      // surface produite
    #local i = 0; #while (i< nb1) // pour chaque point du profil
        #local profil = c1[i];    // un point du profil
        #local pp = array[nb2]
        #local j=0; #while (j< nb2)// pour chaque pt  section
            #local section = c2[j];// un pt de la section
        #local pp[j] = <
            section.x*profil.x,    // base sur x
              section.t*profil.y,  // le long de l'axe y
              section.y*profil.x,  // base sur z
              section.t*profil.t  >  ;
        #local j=j+1; #end
        #local surf[i] = pp
    #local i=i+1; #end
    surf
#end

/*
     SURFACE TUBULAIRE
input:       une pLm chemin et une pLn section
             les valeurs de recusrion sur u et v (pre-subdivision)
output:      une pSmn (qui pourra etre subdivisee)
nota:        vérifier le fonctionnement avec les courbes rationnelles
*/
```

```
#macro pipe(che, sec, r1, r2)
    #local chemin  = pFsubdivision( 1, che, < r1,0,0,0 >   )
    #local section = pFsubdivision( 1, sec, < r2,0,0,0 >   )
    #local vmax = taille(chemin);
    #local umax = taille(section);
    #local tube = array[vmax]
    #local pp = array[umax]
        #local i = 0; #while (i< vmax)
            #local mat = pLgetFrenet( che, i/(vmax-1) )
            #local j = 0; #while (j< umax)
            #local pp[j] = pFtransform( mat, section[j] );
            #local j=j+1; #end
            #local tube[i] = pp
        #local i=i+1; #end
    tube
#end

/*
     SURFACE TUBULAIRE ONDULEE
    #local surf = waving_pipe( chemin,section,<4,3>,<0.5,5> )
*/

#macro waving_pipe(che, sec, r, ondulation )
    #local chemin  = pFsubdivision( 1, che, < r.x,0,0,0 >   )
    #local section = pFsubdivision( 1, sec, < r.y,0,0,0 >   )
    #local vmax = taille(chemin);
    #local umax = taille(section);
    #local tube = array[vmax]
    #local pp = array[umax]
        #local i = 0; #while (i< vmax)
             #local uu = i/(vmax-1);
             #local mat = pLgetFrenet( che, uu )
             #local coeff = 1
                    + ondulation.x *sin( 2*pi*uu * ondulation.y );
             #local temp = mat
             #local temp[0][0] = mat[0][0]* coeff;
             #local temp[1][1] = mat[1][1]* coeff;
             #local temp[2][2] = mat[2][2]* coeff;
            #local j = 0; #while (j< umax)
        #local pp[j] = pFtransform( temp, section[j] );
            #local j=j+1; #end
            #local tube[i] = pp
        #local i=i+1; #end
    tube
#end
```

```
/*
    SURFACES PARALLELES
    crée une surface située à la distance dd d'une surface
    autres cas possibles avec modulation de dd
*/

#macro surface_parallele( surf, dd )
    #local m = taille( surf );
    #local n = taille( surf[0] );
    #local S = surf
    #local i = 0; #while (i< n)
        #local j=0; #while (j< m)
        #local mat = pSgetPijk( surf, < j/(n-1),i/(m-1),0,1 >   )
        #local nn = < mat[0][0], mat[1][0], mat[2][0]  >  ;
            #local pp = S[j][i];
            #local pt = pp.t;
            #local qq = < pp.x/pp.t, pp.y/pp.t, pp.z/pp.t >  ;
            #local qq = qq + nn*dd;
        #local S[j][i] = < qq.x*pt,qq.y*pt,qq.z*pt, pt >  ;
        #local j=j+1; #end
    #local i=i+1; #end
    S
#end

/*
    SURFACES COMBINAISONS LINEAIRES
    exemples de surfaces résultant d'une combinaison linéaire d'autres
surfaces ; la somme des coefficients doit être égale à 1.
    le cas le plus connu est celui des surfaces de Coons
*/

/*
    SURFACES SYMETRIQUES
    crée une surface symétrique par rapport à une autre surface
    cas particulier: symétrie plan
*/

#macro surface_symetrique( s1, s2 )    // surf = 2*s1 - s2
    #local M = taille( s1 );
    #local N = taille( s2[0] );
    #local surf = array[M]
    #local i=0; #while (i< M)
        #local pp = array[N]
        #local j=0; #while (j< N)
            #local pp[j] = 2*s1[i][j] - s2[i][j];
```

```
        #local j=j+1; #end
        #local surf[i] = pp
    #local i=i+1; #end
    surf
#end

/*
     SURFACE DE COONS:
input:     4 pLi deux a deux concourantes en quatre points
output:    une pSmn interpolant ces courbes
nota:     les 4 courbes peuvent être de degrés différents
*/

#macro creer_coons( L1, L2, L3, L4 )
    // ajuster les degrés de L1 et L2, et de L3 et L4
    #local M = taille(L1);
    #local m = taille(L2);
    #if (M >  m)     #local L2 = pLup( L2, M-m )    #end
    #if (M< m)       #local L1 = pLup( L1, m-M )    #end
    #local N = taille(L3);
    #local n = taille(L4);
    #if (N >  n)     #local L4 = pLup( L4, N-n )    #end
    #if (N< n)       #local L3 = pLup( L3, n-N )    #end
    #local M = taille(L1);
    #local N = taille(L3);
    // construire s1 (pSM2) sur L1/L2 et élever à pSMN:
    #local s1 = array[2] { L1, L2 }
    #local s1 = pSup( s1, < 0,N-2 >   )
    // construire s2 (pS2N) sur L3/L4 et élever à pSMN:
    #local s2 = array[N]
    #local i=0; #while (i< N)
        #local s2[i] = array[2] { L3[i], L4[i] }
    #local i=i+1; #end
    #local s2 = pSup( s2, < M-2,0 >   )
    // construire s3 (pS22) sur les 4 points d'angle
    // et élever à pSMN:
    #local L21 = array[2] { L1[0], L1[M-1] }
    #local L22 = array[2] { L2[0], L2[M-1] }
    #local s3 = array[2] { L21, L22 }
    #local s3 = pSup( s3, < M-2,N-2 >   )
    // construire coons = s1 + s2 - s3:
    #local coons = array[M]
    #local i=0; #while (i< M)
        #local pp = array[N]
        #local j=0; #while (j< N)
```

```
            #local pp[j] = s1[i][j] + s2[i][j] - s3[i][j];
        #local j=j+1; #end
        #local coons[i] = pp
    #local i=i+1; #end
    coons
#end

// GEODESIQUES
// macro calculant les points d'une geodesique
// input: surface, point depart, angle de tir, nombre de points
// output: tableau des points, not a pCourbe !!!

#macro geodesique( surf, P, A, N, dt )
       #local EPS   = 1.0e-4;
       // premier point en coords locales :
       #local U    = P.x;
       #local V    = P.y;
       #local du    = EPS;
       #local dv    = EPS;
       // calcul du premier point en coords globales :
       #local f     = pFgetPoint( 2, surf, <U,V,0,1> )
       #local fdu   = pFgetPoint( 2, surf, <U+du,V,0,1> )
       #local fdv   = pFgetPoint( 2, surf, <U,V+dv,0,1> )
       #local F            = <f.x/f.t, f.y/f.t, f.z/f.t> ;
       #local FDU   = <fdu.x/fdu.t, fdu.y/fdu.t, fdu.z/fdu.t> ;
       #local FDV   = <fdv.x/fdv.t, fdv.y/fdv.t, fdv.z/fdv.t> ;
       // derivees premieres sur U et V
       #local dfdu = (FDU - F) / du;
       #local dfdv = (FDV - F) / dv;
       #local coeff = vlength( dfdu ) / vlength( dfdv );
       #local tu = cos( A*pi/180 );              // pente sur U
       #local tv = sin( A*pi/180 ) * coeff;      // pente sur V
       #local tableau = array[N]
       #local i=0; #while (i<N)
       // calcul du point en coords globales :
       #local f     = pFgetPoint( 2, surf, <U,V,0,1> )
       #local fdu   = pFgetPoint( 2, surf, <U+du,V,0,1> )
       #local fdv   = pFgetPoint( 2, surf, <U,V+dv,0,1> )
       #local f2du  = pFgetPoint( 2, surf, <U+2*du,V,0,1> )
       #local f2dv  = pFgetPoint( 2, surf, <U,V+2*dv,0,1> )
       #local fdudv = pFgetPoint( 2, surf, <U+du,V+dv,0,1> )
       #local F = <f.x/f.t, f.y/f.t, f.z/f.t> ;
       #local FDU = <fdu.x/fdu.t, fdu.y/fdu.t, fdu.z/fdu.t> ;
       #local FDV = <fdv.x/fdv.t, fdv.y/fdv.t, fdv.z/fdv.t> ;
       #local F2DU = <f2du.x/f2du.t,f2du.y/f2du.t,f2du.z/f2du.t> ;
```

```
	#local F2DV = <f2dv.x/f2dv.t,f2dv.y/f2dv.t,f2dv.z/f2dv.t> ;
	#local FDUDV=<fdudv.x/fdudv.t,fdudv.y/fdudv.t,fdudv.z/fdudv.t>;
	#local dfdu = (FDU - F) / du;  // derivee premiere sur U
	#local dfdv = (FDV - F) / dv;  // derivee premiere sur V
	// derivees secondes sur UU, VV, UV :
	#local d2fduu = ((F2DU  - FDU)/du - (FDU-F)/du)/du;
	#local d2fduv = ((FDUDV - FDU)/du - (FDV-F)/du)/dv;
	#local d2fdvv = ((F2DV  - FDV)/dv - (FDV-F)/dv)/dv;
	#local U = U + tu*dt; // point suivant en coords locales
	#local V = V + tv*dt;   // ...
	#local dudu = vdot(dfdu, dfdu);
	#local dudv = vdot(dfdu, dfdv);
	#local dvdv = vdot(dfdv, dfdv);
	#local cu = -(  tu*tu*vdot(dfdu, d2fduu)
	+ 2*tu*tv*vdot(dfdu, d2fduv) + tv*tv*vdot(dfdu, d2fdvv) );
	#local cv = -(  tu*tu*vdot(dfdv, d2fduu)
	+ 2*tu*tv*vdot(dfdv, d2fduv) + tv*tv*vdot(dfdv, d2fdvv) );
	#local kk = dudu*dvdv - dudv*dudv;
	#if (abs(kk)<EPS)
		#local kk = (kk>=0) ? EPS : -EPS ;
	#end
	#local gu = (cu*dvdv - cv*dudv)/kk;     // acceleration
	#local gv = (cv*dudu - cu*dudv)/kk;     // ...
	#local tu = tu + gu*dt;                 // vitesse
	#local tv = tv + gv*dt;                 // ...
	#local tableau[i] = F;//chargement du point dans tableau
	#local i=i+1; #end
	tableau      // retourne le tableau.
#end

#macro tracer_geodesique( surf, P, A, N, dt, coul, avec_normales )
	var tableau = geodesique( surf, P, A, N, dt )
	union
		{      #local i=0; #while (i<taille(tableau))
			sphere { tableau[i], 0.01  }
			#local i=i+1; #end
			une_couleur(coul)
		}
		#if (avec_normales)
		#local i=0; #while (i<taille(tableau)-2)
				#local p0 = tableau[i];
				#local p1 = tableau[i+1];
				#local p2 = tableau[i+2];
				#local tt = p1-p0;
				#local bb = vcross( p1-p0, p2-p0 );
```

```
                            #local nn = vcross( bb, tt );
                            #local nn = vnormalize( nn );
                            cylinder { p0,p0 - nn*0.1,
                            0.005 pigment { color rgb <1,0,0> } }
              #local i=i+1; #end
              #end
#end

/*----------- MACROS COULEUR ET TEXTURE ------------*/

//#include "golds.inc"
#declare GOLD       = 1;
#declare MIROIR     = 2;
#declare GRANIT     = 3;
#declare MARBRE     = 4;

#macro une_couleur( c )
       texture
       {      pigment { color rgbt c }
              finish {      ambient 0.2
                            diffuse 0.6
                            specular 0.9
                            roughness 0.001
                     //     reflection 0.4 }
       }
#end

#macro une_texture( choix )
    #switch (choix)
    #case (GOLD)
        texture { T_Gold_5A }
    #break
    #case (MIROIR)
        texture
        {    pigment { color rgbt < 0,0,0 >   }
              finish {     ambient 0.2
                           diffuse 0.7
                           specular 0.9
                           roughness 0.005
                           reflection 0.9 }
          }
    #break
    #case (GRANIT)
        texture
        {    pigment
```

```
            {     granite
                color_map
                {
                [ 0.0    color rgbt < 1,1,1,0.0 >   ]
                [ 0.8    color rgbt < 0.2,0.1,0,0.0 >   ]
                [ 1.0    color rgbt < 1,1,0,0.0 >   ]
                }
                scale 1/16
            }
              finish {    ambient 0.2
                          diffuse 0.6
                          specular 0.9
                          roughness 0.005
                          reflection 0.0 }
          }
    #break
    #case (MARBRE)
        texture
        {    pigment
            {     marble
                color_map
                {
                [ 0.0    color rgbt < 0.9,0.9,0.9,0.7 >   ]
                [ 1.0    color rgbt < 0.2,0.2,0.2,0.0 >   ]
                }
                scale 1/32
                //turbulence 0.5
                rotate < 0,0,90 >
            }
              finish {     ambient 0.2
                          diffuse 0.7
                          specular 0.9
                          roughness 0.005
                          reflection 0.0 }
          }
    #break
    #end
#end

/*----------- MACROS POUR LE DESSIN DES COURBES,
            DES SURFACES ET DES VOLUMES ------------*/

//    PROJECTIONS R4 - >   R3:
#macro pLprojection( f )   // cree un tableau de points dans R3
    #local n1 = taille( f );
```

```
    #local ff = array[n1]
    #local i=0; #while (i< n1)
        #local p = f[i];
        #local temp = < p.x/p.t, p.y/p.t, p.z/p.t  >  ;
        #local ff[i] = temp;
    #local i=i+1; #end
    ff
#end

#macro pSprojection( f )    // cree un tableau bidim dans R3
    #local n1 = taille( f );
    #local n2 = taille( f[0] );
    #local ff = array[n1]
    #local pp = array[n2]
    #local i=0; #while (i< n1)
        #local j=0; #while (j< n2)
            #local p = f[i][j];
            #local temp = < p.x/p.t, p.y/p.t, p.z/p.t  >  ;
            #local pp[j] = temp;
        #local j=j+1; #end
        #local ff[i] = pp
    #local i=i+1; #end
    ff
#end

#macro pLgetFrenet( curv, uu )
    #local ff = pFstretch( 1, curv, uu, uu+1 )
#local p0 = <ff[0].x/ff[0].t,ff[0].y/ff[0].t,ff[0].z/ff[0].t> ;
#local p1 = <ff[1].x/ff[1].t,ff[1].y/ff[1].t,ff[1].z/ff[1].t> ;
#local p2 = <ff[2].x/ff[2].t,ff[2].y/ff[2].t,ff[2].z/ff[2].t> ;
    #local tt = p1-p0;
    #local bb = vcross( tt, (p2-p0) );
    #local tt = vnormalize( tt );
    #local bb = vnormalize( bb );
    #local nn = vcross( bb, tt );
    array[4][4] {   { nn.x, bb.x, tt.x, p0.x },
                    { nn.y, bb.y, tt.y, p0.y },
                    { nn.z, bb.z, tt.z, p0.z },
                    { 0,0,0,1 } }
#end

#macro pSgetPijk( surf, p )
    #local ff = pFstretch( 2, surf,
            < p.x,p.y,p.z,p.t >  , < p.x+1,p.y+1,p.z,p.t >   )
    #local p0 = < ff[0][0].x/ff[0][0].t, ff[0][0].y/ff[0][0].t,
```

```
                    ff[0][0].z/ff[0][0].t  >  ;
    #local px = < ff[0][1].x/ff[0][1].t, ff[0][1].y/ff[0][1].t,
                    ff[0][1].z/ff[0][1].t  >  ;
    #local py = < ff[1][0].x/ff[1][0].t, ff[1][0].y/ff[1][0].t,
                    ff[1][0].z/ff[1][0].t  >  ;
    #local tx = px-p0;    #local tx = vnormalize( tx );
    #local ty = py-p0;    #local ty = vnormalize( ty );
    #local nn = vcross( tx, ty );
    array[4][4] {   { nn.x, tx.x, ty.x, p0.x },
                    { nn.y, tx.y, ty.y, p0.y },
                    { nn.z, tx.z, ty.z, p0.z },
                    { 0,0,0,1 } }
#end

#macro pSgetNormale( surf, p )
    #local ff = pFstretch( 2, surf,
            < p.x,p.y,p.z,p.t >  , < p.x+1,p.y+1,p.z,p.t >   )
    #local p0 = < ff[0][0].x/ff[0][0].t, ff[0][0].y/ff[0][0].t,
                    ff[0][0].z/ff[0][0].t  >  ;
    #local px = < ff[0][1].x/ff[0][1].t, ff[0][1].y/ff[0][1].t,
                    ff[0][1].z/ff[0][1].t  >  ;
    #local py = < ff[1][0].x/ff[1][0].t, ff[1][0].y/ff[1][0].t,
                    ff[1][0].z/ff[1][0].t  >  ;
    #local nn = vcross( px-p0, py-p0 );
    #local nn = vnormalize( nn );
    nn
#end

#macro _pScalculer_normales( surf, M, N )
    // methode exacte mais longue
    #local nn = array[M][N]// surf est la surface NON subdivisée
    #local i=0; #while (i< M)// M et N : dims surface subdivisée
        #local j=0; #while (j< N)
    #local nn[i][j] = pSgetNormale(surf,<i/(M-1),j/(N-1),0,1>);
        #local j=j+1; #end
    #local i=i+1; #end
    nn
#end

#macro pScalculer_normales( aa )
    // methode rapide mais approchée
    #local umax = taille(aa); // tableau des normales unitaires
    #local vmax = taille(aa[0]);        //
    #local nn = array[umax][vmax]
    // calcul des nn sauf aux bords maxis
```

```
    #local i = 0; #while (i< umax-1)
        #local j = 0; #while (j< vmax-1)
            #local p0 = aa[i][j];
            #local p1 = aa[i+1][j];
            #local p2 = aa[i][j+1];
            #local nn[i][j] =
                    vnormalize(vcross((p1-p0),(p2-p0)));
        #local j=j+1; #end
    #local i=i+1; #end
    // calcul aux bords
    #local i = 0; #while (i< umax-1)
      #local nn[i][vmax-1] = nn[i][vmax-2]*vdot(nn[i][vmax-3],
                          nn[i][vmax-2])*2-nn[i][vmax-3];
    #local i=i+1; #end
    #local i = 0; #while (i< vmax-1)
       #local nn[umax-1][i] = nn[umax-2][i]*vdot(nn[umax-2][i],
                          nn[umax-3][i])*2-nn[umax-3][i];
    #local i=i+1; #end
    #local nn[umax-1][vmax-1] =
                    nn[umax-2][vmax-2]*vdot(nn[umax-2][vmax-2],
                    nn[umax-3][vmax-3])*2 - nn[umax-3][vmax-3];
// moyenne des normales sur 2 triangles adjacents sur la diag
    #local i = 1; #while (i< umax-1)
        #local j = 1; #while (j< vmax-1)
            #local nn[i][j] = (nn[i-1][j-1]
                          + 2*nn[i][j] + nn[i+1][j+1])/4;
        #local j=j+1; #end
    #local i=i+1; #end
    nn
#end

//    MACROS DE DESSIN DE COURBES, DE SURFACES ET DE VOLUMES:

#macro _pLdraw( f, rayon ) // dessine une suite de cylindres
    #local ff = pLprojection( f )
    #local n = taille( f );
    #local i = 0; #while (i< n-1)
        cylinder { ff[i], ff[i+1], rayon }
    #local i=i+1; #end
#end

#macro _pSdraw( ff, nn, qualite )
    // dessine un pFmaillage de triangles
    #local umax = taille(ff);
    #local vmax = taille(ff[0]);
```

```
    mesh
    {    #local i = 0; #while (i< umax-1)
        #local j = 0; #while (j< vmax-1)
            #local p00 = ff[i][j];
            #local p10 = ff[i+1][j];
            #local p01 = ff[i][j+1];
            #local p11 = ff[i+1][j+1];
            #if (qualite)
                #local n00 = nn[i][j];
                #local n10 = nn[i+1][j];
                #local n01 = nn[i][j+1];
                #local n11 = nn[i+1][j+1];
            #end
            #if (qualite)
        smooth_triangle { p00, n00, p10, n10, p11, n11 }
        smooth_triangle { p00, n00, p11, n11, p01, n01 }
            #else
                triangle { p00, p10, p11 }
                triangle { p00, p11, p01 }
            #end
        #local j=j+1; #end
        #local i=i+1; #end
    }
#end

#macro _pVfaces( vvv )// retourne les 6 surfaces limites d'un volume
    #local nx = taille( vvv );
    #local ny = taille( vvv[0] );
    #local nz = taille( vvv[0][0] );
    #local surf = array[6]
    #local ss = array[nx]
    #local i=0;    #while (i< nx)
        #local pp = array[ny]
        #local j=0;    #while (j< ny)
            #local pp[j] = vvv[nx-1-i][ny-1-j][nz-1];
        #local j=j+1; #end
        #local ss[i] = pp
    #local i=i+1; #end
    #local surf[0] = ss
    #local ss = array[nx]
    #local i=0;    #while (i< nx)
        #local pp = array[nz]
        #local j=0;    #while (j< nz)
            #local pp[j] = vvv[nx-1-i][ny-1][j];
        #local j=j+1; #end
```

```
        #local ss[i] = pp
    #local i=i+1; #end
    #local surf[1] = ss
    #local ss = array[ny]
    #local i=0;    #while (i< ny)
        #local pp = array[nz]
        #local j=0;    #while (j< nz)
            #local pp[j] = vvv[nx-1][i][j];
        #local j=j+1; #end
        #local ss[i] = pp
    #local i=i+1; #end
    #local surf[2] = ss
    #local ss = array[nx]
    #local i=0;    #while (i< nx)
        #local pp = array[ny]
        #local j=0;    #while (j< ny)
            #local pp[j] = vvv[nx-1-i][j][0];
        #local j=j+1; #end
        #local ss[i] = pp
    #local i=i+1; #end
    #local surf[3] = ss
    #local ss = array[nx]
    #local i=0;    #while (i< nx)
        #local pp = array[nz]
        #local j=0;    #while (j< nz)
            #local pp[j] = vvv[nx-1-i][0][nz-1-j];
        #local j=j+1; #end
        #local ss[i] = pp
    #local i=i+1; #end
    #local surf[4] = ss
    #local ss = array[ny]
    #local i=0;    #while (i< ny)
        #local pp = array[nz]
        #local j=0;    #while (j< nz)
            #local pp[j] = vvv[0][i][nz-1-j];
        #local j=j+1; #end
        #local ss[i] = pp
    #local i=i+1; #end
    #local surf[5] = ss
    surf
#end

#macro pVdrawPijk( vol, p, longueur, rayon )
// A REVOIR, REPERE NON ORTHONORME
    // dessine le repère tangent en p au volume
```

```
    #local p0 = pFgetPoint( 3, vol, <p.x, p.y, p.z, p.t> )
    #local pu = pFgetPoint( 3, vol, <p.x+1, p.y, p.z, p.t> )
    #local pv = pFgetPoint( 3, vol, <p.x, p.y+1, p.z, p.t> )
    #local pw = pFgetPoint( 3, vol, <p.x, p.y, p.z+1, p.t> )
    #local q0 = <p0.x/p0.t, p0.y/p0.t, p0.z/p0.t> ;
       #local qu = <pu.x/pu.t, pu.y/pu.t, pu.z/pu.t> ;
       #local qv = <pv.x/pv.t, pv.y/pv.t, pv.z/pv.t> ;
       #local qw = <pw.x/pw.t, pw.y/pw.t, pw.z/pw.t> ;
    #local ti = vnormalize( qu-q0 )
    #local tj = vnormalize( qv-q0 )
    #local tk = vnormalize( qw-q0 )
    union
{ cylinder{q0,q0+ti*longueur, rayon une_couleur( < 1,0,0 >  ) }
  cylinder{q0,q0+tj*longueur, rayon une_couleur( < 0,1,0 >  ) }
  cylinder{q0,q0+tk*longueur, rayon une_couleur( < 0,0,1 >  ) }
}
#end

#macro pSdrawPijk( surf, p, longueur, rayon )
    // dessine le repère tangent en p à la surface
    #local mat = pSgetPijk( surf, p )
    #local nn = < mat[0][0], mat[1][0], mat[2][0]  >  ;
    #local tx = < mat[0][1], mat[1][1], mat[2][1]  >  ;
    #local ty = < mat[0][2], mat[1][2], mat[2][2]  >  ;
    #local p0 = < mat[0][3], mat[1][3], mat[2][3]  >  ;
    union
{ cylinder{p0,p0-nn*longueur, rayon une_couleur( < 1,0,0 >  ) }
  cylinder{p0,p0+tx*longueur, rayon une_couleur( < 0,1,0 >  ) }
  cylinder{p0,p0+ty*longueur, rayon une_couleur( < 0,0,1 >  ) }
}
#end

#macro pSdrawNormale( surf, p, longueur, rayon )
    // dessine la normale en p à la surface
    #local ff = pFstretch( 2, surf,
             < p.x,p.y,p.z,p.t >, < p.x+1,p.y+1,p.z,p.t >   )
    #local p0 = < ff[0][0].x/ff[0][0].t,
             ff[0][0].y/ff[0][0].t, ff[0][0].z/ff[0][0].t  >;
    #local px = < ff[0][1].x/ff[0][1].t,
             ff[0][1].y/ff[0][1].t, ff[0][1].z/ff[0][1].t  >;
    #local py = < ff[1][0].x/ff[1][0].t,
             ff[1][0].y/ff[1][0].t, ff[1][0].z/ff[1][0].t  >;
    #local nn = vcross( px-p0, py-p0 );
    #local nn = vnormalize( nn );
    cylinder { p0, p0-nn*longueur, rayon }
```

```
#end

#macro pLdrawFrenet( curv, uu, longueur, rayon )
    // dessine le repère de frenet en uu à la courbe
    #local mat = pLgetFrenet( curv, uu )
    #local nn = < mat[0][0], mat[1][0], mat[2][0]  >  ;
    #local bb = < mat[0][1], mat[1][1], mat[2][1]  >  ;
    #local tt = < mat[0][2], mat[1][2], mat[2][2]  >  ;
    #local p0 = < mat[0][3], mat[1][3], mat[2][3]  >  ;
    union
    { cylinder{ p0,p0+nn*longueur, rayon une_couleur(<0,1,0>) }
      cylinder{ p0,p0+bb*longueur, rayon une_couleur(<0,0,1>) }
      cylinder{ p0,p0+tt*longueur, rayon une_couleur(<1,0,0>) }
    }
#end

//    INTERFACE POUR LE DESSIN DES FORMES:

//    1)    GESTION DES PARAMETRES D'AFFICHAGE

#declare STANDARD       = -1;
#declare POINT          = 0;
#declare COURBE         = 1;
#declare SURFACE        = 2;
#declare ENVELOPPE      = 3;
#declare HYPER          = 4;
#declare FEUILLES       = 5;
#declare FIBRES         = 6;
#declare NORMALES       = 7;
#declare REPERE         = 8;
#declare COULEUR        = 0;
#declare TEXTURE        = 1;
#declare FACETTES       = -1;
#declare LISSE          = 0;
#declare SUPER          = 1;
#declare FORME          = POINT;
#declare ASPECT         = COULEUR;
#declare QUALITE        = FACETTES;
#declare RAYON          = 0.05;
#declare LONGUEUR       = 0.1;
#declare CHOIX_COULEUR  = < 1,0,0 >;
#declare CHOIX_TEXTURE  = 1;

#macro standards()
    #declare FORME          = POINT;
```

```
    #declare ASPECT         = COULEUR;
    #declare QUALITE        = FACETTES;
    #declare RAYON          = 0.05;
    #declare LONGUEUR       = 0.1;
    #declare CHOIX_TEXTURE  = 1;
    #ifdef (CHOIX_COULEUR) #undef CHOIX_COULEUR #end
    #declare CHOIX_COULEUR  = < 1,0,0 >;
    #ifdef (RECURSION) #undef RECURSION #end
#end

#macro finesse( r )
    #ifdef (RECURSION) #undef RECURSION #end
    #declare RECURSION = r; 0
#end

#macro point( r )
    #declare FORME = POINT; #declare RAYON = r; 0
#end

#macro courbe( r )
    #declare FORME = COURBE; #declare RAYON = r; 0
#end

#macro surface( q )
    #declare FORME = SURFACE; #declare QUALITE = q; 0
#end

#macro enveloppe( q )
    #declare FORME = ENVELOPPE; #declare QUALITE = q; 0
#end

#macro feuilles( q )
    #declare FORME = FEUILLES; #declare QUALITE = q; 0
#end

#macro fibres( r )
    #declare FORME = FIBRES; #declare RAYON = r; 0
#end

#macro normales( r, l )
    #declare FORME = NORMALES; #declare RAYON = r;
    #declare LONGUEUR = l; 0
#end

#macro repere( r, l )
```

```
    #declare FORME = REPERE; #declare RAYON = r;
    #declare LONGUEUR = l; 0
#end

#macro ma_couleur( c )
    #declare ASPECT = COULEUR;
    #ifdef (CHOIX_COULEUR) #undef CHOIX_COULEUR #end
    #declare CHOIX_COULEUR = c; 0
#end

#macro ma_texture( c )
    #declare ASPECT = TEXTURE; #declare CHOIX_TEXTURE = c; 0
#end

//    2)    APPEL DES MACROS

#macro __pSdraw( f, ff, etat )
    #switch (etat)
    #case (FACETTES)
        #local nn = ff    // pour leurer la macro _draw_surface
        _pSdraw( ff, nn, false)
    #break
    #case (LISSE)
        #local nn = pScalculer_normales( ff )
        _pSdraw( ff, nn, ((QUALITE >=LISSE) ? true : false))
    #break
    #case (SUPER)
        #local nn =
    _pScalculer_normales( f, taille(ff), taille(ff[0]) )
        _pSdraw( ff, nn, ((QUALITE >=LISSE) ? true : false))
    #break
    #end
#end

#macro pPdraw( f )
    pFdraw( 0, f, RAYON )
#end

#macro pLdraw( f )
    #ifndef (RECURSION) #declare RECURSION = 0; #end
    #local pf = pFsubdivision( 1, f, < RECURSION,0,0,0 >   )
    #local ppf = pLprojection( pf )
    #switch (FORME)
    #case (POINT)
        union { pFdraw( 1, pf, RAYON )     }
```

```
    #break
    #case (COURBE)
        union { _pLdraw( pf, RAYON ) }
    #break
    #case (REPERE)
        #local n = taille( pf );
        union
        {    #local i = 0; #while (i< n)
            pLdrawFrenet( f, i/(n-1), LONGUEUR, RAYON )
            #local i=i+1; #end
        }
    #break
    #end
#end

#macro pSdraw( f )
    #ifndef (RECURSION) #declare RECURSION = < 0,0 >  ; #end
    #local pf =
    pFsubdivision( 2, f, < RECURSION.x,RECURSION.y,0,0 >   )
    #local ppf = pSprojection( pf )
    #switch (FORME)
    #case (POINT)
        union { pFdraw( 2, pf, RAYON ) }
    #break
    #case (COURBE)
        union
        {   #local i=0; #while(i< taille(pf))
                _pLdraw( pf[i], RAYON )
            #local i=i+1; #end
        }
    #break
    #case (SURFACE)
        __pSdraw( f, ppf, QUALITE )
    #break
    #case (NORMALES)
    // il faut calculer le juste point base de la normale...
        #local m = taille(ppf);
        #local n = taille(ppf[0]);
        #local nn = _pScalculer_normales( f, m, n  )
        union
        {    #local i=0; #while(i< m)
            #local j=0; #while(j< n)
    cylinder{ ppf[i][j], ppf[i][j] + nn[i][j]*LONGUEUR, RAYON}
            #local j=j+1; #end
            #local i=i+1; #end
```

```
        }
    #break
    #end
#end

#macro pVdraw( f )
    #ifndef (RECURSION) #declare RECURSION = < 0,0,0 >  ; #end
    #switch (FORME)
    #case (POINT)
        #local pf = pFsubdivision( 3, f, <
                    RECURSION.x,RECURSION.y,RECURSION.z,0 >   )
        union { pFdraw( 3, pf, RAYON ) }
    #break
    #case (ENVELOPPE)
        #local surf = _pVfaces( f )
        union
        {   #local i=0; #while (i< 6)
            #local psurfi = pFsubdivision( 2, surf[i], <
                     RECURSION.x,RECURSION.y,RECURSION.z,0 > )
            #local ffi = pSprojection( psurfi )
            __pSdraw( surf[i], ffi, QUALITE )
            #local i=i+1;    #end
        }
    #break
    #case (FEUILLES)
         #local pf = pFsubdivision(3,f,RECURSION )
        union
        {   #local n = taille(pf);
            #local i=0; #while (i< n)
            #local psurfi = pf[i]
            #local ffi = pSprojection( psurfi )
            __pSdraw( pf[i], ffi, QUALITE )
            #local i=i+1; #end
        }
    #break
    #case (FIBRES)
        #local pf = pFsubdivision( 3, f, <
                     RECURSION.x,RECURSION.y,RECURSION.z,0 >  )
        union
        {   #local nb = taille(pf);
            #local m = taille(pf[0]);
            #local n = taille(pf[0][0]);
            #local i=0; #while (i< m)
            #local j=0; #while (j< n)
                #local poil = array[nb]
```

```
                #local k=0; #while (k< nb)
                    #local poil[k] = pf[k][i][j];
                #local k=k+1; #end
                _pLdraw( poil, RAYON )
            #local j=j+1; #end
            #local i=i+1; #end
        }
    #break
    #end
#end

#macro pHdraw( f )
    #ifndef (RECURSION) #declare RECURSION = < 0,0,0,0 >  ; #end
    #local pf = pFsubdivision( 4, f, RECURSION )
    union { pFdraw( 4, pf, RAYON ) }
#end

//    2)    MACRO APPELANTE GENERALE

#macro draw( dim, f, options )
    object
    {    #switch (dim)
            #case (0)    pPdraw( f )    #break
            #case (1)    pLdraw( f )    #break
            #case (2)    pSdraw( f )    #break
            #case (3)    pVdraw( f )    #break
            #case (4)    pHdraw( f )    #break
            #else    #render "...nothing out there! Yet..."
        #end
        #if (ASPECT = COULEUR)
            une_couleur( CHOIX_COULEUR ) #end
        #if (ASPECT = TEXTURE)
            une_texture( CHOIX_TEXTURE ) #end
    }
    standards() // réinitialise les valeurs des paramètres
#end

/*----------- MACROS UTILES ------------*/

#macro axes( Ox, Oy, Oz )
    #if (Ox) cylinder { < -10,0,0 >  ,< 10,0,0 >  , 0.005
            pigment{ color rgb  < 1,0,0 >   } }     #end
    #if (Oy) cylinder { < 0,-10,0 >  , < 0,10,0 >  , 0.005
            pigment { color rgb  < 0,1,0 >   } }    #end
    #if (Oz) cylinder { < 0,0,-10 >  , < 0,0,10 >  , 0.005
```

```
            pigment { color rgb  < 0,0,1 >   } }    #end
#end

#declare OX  = 1;
#declare OY  = 2;
#declare OZ  = 3;
#declare AXO = 4;

#macro vue_axonometrique( axe, zoom )
    camera
    {    orthographic
        #switch (axe)
            #case (OX)    location < 10,0,0 >        #break
            #case (OY)    location < 0,10,0 >        #break
            #case (OZ)    location < 0,0,-10 >       #break
            #case (AXO)   location < 10,10,-10 >     #break
        #end
        right < 1,0,0 >
        look_at < 0,0,0 >   scale 1/zoom
    }
    #switch (axe)
        #case (OX)    axes( false, true, true ) #break
        #case (OY)    axes( true, false, true ) #break
        #case (OZ)    axes( true, true, false ) #break
        #case (AXO)   axes( true, true, true )  #break
    #end
#end

#macro vue_perspective( observateur )
    camera {    location observateur
                right < 1,0,0 > look_at < 0,0,0 >   }
    axes( true, true, true )
#end

#macro sol( hauteur )
    plane
    {     < 0,1,0 >  , hauteur
        texture
        {    pigment {  color rgb 0.5 }
        // checker color rgb 0.3 color rgb 0.7 scale 1/4
            finish {     ambient 0.2
                    diffuse 0.5
                    specular 0.7
                    roughness 0.01
                    reflection 0.8
```

```
                }
        }
    }
#end

#macro ciel()
    sky_sphere
    {     pigment
        {     marble    // gradient y
            color_map {
                    [0.0 color rgb 1/4]
                    [1.0 color rgb 3/4]
                }
            //scale 1/2
            turbulence 0.5
            rotate < 0,0,90 >
        }
    }
#end

/*
       DIVERSES MACROS UTILES
*/

#macro taille( _f ) // retourne la taille d'un tableau
    dimension_size( _f,1 )
#end

/*
       AFFICHAGE DES COMPOSANTES D'UN VECTEUR
*/
#macro affiche_vect( vvv )
    #render "\n vecteur: < "
    #local n = taille( vvv );
    #local i=0; #while (i< n)
        #render concat( str( vvv[i], 4,4 ), "," )
    #local i=i+1; #end
    #render " >  \n"
#end

/*
    AFFICHAGE DES TERMES D'UNE MATRICE
*/
#macro affiche_mat( mat )    //
    #render "\n matrice:\n "
```

```
    #local n1= dimension_size( mat, 1 );
    #local n2= dimension_size( mat, 2 );
    #local i=0; #while (i< n1)        // pour chaque ligne i
        #render concat( "ligne ", str(i,0,0), ": " )
        #local j=0; #while (j< n2)    // pour chaque colonne j
            #render concat( str( mat[i][j],4,4 ), ", " )
        #local j=j+1; #end
        #render "\n"
    #local i=i+1; #end
    #render "\n"
#end


/*
    INVERSION D'UNE MATRICE CARREE
    PAR LA METHODE DE GAUSS_JORDAN
    test:
    #local mat = array[3][3] { {3,2,1}, {1,3,2}, {2,4,3} }
    affiche_mat( mat )
    #local tam = inverse_mat( mat )
    affiche_mat( tam )
    stop
*/

#macro inverse_mat( ma )
    #local N = taille(ma);
    // creation d'une matrice [N][2*N]
    #local mat = array[N][2*N]
    #local I=0; #while (I< N)
        #local J=0; #while (J< N)
            #local mat[I][J] = ma[I][J];
        #local J=J+1; #end
    #local I=I+1; #end
    #local K=0; #while (K< N)
        #local I=0; #while (I< N)
            #if (K=I)
                #local mat[I][N+K] = 1;
            #else
                #local mat[I][N+K] = 0;
            #end
        #local I=I+1; #end
    #local K=K+1; #end
    // start:
    #local EPS  = 0.0000001;
    #local singularite = false;
```

```
#local DetA = 1;
#local K = 0;    #while ( (K< N) & (!singularite) )
// recherche de l'indice L du pivot maximum
    #local L=K;
    #local I=K; #while (I< N)
        #if ( abs(mat[I][K])  >   abs(mat[L][K]) )
            #local L=I;
        #end
    #local I=I+1; #end
    #if ( abs(mat[L][K]) <= EPS )
        #local singularite = true;
    #else
        #if (L!=K)
            #local I=K; #while (I< 2*N)
                #local mmm = mat[K][I];
                #local mat[K][I] = mat[L][I];
                #local mat[L][I] = mmm;
            #local I=I+1; #end
            #local DetA = -DetA;
        #end
    #end
    #local DetA = DetA*mat[K][K];
// elimination de xk
    #if (!singularite)
        #local J=K+1; #while (J< 2*N)
        #local mat[K][J] = mat[K][J]/mat[K][K];
        #local J=J+1; #end
        #local mat[K][K] = 1;
        #local I=0; #while (I< N)
            #if (I!=K)
                #local J=K+1; #while (J< 2*N)
                #local mat[I][J] = mat[I][J] -
                     (mat[I][K]*mat[K][J]);
                #local J=J+1; #end
                #local mat[I][K] = 0;
            #end
        #local I=I+1; #end
    #end
#local K=K+1; #end
#local tam = array[N][N]
#local I=0; #while (I< N)
    #local J=0; #while (J< N)
        #local tam[I][J] = mat[I][J+N];
    #local J=J+1; #end
#local I=I+1; #end
```

```
    tam
#end
#macro produit_mat( mat, vect )
    #local n= dimension_size( mat, 1 );
    #local tcev = array[n]
    #local i=0; #while (i< n)
        #local dd = 0;
        #local j=0; #while (j< n)
            #local dd = dd + mat[i][j]*vect[j];
        #local j=j+1; #end
        #local tcev[i] = dd;
    #local i=i+1; #end
    tcev
#end

/* ----------- EXPORT ET IMPORT DE FICHIERS FORMES --------- */

/*
    conserve une forme calculée sous forme texte
    utile dans le cas d'animations ou éventuellement
    pour transmettre des informations à d'autres programmes
*/

/*
    ECRITURE D,UNE COURBE DANS UN FICHIER:
    input:   la courbe et le nom du fichier
    output:  le fichier
    nota:    attention a ne pas ecraser un fichier
             de meme nom, il n'y a pas de controle...
    appel:   #local courb = creer_une_courbe(x,x,...)
             ecrire_courbe( courb, "fichier.txt" )
*/
#macro pLwrite( f, fichier )
    #fopen myfile fichier write
    #local umax = taille(f);
    #write( myfile, "\"Une pCourbe\",\n" )
    #write( myfile, umax, ",\n" )
    #local i=0; #while (i< umax)
        #write( myfile, f[i], ",\n" )
    #local i=i+1; #end
    #fclose myfile
#end
```

```
/*
    LECTURE D,UNE COURBE DEPUIS UN FICHIER:
    input:          le nom du fichier
    output:         la courbe
    nota:
    appel:          #local courb = lire_courbe( "fichier.txt" )
*/
#macro pLread( fichier )
    #fopen myfile fichier read
    #read( myfile, titre )
    #read( myfile, umax )
    #local ff = array[umax]
    #local i=0; #while (i< umax)
        #ifdef (ddd) #undef ddd #end    // peut etre a revoir
        #read( myfile, ddd )
        #local ff[i] = ddd;
    #local i=i+1; #end
    #fclose myfile
    /*return*/ ff
#end

/*
    ECRITURE D,UNE SURFACE DANS UN FICHIER:
    input:   la surface et le nom du fichier
    output:  le fichier
    nota:    attention a ne pas ecraser un fichier de meme nom
    appel:   #local surf = creer_une_surface(x,x,...)
             ecrire_surface( surf, "fichier.txt" )
*/
#macro pSwrite( f, fichier )
    #fopen myfile fichier write
    #local umax = taille(f);
    #local vmax = taille(f[0]);
    #write( myfile, "\"Une pSurface\",\n" )
    #write( myfile, umax, ",\n" )
    #write( myfile, vmax, ",\n" )
    #local i=0; #while (i< umax)
        #local j=0; #while (j< vmax)
            #write( myfile, f[i][j], ",\n" )
        #local j=j+1; #end
    #local i=i+1; #end
    #fclose myfile
#end
```

```
/*
     LECTURE D,UNE SURFACE DEPUIS UN FICHIER:
    input:   le nom du fichier
    output:  la surface
    nota:
    appel:   #local surf = lire_surface( "fichier.txt" )
*/
#macro pSread( fichier )
    #fopen myfile fichier read
    #read( myfile, titre )
    #read( myfile, umax )
    #read( myfile, vmax )
    #local ff = array[umax]
    #local pp = array[vmax]
    #local i=0; #while (i< umax)
        #local j=0; #while (j< vmax)
            #ifdef (ddd) #undef ddd #end
            #read( myfile, ddd )
            #local pp[j] = ddd;
        #local j=j+1; #end
        #local ff[i] = pp
    #local i=i+1; #end
    #fclose myfile
    /*return*/ ff
#end


//    end of the pFlibs.inc file, the show can start now...
```

pFbook.inc file

Macros used for rendering most of the pictures in this book habe been compiled in the « pFbook.inc » file and are listed below, following the structure of the different sections. They illustrate the unitary approach and can be used as a basis for further explorations.

```
// file « pFbook.inc »
// under GPL open source licence
// http://marty.alain.free.fr
// version of 01/06/2004

// 11 : FORMES MULTILINEAIRES RECURSIVES

#macro pF_111() // 4 points
    draw(0,<-0.5,-0.5,-0.5,1.0>, point(0.1)+ ma_couleur(<1,0,0,0>))
    draw(0,< 0.5, 0.5, 0.5,1.0>, point(0.1) + ma_texture(GRANIT))
    draw(0,< 0.5, 0.5,-0.5,1.0>, point(0.1) + ma_texture(MARBRE))
    draw(0,< 0.0, 0.0, 0.0, 1.0>, point(0.3) + ma_texture(GOLD))
#end

#macro pF_112( aspect ) // pL2
    #local p0 = <-0.5,-0.5,-0.5, 1.0> ;
    #local p1 = < 0.5,-0.0,-0.5, 1.0> ;
    #local pL2 = array[2] { p0, p1 }
    #switch (aspect)
    #case (0) draw( 1, pL2,
       finesse(0)+ point(0.02) + ma_couleur(<1,1,0,0>) ) #break
    #case (1) draw( 1, pL2,
       finesse(1)+ point(0.02) + ma_couleur(<1,1,0,0>) ) #break
    #case (2) draw( 1, pL2,
       finesse(2)+ point(0.02) + ma_couleur(<1,1,0,0>) ) #break
    #case (3) draw( 1, pL2,
       finesse(3)+ point(0.02) + ma_couleur(<1,1,0,0>) ) #break
    #case (4) draw( 1, pL2,
       finesse(0)+ courbe(0.02)+ ma_couleur(<1,1,0,0>) ) #break
    #end
#end

#macro pF_113( aspect ) // pS22
    #local pL2_0 = array[2] { <-0.5,-0.5,-0.5, 1.0>,
                                        < 0.5, 0.0,-0.5, 1.0> }
    #local pL2_1 = array[2] { <-0.5, 0.0, 0.5, 1.0>,
```

```
                                  < 0.5,-0.5, 0.5, 1.0> }
    #local pS22  = array[2] { pL2_0, pL2_1 }
    #switch (aspect)
    #case (0) draw( 2, pS22,
       finesse(<0,0>) + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (1) draw( 2, pS22,
       finesse(<1,0>) + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (2) draw( 2, pS22,
       finesse(<2,0>) + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (3) draw( 2, pS22,
       finesse(<3,0>) + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (4) draw( 2, pS22,
       finesse(<3,3>)+surface(LISSE)+ma_couleur(<1,1,0,0.5>))  #break
    #case (5) draw( 2, pS22,
       finesse(<3,3>)+point(0.01) + ma_couleur(<1,1,0,0.5>) ) #break
    #case (6) draw( 2, pS22,
       finesse(<3,3>)+normales(0.01,-0.1)
                                   + ma_couleur(<1,1,0,0.5>)) #break
    #end
#end

#macro pF_114( aspect ) // pV222
    #local pL2_0 = array[2] { <-0.5,-0.5,-0.5, 1.0>,
                                        < 0.5, 0.0,-0.5, 1.0> }
    #local pL2_1 = array[2] { <-0.5, 0.0, 0.5, 1.0>,
                                        < 0.5,-0.5, 0.5, 1.0> }
    #local pS22  = array[2] { pL2_0, pL2_1 }
    #local pS22_0 = pS22
    #local pS22_1 = pS22
       pFtranslate( 2, pS22_1, <0, 0.5,0> )
       pFrotate( 2, pS22_1, <0,-15,0> )
    #local pV222 = array[2] { pS22_0, pS22_1 }
    #switch (aspect)
    #case (0) draw( 3, pV222, finesse(<0,3,3,0>)
              + feuilles(LISSE) + ma_couleur(<1,1,0,0>) ) #break
    #case (1) draw( 3, pV222, finesse(<1,3,3,0>)
              + feuilles(LISSE) + ma_couleur(<1,1,0,0>) ) #break
    #case (2) draw( 3, pV222, finesse(<2,3,3,0>)
              + feuilles(LISSE) + ma_couleur(<1,1,0,0>) ) #break
    #case (3) draw( 3, pV222, finesse(<3,3,3,0>)
              + feuilles(LISSE) + ma_couleur(<1,1,0,0>) ) #break
    #case (4) draw( 3, pV222,finesse(<2,3,3,0>)
              + enveloppe(LISSE) + ma_couleur(<1,1,0,0.5>) ) #break
    #case (5) draw( 3, pV222, finesse(<0,3,3,0>)
              + fibres(0.01) + ma_couleur(<1,1,0,0>) ) #break
```

```
    #case (6) draw( 3, pV222, finesse(<3,3,3,0>)
              + point(0.01) + ma_couleur(<1,0,0,0>) ) #break
    #end
#end

#macro pF_115() // pH2222
    #local pL2_0 = array[2] { <-0.5,-0.5,-0.5, 1.0>,
                                        < 0.5, 0.0,-0.5, 1.0> }
    #local pL2_1 = array[2] { <-0.5, 0.0, 0.5, 1.0>,
                                        < 0.5,-0.5, 0.5, 1.0> }
    #local pS22  = array[2] { pL2_0, pL2_1 }
    #local pS22_0 = pS22
    #local pS22_1 = pS22
      pFtranslate( 2, pS22_1, <0, 0.5,0> )
      pFrotate( 2, pS22_1, <0,-15,0> )
    #local pV222 = array[2] { pS22_0, pS22_1 }
    #local pV222_0 = pV222
      pFscale( 3, pV222_0, <1.0,1.8,1.0> )
    #local pV222_1 = pV222
      pFscale( 3, pV222_1, <0.8,0.8,0.8> )
    #local pH2222 = array[2] { pV222_0, pV222_1 }
    draw( 4, pH2222,
      finesse(<2,3,3,3>) + point(0.01) + ma_couleur(<1,1,0,0>) )
#end

// 12 : DIAGONALISATION

#macro pF_121( aspect ) // pS22 et pL3
    #local pL2_0 = array[2] {    <-0.5,-0.5,-0.5, 1.0>,
                                 < 0.5, 0.0,-0.5, 1.0> }
    #local pL2_1 = array[2] {    <-0.5, 0.5, 0.5, 1.0>,
                                 < 0.5,-0.5, 0.5, 1.0> }
    #local pS22  = array[2] { pL2_0, pL2_1 }
    #local pL3 = pFdiagonalisation( 2, pS22 )
    draw( 2, pS22,
      finesse(<3,3>)+surface(LISSE)+ma_couleur(<1,1,0,0.5>) )
    #switch (aspect)
    #case (0) draw( 1, pL3,
      finesse(0) + point(0.02) + ma_couleur(<1,0,0,0>) ) #break
    #case (1) draw( 1, pL3,
      finesse(1) + point(0.02) + ma_couleur(<1,0,0,0>) ) #break
    #case (2) draw( 1, pL3,
      finesse(2) + point(0.02) + ma_couleur(<1,0,0,0>) ) #break
    #case (3) draw( 1, pL3,
      finesse(3) + point(0.02) + ma_couleur(<1,0,0,0>) ) #break
```

```
    #case (4) draw( 1, pL3,
       finesse(6) + courbe(0.02) + ma_couleur(<1,1,0,0>) )
       draw( 1, pL3,
       finesse(0) + point(0.04) + ma_couleur(<1,0,0,0>) )
    #break
    #end
#end

#macro pF_122( aspect ) // pS23 et pL4
    #local pL3_0 = array[3] { <-0.5,-0.5,-0.5, 1.0>,
             < 0.0, 0.5,-0.5, 1.0>, < 0.5,-0.5,-0.5, 1.0> }
    #local pL3_1 = array[3] { <-0.5, 0.5, 0.5, 1.0>,
             < 0.0,-0.5, 0.5, 1.0>, < 0.5, 0.5, 0.5, 1.0> }
    #local pS23  = array[2] { pL3_0, pL3_1 }
    #local pL4 = pFdiagonalisation( 2, pS23 )
    #switch (aspect)
    #case (0) draw( 2, pS23, finesse(<0,4>)
             + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (1) draw( 2, pS23, finesse(<1,4>)
             + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (2) draw( 2, pS23, finesse(<2,4>)
             + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (3) draw( 2, pS23, finesse(<3,4>)
             + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (4) draw( 2, pS23, finesse(<4,4>)
             + surface(LISSE) + ma_couleur(<1,1,0,0.5>) ) #break
    #end
    #switch (aspect)
    #case (0) draw( 1, pL4, finesse(0) + point(0.02)
                   + ma_couleur(<1,0,0,0>) ) #break
    #case (1) draw( 1, pL4, finesse(1) + point(0.02)
                   + ma_couleur(<1,0,0,0>) ) #break
    #case (2) draw( 1, pL4, finesse(2) + point(0.02)
                   + ma_couleur(<1,0,0,0>) ) #break
    #case (3) draw( 1, pL4, finesse(3) + point(0.02)
                   + ma_couleur(<1,0,0,0>) ) #break
    #case (4)
       draw( 1, pL4, finesse(5) + courbe(0.02)
                   + ma_couleur(<1,1,0,0>) )
       draw( 1, pL4, finesse(0) + point(0.04)
                   + ma_couleur(<1,0,0,0>) )
    #break
    #end
#end
```

```
#macro pF_123() // pV222, pS23 et pL4
    #local pL2_0 = array[2] { <-0.5,-0.5,-0.5, 1.0>,
                              < 0.5, 0.0,-0.5, 1.0> }
    #local pL2_1 = array[2] { <-0.5, 0.0, 0.5, 1.0>,
                              < 0.5,-0.5, 0.5, 1.0> }
    #local pS22  = array[2] { pL2_0, pL2_1 }
    #local pS22_0 = pS22
    #local pS22_1 = pS22  pFtranslate( 2, pS22_1, <0, 0.5,0> )
                   pFrotate( 2, pS22_1, <0,-15,0> )
    #local pV222 = array[2] { pS22_0, pS22_1 }
    #local pS23 = pFdiagonalisation( 3, pV222 )
    #local pL4 = pFdiagonalisation( 2, pS23 )
    draw( 3, pV222, finesse(<1,3,3,0>) + enveloppe(LISSE)
                   + ma_couleur(<1,1,0,0.5>) )
    draw( 2, pS23, finesse(<4,4>) + surface(LISSE)
                   + ma_couleur(<0,1,0,0>) )
    draw( 1, pL4, finesse(5) + courbe( 0.02 )
                   + ma_couleur(<1,0,0,0>) )
#end

#macro pF_124( aspect ) // pS33 et pL5
    #local pL3_0 = array[3] { <-0.5,-0.5,-0.5, 1.0>,
             < 0.0, 0.5,-0.5, 1.0>, < 0.5,-0.5,-0.5, 1.0> }
    #local pL3_1 = array[3] { <-0.5, 0.5,-0.0, 1.0>,
             < 0.0, 0.5,-0.0, 1.0>, < 0.5,-0.5,-0.0, 1.0> }
    #local pL3_2 = array[3] { <-0.5, 0.5, 0.5, 1.0>,
             < 0.0,-0.5, 0.5, 1.0>, < 0.5, 0.5, 0.5, 1.0> }
    #local pS33  = array[3] { pL3_0, pL3_1, pL3_2 }
    #local pL5 = pFdiagonalisation( 2, pS33 )
    #switch (aspect)
    #case (0) draw( 2, pS33,
      finesse(<0,4>) + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (1) draw( 2, pS33,
      finesse(<1,4>) + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (2) draw( 2, pS33,
      finesse(<2,4>) + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (3) draw( 2, pS33,
      finesse(<3,4>) + courbe(0.01) + ma_couleur(<1,1,0,0>) ) #break
    #case (4) draw( 2, pS33,
      finesse(<4,4>) +surface(LISSE)+ma_couleur(<1,1,0,0.5>)) #break
    #end
    #switch (aspect)
    #case (0) draw( 1, pL5, finesse(0) + point(0.02)
                          + ma_couleur(<1,0,0,0>) ) #break
    #case (1) draw( 1, pL5, finesse(1) + point(0.02)
```

```
                          + ma_couleur(<1,0,0,0>) ) #break
    #case (2) draw( 1, pL5, finesse(2) + point(0.02)
                          + ma_couleur(<1,0,0,0>) ) #break
    #case (3) draw( 1, pL5, finesse(3) + point(0.02)
                          + ma_couleur(<1,0,0,0>) ) #break
    #case (4)
       draw( 1, pL5, finesse(5) + courbe(0.02)
                          + ma_couleur(<1,1,0,0>) )
       draw( 1, pL5, finesse(0) + point(0.04)
                          + ma_couleur(<1,0,0,0>) )
    #break
    #end
#end

// 13 : GENERALISATION, LES pFORMES

#macro pF_130() // pV333, pS35 et pL7
    #local pL3_0 = array[3] { <-0.5,-0.5,-0.5, 1.0>,
             < 0.0, 0.5,-0.5, 1.0>, < 0.5,-0.5,-0.5, 1.0> }
    #local pL3_1 = array[3] { <-0.5, 0.5,-0.0, 1.0>,
             < 0.0, 0.5,-0.0, 1.0>, < 0.5,-0.5,-0.0, 1.0> }
    #local pL3_2 = array[3] { <-0.5, 0.5, 0.5, 1.0>,
             < 0.0,-0.5, 0.5, 1.0>, < 0.5, 0.5, 0.5, 1.0> }
    #local pS33  = array[3] { pL3_0, pL3_1, pL3_2 }
    #local pS33_0 = pS33  pFtranslate( 2, pS33_0, <0,-0.25,0> )
    #local pS33_1 = pS33  pFscale( 2, pS33_1, <1.5,1,1.5> )
    #local pS33_2 = pS33  pFtranslate( 2, pS33_2, <0, 0.25,0> )
       #local pV333 = array[3] { pS33_0, pS33_1, pS33_2 }
       #local pS35 = pFdiagonalisation( 3, pV333 )
                          // 3 et (3+3-2)*(2-1)+1 = 5 -> 35
       //#local pS35 = pFstretch( 2, pS35, <-0.1,-0.1,0,1>,
                                            <1.1,1.1,0,1> )
       #local pL7 = pFdiagonalisation( 2, pS35 )
                          // (3+5-2)*(2-1)+1 = 7
//     #local pL7 = pFstretch(1, pL7,<-0.1,0,0,1>,<1.1,0,0,1>)
    draw( 3, pV333, finesse(<3,3,3,0>) + fibres(0.005)
                          + ma_couleur(<1,1,1,0.5>) )
//     draw( 3, pV333, finesse(<3,3,3,0>)
             + enveloppe(LISSE) + ma_couleur(<1,1,1,0.8>) )
       //draw( 3, pV333, finesse(<2,3,3,0>)
             + feuilles(LISSE) + ma_couleur(<1,1,1,0.8>) )
       //draw( 3, pV333, finesse(<3,3,3,0>)
       + fibres(0.005) + ma_couleur(<1,1,1,0>) )
    draw( 2, pS35, finesse(<2,4>) + surface(LISSE)
                                  + ma_couleur(<0,1,0,0.5>) )
```

```
    //draw( 2, pS35, finesse(<2,4>) + courbe(0.01)
                                + ma_couleur(<1,1,0,0.5>) )
    draw( 1, pL7, finesse(5) + courbe(0.02)
                                + ma_couleur(<1,0,0,0>) )
    draw( 2, pS35, finesse(<0,0>) + point(0.02)
                                + ma_couleur(<1,1,0,0>) )
    //draw( 1, pL7, finesse(0) + courbe(0.005)
                                + ma_couleur(<1,1,1,0>) )
    draw( 1, pL7, finesse(0) + point(0.03)
                                + ma_couleur(<1,0,0,0>) )
#end

// 21 : OPERATIONS FONDAMENTALES

#macro pF_212()     // ondulations sur un arc de cercle
       cylinder { <0,-0.5,0>, <0,0.5,0>, 0.5
                           pigment { color rgb <1,1,1,0.5> } }
       #local arc = quart_cercle_3( 0.5, 1 )
       pFrotate( 1, arc, <-90,0,0> )
       #local ond = pLup( arc, 2 )      //
       #local n = taille(ond);
       #local i=0; #while (i<n)
       pFtranslate( 0, ond[i], <0,0.5*sin(2*pi*i/(n-1)),0> )
       #local i=i+1; #end
       draw( 1, ond, finesse(0) + point(0.03)
                                + ma_couleur(<1,0,0>) )
       draw( 1, ond, finesse(5) + courbe(0.01)
                                + ma_couleur(<1,1,0>) )
#end

#macro pF_2131() // pL4 reparamétrisée
       #local pL4 = array[4]
{      <-1/2,-1/2,-1/2,1>, < 1/2,-1/2,-1/2,1>,
       < 1/2, 1/2,-1/2,1>, < 1/2, 1/2, 1/2,1> }
       draw(1,pL4,finesse(0)+point(0.05)+ma_couleur(<1,0,0>))
       draw(1,pL4, finesse(5)+courbe(0.01)+ma_couleur(<1,0,0>))
       #local pL4 = pFstretch(1,pL4,< 1/4,0,0,1>,<3/4,0,0,1>)
       draw( 1, pL4, finesse(0) + point(0.05)
                                + ma_couleur(<0,1,0,0.5>) )
       draw( 1, pL4, finesse(5) + courbe(0.02)
                                + ma_couleur(<0,1,0,0.5>) )
#end

#macro pF_2132() // pS35 reparamétrisée
    #local pL3_0 = array[3] { <-0.5,-0.5,-0.5, 1.0>,
```

```
             < 0.0, 0.5,-0.5, 1.0>, < 0.5,-0.5,-0.5, 1.0> }
    #local pL3_1 = array[3] { <-0.5, 0.5,-0.0, 1.0>,
             < 0.0, 0.5,-0.0, 1.0>, < 0.5,-0.5,-0.0, 1.0> }
    #local pL3_2 = array[3] { <-0.5, 0.5, 0.5, 1.0>,
             < 0.0,-0.5, 0.5, 1.0>, < 0.5, 0.5, 0.5, 1.0> }
    #local pS33  = array[3] { pL3_0, pL3_1, pL3_2 }
       draw( 2, pS33, finesse(<0,0>) + point(0.02)
                           + ma_couleur(<1,1,0,0.5>) )
       draw( 2, pS33, finesse(<3,3>) + surface(LISSE)
                           + ma_couleur(<1,1,0,0.5>) )
       #local pS33 = pFstretch( 2, pS33,
                           <1/8,1/8,0,1>, <7/8,7/8,0,1> )
       pFtranslate( 2, pS33, <0,0.01,0> )
       draw( 2, pS33, finesse(<0,0>) + point(0.02)
                           + ma_couleur(<1,1,1,0.5>) )
       draw( 2, pS33, finesse(<3,3>) + surface(LISSE)
                           + ma_couleur(<1,1,1,0.5>) )
#end

#macro pF_214( ) // pFgetSubForm, pFgetPoint, pFgetPijk
    #local pL3_0 = array[3] { <-0.5,-0.5,-0.5, 1.0>,
             < 0.0, 0.5,-0.5, 1.0>, < 0.5,-0.5,-0.5, 1.0> }
    #local pL3_1 = array[3] { <-0.5, 0.5,-0.0, 1.0>,
             < 0.0, 0.5,-0.0, 1.0>, < 0.5,-0.5,-0.0, 1.0> }
    #local pL3_2 = array[3] { <-0.5, 0.5, 0.5, 1.0>,
             < 0.0,-0.5, 0.5, 1.0>, < 0.5, 0.5, 0.5, 1.0> }
    #local pS33  = array[3] { pL3_0, pL3_1, pL3_2 }
    draw( 2, pS33, finesse(<4,4>) + surface(LISSE)
                           + ma_couleur(<1,1,0,0.5>) )
    draw( 2, pS33, finesse(<3,3>) + normales(0.01,-0.05)
                           + ma_couleur(<1,1,0,0.8>) )

    #local pL3 = pFgetSubForm( 2, pS33, 1/4 )
    draw( 1, pL3, finesse(5) + courbe(0.02)
                           + ma_couleur(<1/2,1/2,1/2,0>) )

    #local p = pFgetPoint( 2, pS33, <1/4,1/4,0,1> )
    draw( 0, p, point(0.05) + ma_couleur(<1,0,0,0.5>) )

    pSdrawPijk( pS33, <4/6,1/3,0,1>, 0.3, 0.01 )
    pLdrawFrenet( pL3, 1/4, 0.3, 0.01 )
#end
```

```
// 22 : IMMERSIONS

// 221 : INTERPOLATION

#macro pF_2211() // pL5 interpolante
       #local pL5 = array[5] {
       <-1/2,-1/2,-1/2,1>, < 1/2,-1/2,-1/2,1>,
       < 1/2, 1/2,-1/2,1>, < 1/2, 1/2, 1/2,1>, < 1/2,-1/2, 1/2,1> }
       draw( 1, pL5, finesse(0) + point(0.05)+ma_couleur(<1,0,0>) )
       #local pL5 = pLinterpolante( pL5 )
       #local pL5 =pFstretch(1,pL5,<-0.03,0,0,1>, <1.03,0,0,1>)
       draw( 1, pL5, finesse(5) + courbe(0.01)
                                +ma_couleur(<1,1,0>) )
#end

#macro pF_2212() // pS35 interpolante
       #local pL3 = array[3] { <-1/2, 0,-1/2,1>,
                           < 0, 0,-1/2,1>, < 1/2, 0,-1/2,1> }
       #local pL3_0 = pL3 pFtranslate( 1, pL3_0, <0,0,0/4> )
       pFtranslate( 0, pL3_0[1], <0,0,-1/4> )
       #local pL3_1 = pL3 pFtranslate( 1, pL3_1, <0,0,1/4> )
       pFtranslate( 0, pL3_1[1], <0,-1/4,0> )
       #local pL3_2 = pL3 pFtranslate( 1, pL3_2, <0,0,2/4> )
       pFtranslate( 0, pL3_2[2], < 1/8,0,0> )
       #local pL3_3 = pL3 pFtranslate( 1, pL3_3, <0,0,3/4> )
       pFtranslate( 0, pL3_3[1], <0, 1/4,0> )
       #local pL3_4 = pL3 pFtranslate( 1, pL3_4, <0,0,4/4> )
       pFtranslate( 0, pL3_4[1], <0,0, 1/4> )
       #local pS35 = array[5] {pL3_0,pL3_1,pL3_2,pL3_3,pL3_4}
       draw( 2, pS35, finesse(<0,0>) + point(0.04)
                                + ma_couleur(<1,0,0>) )
       #local pS35 = pSinterpolante( pS35 )
       #local pS35 = pFstretch( 2, pS35,
                           <-0.06,-0.06,0,1>, <1.06,1.06,0,1> )
       draw( 2, pS35, finesse(<4,4>) + surface(LISSE)
                                + ma_texture(MARBRE) )
       #local pL7 = pFdiagonalisation( 2, pS35 )
       #local pL7 = pFstretch(1,pL7,<-0.3,0,0,1>,<1.01,0,0,1>)
       draw( 1, pL7, finesse(6)
                           +courbe(0.01)+ma_couleur(<1,1,0>) )
#end

#macro pF_2235() // pL3 immergée dans une PS33
    #local pL3_0 = array[3] { <-0.5,-0.5,-0.5, 1.0>,
              < 0.0, 0.5,-0.5, 1.0>, < 0.5,-0.5,-0.5, 1.0> }
```

```
#local pL3_1 = array[3] { <-0.5, 0.5,-0.0, 1.0>,
         < 0.0, 0.5,-0.0, 1.0>, < 0.5,-0.5,-0.0, 1.0> }
#local pL3_2 = array[3] { <-0.5, 0.5, 0.5, 1.0>,
         < 0.0,-0.5, 0.5, 1.0>, < 0.5, 0.5, 0.5, 1.0> }
#local pS33  = array[3] { pL3_0, pL3_1, pL3_2 }

   #local p0 = <1/4,1/8,0,1> ;
   #local p1 = <1/8,3/4,0,1> ;
   #local p2 = <7/8,2/4,0,1> ;
   #local q0 = (p0+p1)/2 ;
   #local q1 = (p1+p2)/2 ;
   #local r0 = (q0+q1)/2 ;

   #local pL2_0 = array[2] { p0, p1 }
   #local pL2_1 = array[2] { p1, p2 }
   #local pL2_2 = array[2] { (p0+p1)/2, (p1+p2)/2 }
   #local pL3 = array[3] { p0, p1, p2 }

   #local ipL2_0 = courbe_in_surface( pL2_0, pS33 )
   #local ipL2_1 = courbe_in_surface( pL2_1, pS33 )
   #local ipL2_2 = courbe_in_surface( pL2_2, pS33 )
   #local ipL3 = courbe_in_surface( pL3, pS33 )

   #local ip = pFgetPoint( 2, pS33, p0 )
   draw( 0, ip, point(0.04) + ma_couleur(<1,0,0,0>) )
   #local ip = pFgetPoint( 2, pS33, p1 )
   draw( 0, ip, point(0.04) + ma_couleur(<1,0,0,0>) )
   #local ip = pFgetPoint( 2, pS33, p2 )
   draw( 0, ip, point(0.04) + ma_couleur(<1,0,0,0>) )
   #local ip = pFgetPoint( 2, pS33, q0 )
   draw( 0, ip, point(0.04) + ma_couleur(<0,1,0,0>) )
   #local ip = pFgetPoint( 2, pS33, q1 )
   draw( 0, ip, point(0.04) + ma_couleur(<0,1,0,0>) )
   #local ip = pFgetPoint( 2, pS33, r0 )
   draw( 0, ip, point(0.05) + ma_couleur(<0,0,1,0>) )

draw( 2, pS33, finesse(<4,4>) + surface(LISSE)
                      + ma_couleur(<1,1,0,0.5>) )
   draw( 1, ipL2_0, finesse(5) + courbe(0.01)
                            + ma_couleur(<1,1,1,0>) )
   draw( 1, ipL2_1, finesse(5) + courbe(0.01)
                            + ma_couleur(<1,1,1,0>) )
   draw( 1, ipL2_2, finesse(5) + courbe(0.01)
                            + ma_couleur(<1,1,0,0>) )
   draw( 1, ipL3, finesse(5) + courbe(0.02)
```

```
                                    + ma_couleur(<0,1,0,0>) )

#end

#macro pF_2236() // pL4 immergée dans une PS33
    #local pL3_0 = array[3] { <-0.5,-0.5,-0.5, 1.0>,
             < 0.0, 0.5,-0.5, 1.0>, < 0.5,-0.5,-0.5,1.0> }
    #local pL3_1 = array[3] { <-0.5, 0.5,-0.0, 1.0>,
             < 0.0, 0.5,-0.0, 1.0>, < 0.5,-0.5,-0.0, 1.0> }
    #local pL3_2 = array[3] { <-0.5, 0.5, 0.5, 1.0>,
             < 0.0,-0.5, 0.5, 1.0>, < 0.5, 0.5, 0.5, 1.0> }
    #local pS33  = array[3] { pL3_0, pL3_1, pL3_2 }
       #local p0 = <1/4,1/8,0,1> ;
       #local p1 = <1/8,3/4,0,1> ;
       #local p2 = <7/8,2/4,0,1> ;
       #local p3 = <4/8,1/4,0,1> ;

    draw( 2, pS33,
       finesse(<4,4>) + surface(LISSE) + ma_couleur(<1,1,0,0.5>) )

    #local ipL2 = courbe_in_surface(array[2] {p0, p1}, pS33 )
       draw( 1, ipL2,
       finesse(5) + courbe(0.01) + ma_couleur(<1,1,1,0>) )
    #local ipL2 = courbe_in_surface( array[2] {p1, p2}, pS33 )
       draw( 1, ipL2,
       finesse(5) + courbe(0.01) + ma_couleur(<1,1,1,0>) )
    #local ipL2 = courbe_in_surface( array[2] {p2, p3},pS33 )
       draw( 1, ipL2,
       finesse(5) + courbe(0.01) + ma_couleur(<1,1,1,0>) )

    #local ipL4 = courbe_in_surface(array[4]{p0,p1,p2,p3},pS33)
       draw( 1, ipL4,
       finesse(5) + courbe(0.02) + ma_couleur(<0,1,0,0.5>) )
#end

// 23 : INTERFACE

// 231 : TRANSFORMATIONS

#macro pF_2311()    // pFrotate sur une pL4
       #local pL4 = array[4] {
       <-1/2,-1/2,-1/2,1>, < 1/2,-1/2,-1/2,1>,
       < 1/2, 1/2,-1/2,1>, < 1/2, 1/2, 1/2,1> }
       #local N = 24;
       #local i=0; #while (i<N)
```

```
              draw( 1, pL4,
              finesse(6) + courbe(0.03) + ma_couleur(<1,1,0,0>) )
              pFrotate( 1, pL4, <0,360/(N-1),0> )
       #local i=i+1; #end
#end

#macro pF_2312()    // pFtranslate sur une pL4
       #local pL4 = array[4] {
       <-1/2,-1/2,-1/2,1>, < 1/2,-1/2,-1/2,1>,
       < 1/2, 1/2,-1/2,1>, < 1/2, 1/2, 1/2,1> }
       pFtranslate( 1, pL4, <0,-1/2,0> )
       #local N = 24;
       #local i=0; #while (i<N)
              draw( 1, pL4,
              finesse(6) + courbe(0.03) + ma_couleur(<1,1,0,0>) )
              pFtranslate( 1, pL4, <0,1/(N-1),0> )
       #local i=i+1; #end
#end

#macro pF_2313()    // pFscale sur une pL4
       #local pL4 = array[4] {
       <-1/2,-1/2,-1/2,1>, < 1/2,-1/2,-1/2,1>,
       < 1/2, 1/2,-1/2,1>, < 1/2, 1/2, 1/2,1> }
       #local N = 24;
       #local i=0; #while (i<N)
              draw( 1, pL4,
              finesse(6) + courbe(0.03) + ma_couleur(<1,1,0,0>) )
              pFscale( 1, pL4, <0.7,1,0.7> )
       #local i=i+1; #end
#end


// 3 : COMPOSITIONS

// 31 : FORMES RATIONNELLES

#macro pF_3111()    // les 4 coniques comme projections de pL3
       #local oo = <0,0.5,0,1> ;
       #local p0 = <0.5,0,0,1> ;
       #local p1 = <0.5,0,0.5,1> ;
       #local p2 = <0,0,0.5,1> ;
       #local k = 1/sqrt(2);
       #local eps = 0.01 ;

#macro conic( uu )
```

```
      union
      {
#local pL3 = array[3] { p0, k/uu*((1-uu)*oo+uu*p1), p2 }
draw( 1, pL3, finesse(0) + point(0.03) + ma_couleur(<1,0,0>))
#local pL3 = pFstretch( 1, pL3,<-k+eps,0,0,1>,<1+k-eps,0,0,1>)
draw( 1, pL3, finesse(6) + courbe(0.02) + ma_couleur(<1,1,0>))
      }
#end

union
{     cone { <0,0.5,0>, 0, <0,-0.5,0>, 1
                  pigment { color rgbt <1,1,1,0.5> } }
      draw( 0, oo, point(0.03) + ma_couleur(<1,1,1>) )
      #local pL2 = array[2] { oo, p1 }
      #local pL2 = pFstretch(1, pL2,<-k,0,0,1>,<1+k,0,0,1>)
      draw( 1, pL2,
            finesse(0)+courbe(0.01)+ma_couleur(<1,1,1>) )
            conic( 0.500 )      // hyperbole verticale
            conic( 0.707 )      // cercle
            conic( 1.000 )      // parabole
            conic( 1.500 )      // ellipse
            rotate <0,45,0>
      }
#end

#macro pF_31121()   // pL3, pL4 et pL5 -> arcs de cercle
      cylinder { <0,0,-1>, <0,0,1>, 0.5
                  pigment { color rgbt <1,1,1,0.5> } }
      #local pL3 = quart_cercle_3( 0.5, 1 )
      pFtranslate( 1, pL3, <0,0,-0.25> )
      #local pL4 = demi_cercle_4( 0.5 )
      #local pL5 = demi_cercle_5( 0.5 )
      pFtranslate( 1, pL5, <0, 0, 0.25> )
      draw( 1, pL3,
      finesse(0) + point(0.03) + ma_couleur(<1,0,0>) )
      draw( 1, pL3,
      finesse(0) + courbe(0.005) + ma_couleur(<1,1,1>) )
      draw( 1, pL3,
      finesse(6) + courbe(0.01) + ma_couleur(<1,1,0>) )
      draw( 1, pL4,
      finesse(0) + point(0.03) + ma_couleur(<1,0,0>) )
      draw( 1, pL4,
      finesse(0) + courbe(0.005) + ma_couleur(<1,1,1>) )
      draw( 1, pL4,
      finesse(6) + courbe(0.01) + ma_couleur(<1,1,0>) )
```

```
       draw( 1, pL5,
       finesse(0) + point(0.03) + ma_couleur(<1,0,0>) )
       draw( 1, pL5,
       finesse(0) + courbe(0.005) + ma_couleur(<1,1,1>) )
       draw( 1, pL5,
       finesse(6) + courbe(0.01) + ma_couleur(<1,1,0>) )
#end

#macro pF_31122( choix )
// pL3, pL4 et pL5, projections R4->R3 et arcs de cercle
       cone { <0,0,-0.5>, 0, <0,0,0.5>, 1.0
                    pigment { color rgbt <1,1,1,0.5> } }
       #local k = 1/sqrt(2);
       #local r = 0.5;
       #local oo = < 0,0,-0.5,1 >;
       #switch (choix)
       #case (1)
           #local p0 = < r, 0, 0, 1 > ;
           #local p1 = < r, r, 0, 1 > ;
           #local p2 = < 0, r, 0, 1 > ;
           #local uu = k;
           #local pL = array[3] {p0, k/uu*((1-uu)*oo+uu*p1),p2}
           #local arc = array[3] { p0, p1*k, p2 }
       #break
       #case (2)
           #local p0 = <  r, 0,  0, 1 > ;
             // k/uu*((1-uu)*oo+uu*p1) = (2*oo+p1)/3
           #local p1 = <  r, 2*r,0, 1 > ;
             // k/uu*oo -k*oo + k*p1 = 2/3*oo + p1/3
           #local p2 = < -r, 2*r,0, 1 > ;            //
           #local p3 = < -r, 0,  0, 1 > ;
           //#local uu = ??; k/uu*((1-uu)*oo+uu*p1) dosn't work
          #local pL = array[4] {p0,(2*oo+p1)/3,(2*oo+p2)/3,p3}
           #local arc = array[4] { p0, p1/3, p2/3, p3 }
       #break
       #case (3)
           #local p0 = <  r, 0,      0, 1 > ;
           #local p1 = <  r, r,      0, 1 > ;
           #local p2 = <  0, 3/2*r,  0, 1 > ;
           #local p3 = < -r, r,      0, 1 > ;
           #local p4 = < -r, 0,      0, 1 > ;
           #local uu = 1/2;
           #local pL = array[5] { p0,   k/uu*((1-uu)*oo+uu*p1),
                                        k/uu*((1-uu)*oo+uu*p2),
                                        k/uu*((1-uu)*oo+uu*p3),
```

```
                                p4 }
            #local arc = array[5] { p0, p1*k, p2*2/3, p3*k, p4 }
      #break
      #end

      union
      {
      #local h = 0.0;
      #switch (choix)
            #case (1)     #local h = 1.0;     #break
            #case (2)     #local h = 0.6;     #break
            #case (3)     #local h = 0.2;     #break
      #end
      draw( 1, pL,
      finesse(0) + point(0.04) + ma_couleur(<0,1,0,0.5>) )
      draw( 1, pL, finesse(0) + courbe(0.01)+ma_couleur(<1,1,1>) )
      #local pL = pFstretch( 1, pL, <-h,0,0,1>, <1+h,0,0,1> )
      draw( 1, pL,
      finesse(6) + courbe(0.02) + ma_couleur(<0,1,0>) )
      draw( 1, arc,
      finesse(0) + point(0.04) + ma_couleur(<1,0,0,0.05>) )
      draw( 1, arc,
      finesse(0) + courbe(0.01)+ma_couleur(<1,1,1>))
      #local arc = pFstretch( 1, arc, <-h,0,0,1>,<1+h,0,0,1>)
      draw( 1, arc,
      finesse(6) + courbe(0.02)+ma_couleur(<1,0,0>))
      rotate <0,0,0>
      }
#end

#macro pF_3113()    // 3  approches du cercle complet
      #local k = 1/sqrt(2) ;
      #local eps = 0.01 ;
//    cylinder { <0,0,-1>, <0,0,1>, 0.5
                        pigment { color rgbt <1,1,1,0.5> } }
      #local pL3 = quart_cercle_3( 0.5, 1 )
      pFtranslate( 1, pL3, <0,0,-1> )
      #local pL4 = demi_cercle_4( 0.5 )
      #local pL5 = demi_cercle_5( 0.5 )
      pFtranslate( 1, pL5, <0,0, 1> )
      #local pL3 = pFstretch( 1,pL3,<-5,0,0,1>,<6,0,0,1>)
      draw( 1,pL3,finesse(0)+point(0.03)+ma_couleur(<1,0,0>))
      draw( 1,pL3,finesse(0)+courbe(0.01)+ma_couleur(<1,1,1>))
      draw( 1,pL3,finesse(6)+point(0.02)+ma_couleur(<1,1,0>))
      #local pL4 = pFstretch( 1, pL4, <-1,0,0,1>, <2,0,0,1>)
```

```
      draw( 1,pL4,finesse(0)+point(0.03)+ma_couleur(<1,0,0>))
      draw(1,pL4,finesse(0)+courbe(0.01)+ma_couleur(<1,1,1>))
      draw( 1,pL4,finesse(6)+point(0.02)+ma_couleur(<1,1,0>))
      #local pL5 = pFstretch( 1, pL5,
                          <-k+eps,0,0,1>, <1+k-eps,0,0,1> )
      draw( 1,pL5,finesse(0)+point(0.03)+ma_couleur(<1,0,0>))
      draw( 1,pL5,finesse(0)+courbe(0.01)+ma_couleu(<1,1,1>))
      draw( 1,pL5,finesse(6)+point(0.02)+ma_couleur(<1,1,0>))
#end

#macro pF_3114( uu )       // cercle immergé dans une pS33
    #local pL3_0 = array[3] { <-0.5,-0.5,-0.5, 1.0>,
             < 0.0, 0.5,-0.5, 1.0>, < 0.5,-0.5,-0.5, 1.0> }
    #local pL3_1 = array[3] { <-0.5, 0.5,-0.0, 1.0>,
             < 0.0, 0.5,-0.0, 1.0>, < 0.5,-0.5,-0.0, 1.0> }
    #local pL3_2 = array[3] { <-0.5, 0.5, 0.5, 1.0>,
             < 0.0,-0.5, 0.5, 1.0>, < 0.5, 0.5, 0.5, 1.0> }
    #local pS33  = array[3] { pL3_0, pL3_1, pL3_2 }
    draw( 2, pS33, finesse(<3,3>) + surface(LISSE)
                                + ma_couleur(<1,1,0,0.5>) )
      #local pL5 = demi_cercle_5( 0.25 )
    pFtranslate( 1, pL5, <0.5,0.5,0> )
    #local pL5 = pFstretch( 1, pL5,<0-uu,0,0,1>,<1+uu,0,0,1>)

/*
      #local i=0; #while (i<5)
            #local ip = pFgetPoint( 2, pS33, pL5[i] )
            draw( 0, ip, point(0.03) + ma_couleur(<1,0,0>) )
      #local i=i+1; #end

      #local i=0; #while (i<4)
            #local pL2 = array[2]{ pL5[i], pL5[i+1] }
            #local ipL2 = courbe_in_surface( pL2, pS33 )
            draw( 1, ipL2, finesse(6) + courbe(0.01)
                                + ma_couleur(<1,1,1>) )
      #local i=i+1; #end
*/
      #local ipL5 = courbe_in_surface( pL5, pS33 )
      draw( 1, ipL5, finesse(6) + courbe(0.02)
                                + ma_couleur(<0,1,0>) )
#end

#macro pF_312()     // le cylindre, le tore et la sphère
      #local R = 0.5;
      #local H = 0.5;
```

```
      #local R1 = 0.5;
      #local R2 = 0.25;
// un quart de cylindre de rayon R et de hauteur H:
      #local k = sqrt(2)/2;
      #local pCylindre = array[2]
      {     array[3] { < R, 0, 0, 1 >,
                       < R, R, 0, 1 >*k, < 0, R, 0, 1 > },
       array[3] { < R, 0, H, 1 >,
                       < R, R, H, 1 >*k, < 0, R, H, 1 > }
      }
// un seizième de tore de rayons R1 et R2:
      #local R12 = R1+R2;
      #local R2 = abs(R2);
      #local pTore = array[3]
      {     array[3] { < 0,   0, -R12, 1 >,
            < 0,  R2, -R12, 1 >*k, < 0,  R2,  -R1, 1 > },
       array[3] { < R12,  0, -R12, 1 >*k,
            < R12, R2, -R12, 1 >*k*k, <  R1, R2, -R1,  1 >*k }
       array[3] { < R12,  0,  0, 1 >,
            < R12, R2,  0, 1 >*k, <  R1, R2,  0, 1 > }
      }
// une huitième de sphère de rayon R:
      #local pSphere = array[3]
      {     array[3] { < R, 0, 0, 1 >,
                  < R, R, 0, 1 >*k, < 0, R, 0, 1 > },
       array[3] { < R, 0, -R, 1 >*k,
                  < R, R, -R, 1 >*k*k, < 0, R,  0, 1 >*k },
       array[3] { < 0, 0, -R, 1 >, < 0, R, -R, 1 >*k,
                  < 0, R,  0, 1 > }
      }
      pFtranslate( 2, pCylindre, <0,0,0.05> )
      pFtranslate( 2, pTore, <0,-0.3,0> )
      pFtranslate( 2, pSphere, <0,0,0> )
      draw( 2, pCylindre, finesse(<0,0>) + point(0.03)
                               + ma_couleur(<1,0,0,0>) )
      draw( 2, pCylindre, finesse(<3,3>) + surface(LISSE)
                               + ma_couleur(<1,1,0,0.5>))
      draw( 2, pTore, finesse(<0,0>) + point(0.03)
                               + ma_couleur(<1,0,0,0>))
      draw( 2, pTore, finesse(<3,3>) + surface(LISSE)
                               + ma_couleur(<0,1,1,0.5>))
      draw( 2, pSphere, finesse(<0,0>) + point(0.03)
                               + ma_couleur(<1,0,0,0>))
      draw( 2, pSphere, finesse(<3,3>) + surface(LISSE)
                               + ma_couleur(<1,0,0,0.5>))
```

```
      #local diag = pFdiagonalisation( 2, pCylindre )
      #local diag = pFstretch(1,diag,<-0.2,0,0,1>,<1.2,0,0,1>)
      draw( 1, diag, finesse(6) + courbe(0.01)
                              + ma_couleur(<1,1,1>) )
      #local diag = pFdiagonalisation( 2, pTore )
      #local diag = pFstretch(1,diag,<-0.2,0,0,1>,<1.2,0,0,1>)
      draw( 1, diag, finesse(6) + courbe(0.01)
                              + ma_couleur(<1,1,1>) )
      #local diag = pFdiagonalisation( 2, pSphere )
      #local diag = pFstretch(1,diag,<-0.2,0,0,1>,<1.2,0,0,1>)
      draw( 1, diag, finesse(6) + courbe(0.01)
                              + ma_couleur(<1,1,1>) )
#end

#macro pF_3131(N)   // diagonales // immergées dans un tore
      torus { 0.5, 0.25 pigment { color rgbt <1,1,1,0.5> } }
      #local c2 = demi_cercle_5( 1 )   // chemin dans xOz
      // demi-diagonale 1
      #local c1 = demi_cercle_5( 0.25 ) // section dans xOy
      pFtranslate( 1, c1, <0.5,0,0> )
      #local pS33 = cross( c1, c2 )
      #local ipL2_1 = pFdiagonalisation( 2, pS33 )
      // demi-diagonale 2
      #local c1 = demi_cercle_5( 0.25 ) // section dans xOy
      pFscale( 1, c1, <-1,-1,1> )
      pFtranslate( 1, c1, <0.5,0,0> )
      pFrotate( 1, c1, <0,180,0> )
      #local pS33 = cross( c1, c2 )
      #local ipL2_2 = pFdiagonalisation( 2, pS33 )

      #local i=0; #while(i<N)
            union
            {     draw( 1, ipL2_1, finesse(5) + courbe(0.01)
                                    + ma_couleur(<1,0,0>) )
                  draw( 1, ipL2_2, finesse(5) + courbe(0.01)
                                    + ma_couleur(<1,0,0>) )
                  rotate <0,360*i/(N-1),0>
            }
      #local i=i+1; #end
#end

#macro pF_3132(N)   // faisceau de droites immergées dans un tore
      torus { 0.5, 0.25 pigment { color rgbt <1,1,1,0.5> } }
      #local c1 = demi_cercle_4( 0.25 )
      pFtranslate( 1, c1, <0.5,0,0> )  // section dans xOy
```

```
      #local c2 = demi_cercle_4( 1 )   // chemin dans xOz
      #local pS33 = cross( c1, c2 )
      pFrotate( 2, pS33, <0,-90,0> )
      #local i=0; #while(i<N)
            #local aa = 2*pi*i/N ;
            #local pL2 =
                  array[2]{<0,0,0,1>,<cos(aa),sin(aa),0,1>}
            pFtranslate( 1, pL2, <0.5,0.5,0> )
            #local ipL2 = courbe_in_surface( pL2, pS33 )
            draw( 1, ipL2, finesse(5) + courbe(0.005)
                                    + ma_couleur(<1,1,1>) )
      #local i=i+1; #end
#end

#macro pF_3133( N ) // cercles concentriques immergés dans un tore
//    torus { 0.5, 0.249 pigment { color rgbt <1,1,1,0.5> } }
      #local c1 = demi_cercle_4( 0.25 )
      pFtranslate( 1, c1, <0.5,0,0> )  // section dans xOy
      #local c2 = demi_cercle_4( 1 )   // chemin dans xOz
      #local pS33 = cross( c1, c2 )
      pFrotate( 2, pS33, <0,-90,0> )
//    draw( 2, pS33, finesse(<3,3>) + point(0.005)
                                    + ma_couleur(<1,1,1,0>) )
      #local i=0; #while (i<N)
            #local uu = i/(N-1);
            #local pL5 = demi_cercle_4( (1-uu)*0.05 + uu*1.0 )
            #local pL5_up = pL5
            #local pL5_down = pL5
      pFrotate( 1, pL5_up, <0,0,180> )
      pFtranslate( 1, pL5_up, <0.5,0.5,0> )
      pFtranslate( 1, pL5_down, <0.5,0.5,0> )
            #local ipL5_up = courbe_in_surface( pL5_up, pS33 )
            #local ipL5_down =courbe_in_surface(pL5_down,pS33)
            draw( 1, ipL5_up, finesse(4) + courbe(0.01)
                                    + ma_couleur(<1,uu,0>) )
            draw( 1, ipL5_down, finesse(4) + courbe(0.01)
                                    + ma_couleur(<1,uu,0>) )
      #local i=i+1; #end
#end

#macro pF_3134( N, R )//cercle et ses rayons (R<2) dans un tore
      torus { 0.5, 0.25 pigment { color rgbt <1,1,1,0.5> } }
      #local c1 = demi_cercle_4( 0.25 )
      pFtranslate( 1, c1, <0.5,0,0> )  // section dans xOy
      #local c2 = demi_cercle_4( 1 )   // chemin dans xOz
```

```
      #local pS33 = cross( c1, c2 )
      pFrotate( 2, pS33, <0,-90,0> )
   // draw( 2, pS33,
      finesse(<3,3>) + point(0.005) + ma_couleur(<1,1,1,0>) )
      #local pL5 = demi_cercle_4( R )
      #local pL5_up = pL5
      #local pL5_down = pL5
      pFrotate( 1, pL5_up, <0,0,180> )
      pFtranslate( 1, pL5_up, <0.5,0.5,0> )
      pFtranslate( 1, pL5_down, <0.5,0.5,0> )
      #local ipL5_up = courbe_in_surface( pL5_up, pS33 )
      #local ipL5_down = courbe_in_surface( pL5_down, pS33 )
      draw( 1, ipL5_up, finesse(6) + courbe(0.02)
                                   + ma_couleur(<1,0,0>) )
      draw( 1, ipL5_down, finesse(6) + courbe(0.02)
                                   + ma_couleur(<1,0,0>) )

      #local i=0; #while(i<N)
            #local aa = 2*pi*i/N ;
            #local pL2 = array[2] {<0,0,0,1>,
                                  <R*cos(aa),R*sin(aa),0,1> }
            pFtranslate( 1, pL2, <0.5,0.5,0> )
            #local ipL2 = courbe_in_surface( pL2, pS33 )
            draw( 1, ipL2, finesse(5) + courbe(0.01)
                                   + ma_couleur(<1,1,0>) )
      #local i=i+1; #end

#end

#macro pF_3221()    // un vase          ( no box + pers 2/3 )
      #local k = 1/sqrt(2);
      #local c1 = array[7] {    <0.01,-0.50,0,1>,
                                <0.10,-0.50,0,1>,
                                <0.50,-0.50,0,1>,
                                <0.50,-0.25,0,1>,
                                <0.10, 0.00,0,1>,
                                <0.10, 0.50,0,1>,
                                <0.15, 0.50,0,1>
                           }    // section dans xOy
      #local c2 = demi_cercle_5( 1 )   // chemin dans xOz
      #local c2 = pFstretch( 1, c2, <-k,0,0,1>, <1+k,0,0,1> )
      #local pS33 = cross( c1, c2 )
      pFtranslate( 0, pS33[6][2], < 0,-0.5,0> )
                                // poignée et bec verseur
      pFrotate( 1, c1, <0,180,0> )
```

```
      draw( 1, c1, finesse(6) + courbe(0.01)
                              + ma_couleur(<1,0,0,0>) )
      pFrotate( 2, pS33, <0,-90,0> )
      draw( 2, pS33, finesse(<4,4>) + surface(LISSE)
                              + ma_texture(MARBRE) )
      object { plane { <0,1,0>, -1/2 } une_texture( GRANIT ) }
 #end

#macro pF_3321()    // surface minimale periodique COONS
      #local p0 = <-1/2,-1/2,-1/2,1> + <0,1/2,-1/2> ;
      #local p1 = <-1/2, 1/2,-1/2,1> + <0,1/2,-1/2> ;
      #local p2 = <-1/2, 1/2, 1/2,1> + <0,1/2,-1/2> ;
      #local p3 = <-1/2,-1/2,-0/2,1> + <0,1/2,-1/2> ;
      #local p4 = <-0/2,-0/2, 1/2,1> + <0,1/2,-1/2> ;
      #local p5 = <-1/2,-0/2, 1/2,1> + <0,1/2,-1/2> ;
      #local p6 = <-0/2,-1/2,-0/2,1> + <0,1/2,-1/2> ;
      #local p7 = <-0/2,-0/2,-0/2,1> + <0,1/2,-1/2> ;
      #local p8 = <-0/2,-0/2, 1/2,1> + <0,1/2,-1/2> ;
      #local L1 = array[6] { p0, p1, p1, p1, p1, p2 }
// approche d'un coin carré
      #local L2 = array[3] { p6, p7/sqrt(2), p8 }
// arc de cercle
      #local L3 = array[3] { p0, p3, p6 }
// arc de parabole
      #local L4 = array[3] { p2, p5, p8 }
// arc de parabole
      #local coons = creer_coons( L3, L4, L1, L2 )
            #local i=0; #while (i<2)
            #local j=0; #while (j<4)
            union
            {     draw(2,coons,finesse(<3,2>)+surface(LISSE)
                              + ma_texture(GRANIT) )
                  draw( 1, L1, finesse(4) + point(0.02)
                              + ma_couleur( < 0,0,1/2 > ) )
                  draw( 1, L2, finesse(4) + point(0.02)
                              + ma_couleur( < 0,0,1/2 > ) )
                  draw( 1, L3, finesse(4) + point(0.02)
                              + ma_couleur( < 1/2,0,0 > ) )
                  draw( 1, L4, finesse(4) + point(0.02)
                              + ma_couleur( < 1/2,0,0 > ) )
                  rotate <90*j,0,0>
                  #if (i=1) scale <-1,1,1> #end
            }
            #local j=j+1; #end
            #local i=i+1; #end
```

```
#end

#macro pF_3322()    // coquillage COONS
      #local L1 = array[3] { <0.0,0.0,-0.5,1>,
            <0.0,0.25,-0.5,1>/sqrt(2), <0.0,0.25,-0.1,1> }
      #local L2 = array[3] { <0.5,0.0, 0.0,1>,
            <0.5,0.25, 0.0,1>/sqrt(2), <0.1,0.25, 0.0,1> }
      #local L3 = array[3] { <0.0,0.25,-0.1,1>,
            <0.1,0.25,-0.1,1>/sqrt(2), <0.1,0.25,0.0,1> }
      #local L4 = array[3] { <0.0,0.0,-0.5,1>,
            <0.5,0.0,-0.5,1>/sqrt(2), <0.5,0.0,0.0,1> }
      #local L4 = pLup( L4, 6 ) //4,6
      #local n = taille(L4);
      #local i=0; #while (i<n)
      pFtranslate( 0, L4[i], <0,0.25*sin(4*pi*i/(n-1)),0> )
      #local i=i+1; #end
      #local coons = creer_coons( L1, L2, L3, L4 )
      #local j=0; #while (j<2)
      #local i=0; #while (i<4)
      union
      {     draw( 2, coons, finesse(<3,3>) + surface(LISSE)
                              + ma_texture(MARBRE) )
            #local diag = pFdiagonalisation( 2, coons )
            draw( 1, diag, finesse(4) + courbe(0.005)
                              + ma_couleur( < 1,1,1 > ) )
            //draw( 1, L1, finesse(4) + point(0.005)
                              + ma_couleur( < 0,0,1/2 > ) )
            //draw( 1, L2, finesse(4) + point(0.005)
                              + ma_couleur( < 0,0,1/2 > ) )
            draw( 1, L3, finesse(4) + courbe(0.005)
                              + ma_couleur( < 1/2,0,0 > ) )
            draw( 1, L4, finesse(4) + courbe(0.005)
                              + ma_couleur( < 1/2,0,0 > ) )
            rotate <0,90*i,0>
            #if (j=1)
                  scale <-1,-1,1>
                  translate <0,-0.05,0>
            #end
      }
      #local i=i+1; #end
      #local j=j+1; #end
#end
```

```
#macro pF_3323( k ) // rideau COONS
                    // (no box+PERS 5/6, k=1,2,3,4,..)
       #local L1 = creer_ligne( 4*k+1, 1 )
       #local L2 = L1
       #local n = taille(L2);
       #local i=0; #while (i<n)
              pFtranslate( 0, L2[i],
                           <0,0,0.125*sin(2*k*pi*i/(n-1))> )
       #local i=i+1; #end
       pFtranslate( 1, L1, <0,0.5,0> )
       pFtranslate( 1, L2, <0,-0.5,0> )
       #local L3 = array[3] {<-0.5,0.5,0,1>,
                    <-0.5,0,-0.125,1>, <-0.5,-0.5,0,1> }
       #local L4 = array[3] {< 0.5,0.5,0,1>,
                    < 0.5,0, 0.125,1>, <0.5,-0.5,0,1> }
       #local L2 = pLinterpolante( L2 )
       #local coons = creer_coons( L1, L2, L3, L4 )
       #local diag = pFdiagonalisation( 2, coons )
       union {
       draw( 1, L2, finesse(6) + point(0.01)
                                  + ma_couleur(<1,1,0,0>) )
       draw( 2, coons, finesse(<3,3>) + surface(LISSE)
                                  + ma_texture(MARBRE) )
       draw( 1, diag, finesse(6) + courbe(0.01)
                                  + ma_couleur( <0,1,0> ) )
       translate <0,0,-0.25>
       }
       union {
       draw( 1, L2, finesse(6) + point(0.01)
                                  + ma_couleur(<1,1,0,0>) )
       draw( 2, coons, finesse(<3,3>) + surface(LISSE)
                                  + ma_texture(GRANIT) )
       draw( 1, diag, finesse(6) + courbe(0.01)
                                  + ma_couleur( <0,1,0> ) )
       translate <0,0,0.25>}
#end

#macro pF_3421( k ) // spline interpolante quadrique
                    // (no box + AXO 7/8, k=1,2,3,4,...)
       #local curv = creer_ligne( 4*k+1, 1.5 )
       #local n = taille(curv);
       #local i=0; #while (i<n)
              pFtranslate( 0, curv[i],
                           <0,0.125*sin(2*k*pi*i/(n-1)),0> )
       #local i=i+1; #end
```

```
      // tracer les points nodaux:
      draw( 1, curv, finesse(0) + point(0.03)
                               + ma_couleur(<1,1,1>) )
      draw( 1, curv, finesse(0) + courbe(0.005)
                               + ma_couleur(<1,1,1>) )
      // tracer une pL interpolante
      #local spi = pLinterpolante( curv )
      draw( 1, spi, finesse(6) + point(0.01)
                               + ma_couleur(<0,1,0,0>) )
      // tracer une spline interpolante ;
      // ATTENTION: spi est un tableau de paraboles !!
      #local b_1 = curv[0] ; // 2*curv[0]-curv[1];
   #local spi = spline_interpolante_quadrique(b_1,curv,false)
      #local i=0; #while (i<taille(spi))
            draw( 1, spi[i], finesse(5) + point(0.01)
                               + ma_couleur(<0,0,1,0>) )
      #local i=i+1; #end
#end

#macro pF_3431()    // le cercle NURBS  (no box + AXO 7/8)
      #local curv = array[5]
      { <0.5,0.0,0.0,1>, <0.0,0.5,0.0,1>,
      <-0.5,0.0,0.0,1>, <0.0,-0.5,0.0,1>, <0.5,0.0,0.0,1> }
      #local b_1 = <0.5,-0.5,0.0,1>;
      // tracer les points nodaux:
      draw( 1, curv, finesse(0) + point(0.04)
                               + ma_couleur(<1,1,1,0.5>) )
      draw( 1, curv, finesse(0) + courbe(0.005)
                               + ma_couleur(<1,1,1>) )
      draw( 0, b_1, point(0.04) + ma_couleur(<1,1,0,0>) )
      // tracer la spline quadrique interpolante
      // (ATTENTION: spi est un tableau de paraboles)
  #local spi = spline_interpolante_quadrique(b_1,curv,true)
      #local i=0; #while (i<taille(spi))
            draw( 1, spi[i], finesse(5) + courbe(0.01)
                               + ma_couleur(<1,0,0,0>) )
      #local i=i+1; #end
#end

... end of the file pFbook.inc !
```

Final note :

1) the algorithms were programmed and the images produced on POVRAY, an outstanding free open source environment for Ray-Tracing development and rendering (GPL licence) ; the sources pFlibs.inc and pFbook.inc are under GPL licence ;

2) the images took a little detour via The Gimp, a superb software for touching up images, that is free open source (GPL licence) and saved in standard open jpg format ; the total weight of the images is around 35 Mo ;

3) this document was finalised on NeoOffice.org, « Bringing the power of OpenOffice on the Macintosh », a magnificent open source (GPL licence) applications suite producing open format (XML compressed ZIP). The text file of the full document composed in WRITER module weighs 108 ko (images are linked, not incorporated in the file) ; this file was directly exported from NeoOffice in pdf format in high quality, its weight is about 17 Mo. The front and back covers were produced in the DRAW module, the file weighs 19.4 ko (images are linked, not incorporated in the file) and was also exported in pdf format to produce a file weighing about 370 ko.

4) the writer, who isn't bad either, 80 ko, produces free open source code, and is looking forward to any comments that will help further this essay on curved forms.

.