Introduction to Computing with Geometry

Adrian Bowyer and John Woodwark

INFORMATION GEOMETERS

First published 1993

Information Geometers Ltd
47 Stockers Avenue
Winchester
SO22 5LB
UK

books@inge.com

ISBN 1-874728-03-8

This PDF version is basically the master at 110% enlargement from which the original edition was printed. Necessary changes have been made to pagination, typefaces and figures, and some typographical errors have been corrected.

Typeset and designed by the author.

. . . we are geometricians only by chance.

Dr Johnson

Contents

Foreword

Information Geometers has run its "Computing with Geometry" course a number of times in the last few years with the authors as presenters. This book is the material presented on the course.

Computing with geometry is a large (and in some places muddy) field. Here we have tried to cover all of it to a more-or-less uniform depth, measured in terms of utility. This means that some topics (such as interval arithmetic) get rather more coverage than would be expected from the frequency with which they appear in the literature, whereas others (such as parametric surfaces) get less. In the former case some extra attention is perhaps overdue, and in the latter the associated literature is so vast that a proportional treatment would have reduced the rest of the book to an appendix. We hope we've struck a reasonable balance. An annotated list of references is provided to allow you to dig deeper into topics that interest you particularly.

We acknowledge the helpful feedback in developing this text that we have received from participants on our courses. This book is a snapshot of an evolving document and, despite our best efforts to stabilize it for this printing, we expect that mistakes and (certainly) opportunities for improvement remain. We would be most grateful to hear of any that you find.

1

Introduction

If computer programs involving *money* are the dullest, then those involving *geometry* are the most interesting. Money is very useful stuff, but it is strictly, *strictly*, one-dimensional[1]. However much we have (and we could certainly do with more) it is just a bigger pile. With geometry, we have two, three or more *dimensions* to play with. These are not convertible—while there *may* be two dollars to the pound, no amount of ups or downs ever make a right or left—and so geometric programs have to be able to carry and maintain *multi-dimensional* information consistently.

But if we can cope with this complexity, geometric programs allow us to escape from the computer and start affecting more than numbers on a page. We can take data from cameras and other scanners, create pictures and animations, have metal cut into pretty shapes by numerically controlled (NC) machine tools, and move robots and autonomous vehicles.

We assume that you have had some experience in programming; C, FORTRAN and PROLOG are the languages we use for examples. If you have experience of one or two of these languages, you should find some hints as to the sorts of geometric elements, operations, structures and algorithms that you may come across when you start computing with geometry.

Computing with geometry is a large area of activity. It can be subdivided into a number of segments based on communities of re-

[1] Our Financial Wizard got very huffy about this. Don't we understand the difference between *Capital* and *Expense*? Alas, this is almost certainly our problem....

search interest, or on applications, or on both—where these coincide. Below, we attempt to beat the bounds of the subject by roughly describing the characteristics of seven such segments, each identified by a buzz-phrase. In this book, we do not try to relate to any particular application area, although there is probably some bias towards computer-aided design and manufacture.

Computer graphics

Computer graphics[2] has been in every sense the most visible manifestation of computing with geometry. The many introductory books dealing with the subject have made a lot of people familiar with:

Algorithms (such as Bresenham's algorithm) for drawing on raster screens.

Clipping and windows.

Transforms (including perspective).

Wire-frames and polygons.

'Hidden-line' and 'hidden-surface' algorithms.

Well, some of the above will be mentioned, but we take an iconoclastic view of this sort of graphics: it is an invaluable *debugging* tool.

Graphics in general is a river which has reached its delta, and recent work is spreading in many different directions, such as:

Global ('radiosity') lighting.

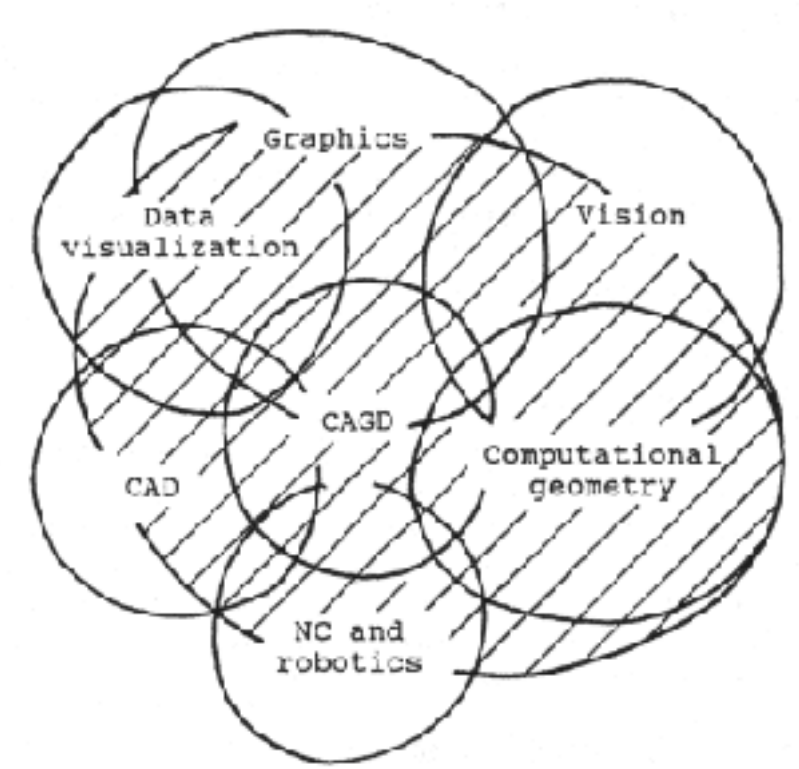
Facial animation.

Fall and motion of fabric.

Dynamics and collisions.

These sorts of problem use much more complicated geometry—in particular geometric structures (for instance to support solutions based on finite-element (FE) methods)—and are therefore of more interest. However, they also overlap with other subjects, from choreography to cubism, which are not on the geometrical menu.

[2] Called "graphics" from now on.



1(i)—An attempt to show the relationship between computing with geometry (the shaded region) and the seven research and application areas itemized in the text. A simple diagram of this sort can only be one try at drawing—with a very broad brush—the very complicated relationships that actually exist. Note that most of the research and application and areas have some part that is not shaded, meaning that they have significant non-geometric aspect. At the same time, the shaded region is itself larger than the application areas, meaning that there are (of course) applications of geometric computing (such as geographic information systems—GIS) which are not shown in this illustration.

Computer vision and image processing

Computer vision and image processing has a large and frighteningly technical literature. Many image-processing techniques can be classed as signal processing—for example, frequency transforms and image filters—and are outside our present scope. So is the artificial intelligence (AI) aspect of vision. Somewhere between pixels and perception is the reconstruction of shapes from one or more views of a scene; many of the topics we will cover are relevant to that activity.

Computer-aided design

Computer-aided design (CAD) is a rather general term; it includes subjects such as logic design which are not geometric. As well as encompassing many geometric topics, subjects such as mechanism design and solid modelling systems fit in here. Real computer-aided design systems often involve nasty-but-practical heuristics and approximations which may not easily be pigeon-holed into recognizable and respectable fields of intellectual endeavour.

Computer-aided geometric design

Computer-aided geometric design (CAGD) is usually used to refer to the study of *free-form* curves and surfaces. The modern grapefruit (to digress for a moment) is said to be a cross between the orange and an East Indian fruit called the *pumilo*, which is the size of a football but only contains as much flesh as the grapefruit; the rest is pith. Computer-aided geometric design has something in common with this fruit: a disproportionate amount of highly speculative academic work surrounding a central core of very useful techniques. We have tried not to be overawed by the size of the literature.

Computational geometry

Computational geometry is a phrase mostly used to refer to the study of geometrical *algorithms*, and particularly to their theoretical efficiency, or *order*. That means, if an algorithm runs in 10 seconds on 10 points, how long does it take on 100? For instance, if it takes 100 seconds, we call it an *order n algorithm;* if it takes 10000, it's order n^2; if it takes 200, it's $O(n \log n)$, and so on. While their

analyses may be complicated, the geometrical entities with which computational geometers concern themselves are often simple, such as a set of points. However, there is growing interest in the efficiency of algebraic manipulations, a lever on more complicated elements.

Data visualization

Data visualization is a new-ish term covering the manipulation and display of large amounts of data typically obtained from sensors such as satellites and body-scanners. These are like image data but the *dimensionality* of the data is higher; for instance, satellites collect electromagnetic radiation across the spectrum, and body-scans can be done with different types of instrument. The central problem is letting someone (often called 'the scientist') make sense of three or more dimensions. And a big part of that problem is constructing algorithms which are efficient when faced with the large amount of data involved in most applications.

Numerical control and robotics

You can worry a lot about geometric code when it's controlling a few tonnes of machine tool or industrial robot. A servomotor cannot move instantaneously to somewhere, as a cursor on a graphics screen will (well, nearly); and you cannot build a mechanical actuator which is infinitely thin, as the light rays coming to a camera are (well, nearly). These twin problems of dynamics and path planning affect the sort of geometry we need to do: things like calculating moments of inertia, trajectories, and offsets from surfaces.

From geometry to program

To get from a geometric concept to a program, we need to go through this sequence:

$$\textit{geometry} \rightarrow \textit{algebra} \rightarrow \textit{algorithm} \rightarrow \textit{program}.$$

Each of these little arrows hides big problems.

Geometry → algebra

We usually start off with some description of a problem, such as "where does such-and-such a straight line meet such-and-such a plane?". To get anywhere, we must first convert the straight line and the plane into an algebraic form in a coordinate system (usually Cartesian coordinates), and decide what has to be solved to find where they intersect. That all has to be done before actually writing an algorithm, although the requirements of an algorithmic solution must be borne in mind from the outset. This is a unique and difficult part of computing with geometry, and often we get no further without needing to think again. For instance "construct a surface a constant distance from an existing surface" (the *offset* problem) develops really fierce algebra very quickly. We will usually recast that problem at this first stage, into the form "find a (useful) set of points a constant distance from a given surface"; that's a lot less general, but a lot easier. Falling back on procedural or approximate solutions (yes, as soon as this) is often the better part of valour.

Even if the coordinate algebra goes well, there are other things to think about at this stage; for instance, if we wish to use only some parts of a whole geometric shape represented by an equation (as we usually do) then we need some formalism to cut and connect them; converting such bounding requirements into a graph-structure, or to set-theoretic algebra, can be another part of the process of interpreting the geometry that this first arrow represents.

Algebra → algorithm

Now we're on common ground with other bits of scientific programming. However, the sort of equations that come out of geometric problems have their own personality. For instance, computer algebra systems[3] are a common and practical tool for many applications, particularly symbolic differentiation and integration. Geometric problems tend to produce large sets of non-linear equations that break algebra systems like eggs; we've seen many omelettes. Only recently has this difficulty been recognized and at least one specialized geometric algebra system has been built.

Even so, in general we must make further concessions to the problem at this stage. We've tried to identify four levels at which we

[3]Called "algebra systems" from now on.

may be able to manage the *algebra* → *algorithm* transition:

Symbolic: the algebra system level; our algorithm will accept a range of equations, and works out what to do with them on the fly. This product is on the market, in the form of libraries of algebraic functions[4].

Analytic: the level at which we're usually happy to be; the algorithm is the direct embodiment of an algebraic solution.

Numerical: the level where we often end up; the problem can be formulated, but there is no *closed-form* solution[5]. We can often use a standard numerical method (e.g. relaxation) for simultaneous equations.

Approximate: the bargain basement; we don't even fancy the 'proper' algebra, and are working with a simplification.

These levels of attack are usually far from the whole story. They can be, and usually are, nested. For instance, we might approximate the geometry of a problem, and then formulate an analytic solution to that simpler geometry. The bounding problem recurs here, and in fact proliferates to become the whole question of an appropriate data structure.

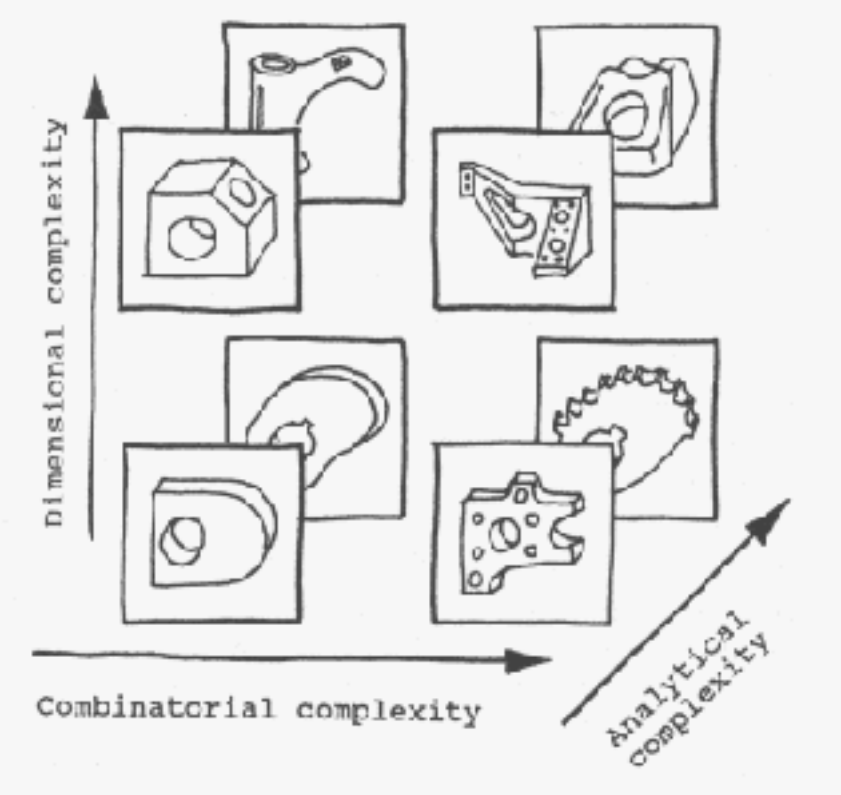
Algorithm → program

Our last 'arrow' is more difficult to differentiate from standard good programming practice. We may be constrained or helped by a particular circumstance relating to the application of geometry (an example constraint: the language available on a numerically controlled machine tool; an example help: integer coordinates on a display screen). We may need to be prepared to descend to low-level code on occasions; geometrical (particularly graphical) programs are notorious for inner loops that must run very fast; and accuracy problems are common. Against this, we have our debugging aid, the display screen, for feedback, even when there's no graphics (odd singular, graphics...) in the final program.

---

[4] Though only just (1993); for example, as an extension to the NAG library.

[5] A closed-form solution is an answer that can be written down straight away; for example the schoolbook formula $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ is a closed-form solution to the problem of finding the roots of a quadratic, but there is no closed-form solution to the problem of finding the roots of a quintic.



1(ii)—Dimensional, analytic and combinatorial complexity are independent, and thus themselves define a sort of 'three-dimensional space'.

Dimensional, analytic and combinatorial snags

Having looked at geometric programming as a process, at the risk of some repetition, we can classify the problems it presents from another viewpoint; that is, where is the *complexity* in a geometric problem? It tends to occur in three separate forms, involving: lots of dimensions, tricksy equations, and too many (different) bits of geometry at once. See Illustration 1(ii), which is not new [6].

Dimensional complexity

Geometry works remarkably *differently* in different numbers of dimensions. For instance, angles are well-behaved things in two dimensions, little devils in three. (Compare a globe and a clock face. What direction is West at the North Pole? You don't get this problem on a clock.) And beyond three dimensions things are worse; you *can* treat a moving solid as a four-dimensional object, but it doesn't help much, the equations are *not* symmetrical; the time dimension sticks out like a sore thumb.

Analytic complexity

We have already said a bit about this. Equations can be nice or they can be nasty; the pecking order goes something like this: *linear, quadratic, cubic, rational quadratic, with square roots, quartic plus, high-degree rational, with trig functions, with transcendentals, complete collapse....* These problems immediately become much worse when intersections, blends, and other combinations of equations must be considered. There are also other algebras—set theory, graph theory—to worry about under the heading of analytic complexity, as if there wasn't enough already.

Combinatorial complexity

This occurs most obviously when we have a lot of data; even a good number of points can cause problems. (You see, we can't order points in more than one dimension, so nice database techniques

[6] It's in *Computing Shape* (see the References and Bibliography for details of books and papers mentioned in the text), but the idea is originally attributable to Charles Lang, we believe.

come unstuck.) A good deal of computational geometry is about points, and efficient algorithms for dealing with lots of them. More complicated geometric entities give other combinatorial problems; we may need to make pairwise comparisons and worse. For example, to find all the edges formed by a number of intersecting surfaces, we need to compare every pair; to find the vertices, every set of three. This is an order n^3 algorithm for starters.

An additional, but different, combinatorial blow hits us when we have a lot of different types of geometry to deal with in one program. If (in a mere two dimensions) we want to find intersections between straight lines, we write a routine to do it; if we introduce circles, we need three routines: line-line, circle-circle, and line-circle. If we have ellipses, we need six routines, and so on. This hits you the programmer (because *you* have to write the routines); it doesn't just affect the length of time the program takes. It's a powerful incentive for algebraically more general routines, but it sends us looping back to the three arrows in the previous section....

2

Geometric basics

Dimensions

Let's assume everyone's familiar with things being in one dimension—along a straight line: in two dimensions—in a plane: or in three dimensions—in space. Now, if you think we're going to charge off into n dimensions at the drop of a hat, you're mistaken. In fact, there's a lot of hyperbole about n *dimensions* around. Permit us to make some statements that will set the ground rules for the following chapters.

Dimensionality looks easy to extend but it isn't

One, two, three dimensions sounds like one, two, three apples (or pears)—i.e. more of the same; but that's not how dimensionality works. Adding dimensions to a problem *qualitatively* changes the structures we can create, the algorithms we can use, and what is and is not feasible.

> In one dimension, everything is very easy (i.e. it's like programming with money); in fact there's not really any geometry at all. *But,* we often solve geometric problems by creating one-dimensional structures, and using the one-dimensional spaces defined to *sort* values into ascending or descending order, which we can't do in any higher-dimensional spaces.
>
> In two dimensions, we can see everything on a computer screen, which is a big help. Many quite complicated structures (e.g.

polygon edges) are one-dimensional structures *embedded* in the space, so we can hop back into a single dimension and do sorting (e.g. to order the vertices of a polygon).

In three dimensions, we have to project even to get on to a computer screen. On the other hand, we can describe objects to be built in the real world. We now have one-dimensional structures (curves) and two-dimensional structures (surfaces) embedded in our space. A polyhedron—for example—is an assembly of straight lines (edges) and surfaces (faces) and, unlike a polygon, there is no nice way of ordering them.

One, two or three dimensions sounds rather elementary. Why not four, five—or more? Many equations generalize deceptively easily into n dimensions, but that doesn't mean we can do anything sensible with them. In particular:

Just because mathematics—and the computer—can deal with more than three dimensions, don't expect this to help your *intuitive* understanding of higher-dimensional spaces: although there have been valiant attempts to persuade us differently (see Banchoff's book).

It is easy to generate data that is many-dimensional: for instance a multi-spectral Landsat picture has two spatial dimensions and perhaps four or five dimensions of sensor data in that space. But that does not mean that we have a multi-dimensional space in which all the dimensions have equal weight and meaning, in the way that spatial dimensions do.

Let's jump ahead of this chapter, and look at some illustrations. *Time* is sometimes said to be the fourth dimension. But the things that happen in time—either physically or algebraically—are not equivalent to things happening in an additional spatial dimension. To be more precise, temporal equations that are useful are rarely symmetrical between x, y, z and time; and those that are symmetrical are rarely useful.

We can generate examples without going above three dimensions. Take the implicit equation of a circle (this is where we get a little ahead of this chapter; if you're worried, come back here later):

$$(x - x_0)^2 + (y - y_0)^2 - r^2 = 0.$$

Add in time: an obvious thing would be to model the circle moving in a straight line. Suppose its trajectory is the parametric line

$$\begin{aligned} x &= x_0 + ft \\ y &= y_0 + gt, \end{aligned}$$

where t is time: seconds if you like. So after t seconds the circle will have become:

$$(x - x_0 + ft)^2 + (y - y_0 + gt)^2 - r^2 = 0.$$

Fine: if we replace t with z we get a quadric surface to be sure, but it's an elliptical cylinder: not very symmetrical, and nothing like a sphere.

Look at the thing the other way around. Take that sphere equation:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0.$$

Supposing we were to replace z with t (for time), what have we got? Not a moving circle at all, but a circle that is changing its radius according to the formula:

$$r = \sqrt{t^2 + pt + q}$$

(where p and q are composite constants derived from z_0 and r). Even in this simple case, there is no obvious intuitive link between the moving two-dimensional shapes and the static three-dimensional ones. In practice things are much worse; the 'temporal equations' of three-dimensional movements contain trigonometric terms, for representing rotations, which make them very difficult to handle.

There have been practical attempts to generalize animation, for instance, to a four-dimensional problem, with time as the fourth dimension; because the generality is to a greater or lesser extent illusory, they have not been notably successful (e.g. see Glassner's 1988 paper, and Woodwark's letter about it).

Projection

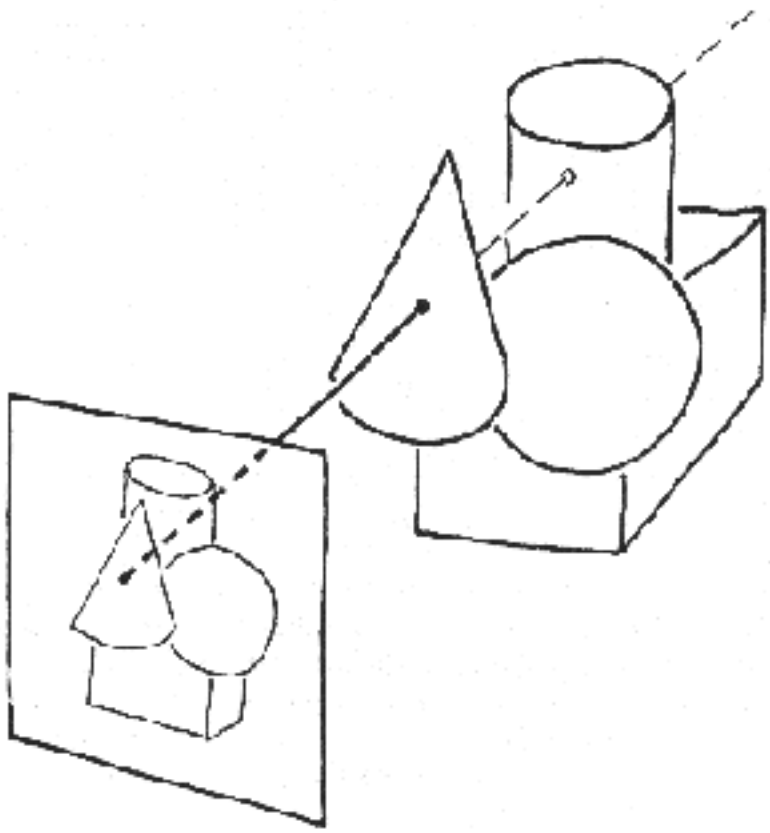
One of the nattiest things we can do to get around dimensional problems is to reduce the dimensionality of our data by throwing some of the dimensions away. That is what we do when we make a picture from a three-dimensional scene, and it is called projection.

Just throwing away a coordinate is seldom the best way of achieving this. For instance, in generating a picture of objects seen in perspective, there is quite a complicated relationship between the original three *object* coordinates and the two new *screen* coordinates. Further, we often want to throw away a good lot of data *en route*: in other words to sample the original geometry. In a picture of a solid object, for instance, we only want to see its faces nearest to the viewer.

That *visibility* problem is in turn itself susceptible to projection techniques. Ray-tracing is a well-known rendering technique; the object to be viewed is projected on to a number of straight lines, each of which corresponds to a ray going from one of the dots (*pixels*, for the initiated) on the graphics screen into the scene. In this case, the sampling is an intersection process (see Illustration 2(i)). The payoff from this approach is just the advantage of working in a single dimension that was mentioned above; the data about how far different objects are from the viewer can be *sorted*, and so the nearest intersection—which is also the nearest part of the scene—to the viewer is found quickly. In this case, the sampling is not an intrinsic property of the problem, but of the graphics device; a television-type picture is of course made up of a lot of dots.

In other cases, projection is *not* a big help. For instance, in applying surface patterns to an object, it would seem natural to work on the two-dimensional space defined by their surfaces. In practice, this is often very difficult because, although the surfaces *are* two-dimensional, they are great distortions of regular two-dimensional space; so it takes some effort to place a pattern on even a simple object without it becoming wildly distorted. Think of trying to draw a chequerboard pattern on to a cone; either we get a nasty seam, or we squash the pattern to nothing at the cone's apex.

To apply a pattern to an object, it turns out to be much easier if we can define that pattern in three dimensions (see Perlin's well-



2(i)—Ray-tracing; the object is projected on to the ray by intersection; the nearest intersection to the viewer determines part of the picture.

known SIGGRAPH paper). Although we have to take care to define it in such a way that we only need to evaluate it on the surface of the object, we don't need to take any account of the (weird and wonderful) shapes of the objects that will actually be patterned.

Points and vectors

If we've got some dimensions—a space—we're happy with, what about something to put in it? Points are a good start: just lists of coordinate values. A point is a list of displacements in each coordinate $(x, y, \ldots)$ from the origin $(0, 0, \ldots)$. Sometimes we'd like to carry these displacements around, and use them to position ourselves from a point we've already got. These 'floating' points are called *vectors*. If you add a vector to a point, you define another point, if you add two vectors you get another vector.... In fact we have a little algebra of the things:

$$\begin{aligned} P - P &= V \\ V + V &= V \\ V - V &= V \\ P + V &= P \\ P - V &= P \\ P + P &= \ldots \end{aligned}$$

Ha! The sum of two points is undefined[1].

Looked at another way, vectors define a movement through a certain distance in a specified direction. We can separate out the distance and the direction by *normalizing* the vector. Take a three-dimensional vector $\mathbf{a}$, (x_a, y_a, z_a). First extract the magnitude:

$$|\mathbf{a}| = \sqrt{x_a^2 + y_a^2 + z_a^2}.$$

This is the Pythagorean distance formula; we'll be seeing more of it later. If we divide all the components of the vector by this magnitude we get a new, *normalized*, vector: with a length of 1, but the

[1] In other words, the set of points and vectors is not *closed* under the operations of addition and subtraction.

direction unchanged:

$$\hat{\mathbf{a}} = \left(\frac{x_a}{|\mathbf{a}|}, \frac{y_a}{|\mathbf{a}|}, \frac{z_a}{|\mathbf{a}|} \right).$$

What about multiplication? We can easily multiply a vector by a constant, to get a longer or shorter one. There are also two very useful ways to combine vectors:

The *dot product* of two vectors $\mathbf{a}$ and $\mathbf{b}$ yields a scalar (i.e. a number) equal to $|\mathbf{a}||\mathbf{b}| \cos\theta$, where θ is the angle between them. This is a good way to find out the angle between vectors, especially unit vectors. We can get the product directly from the components of the vector, just by multiplying them together:

$$\mathbf{a}.\mathbf{b} = x_a x_b + y_a y_b + z_a z_b.$$

The *cross product* generates a new vector perpendicular to the two that are being multiplied, with a length equal to the (ordinary) product of their lengths:

$$\mathbf{a} \times \mathbf{b} = ((y_a z_b - y_b z_a), (x_b z_a - x_a z_b), (x_a y_b - x_b y_a)).$$

These two products of vectors are classic material for the 'inner loops' of geometric programs. They need to run very quickly, although there is little that can be done to reduce the number of arithmetic operations necessary. Think twice before encapsulating them in subroutines or functions however; the overhead from calling them could become a big factor in your code's performance. The slightly unfashionable idea of *macros*—routines that are expanded into 'in-line' code at the time of compilation—are an excellent compromise between legibility and efficiency in this context.

Implicit and parametric geometry

What about some more exciting geometric elements? When you were at school you probably learned about the straight line $y = mx + c$, and then found out that that equation couldn't represent vertical lines, which had to be $x = k$; oh, and then lines near vertical

have very large values of m, so you'd be better off with a form $x = m'y + c'$... and a lot more of that nasty sort of stuff.

Those *explicit* equations of curves of the form $y = f(x)$ or $x = f(y)$ (and explicit surfaces, $z = f(x, y)$ etc.) are only useful for describing *functions*, such as a signal varying with time, which are single-valued—so we know that they won't double back on themselves. In those cases (see Chapter 4) explicit equations are actually much easier to deal with than the more general geometric elements that we shall now look at.

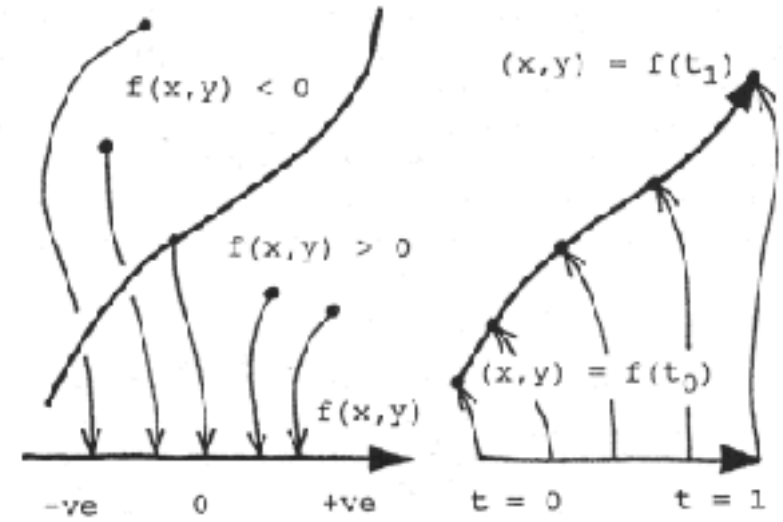
If we wish to formulate geometric elements that are not tied to alignment to a particular axis, then we have two choices:

Implicit equations

Implicit equations *classify* all the points in the plane, or in space, into two sets; so the curve or surface you are trying to define is the boundary between the two sets. The simplest way to do this is to evaluate a formula $f(x, y)$—or $f(x, y, z)$ in three dimensions—at every point. The result is a number; if it's negative, the point is on one side of the curve or surface; if it's positive, the point is on the other. We can also think of this as a *mapping* from the space on to a one-dimensional straight line, as shown in Illustration 2(ii).

The curve or surface itself is the set of points which map on to the origin of that one-dimensional straight line: those for which $f(x, y) = 0$ or $f(x, y, z) = 0$. They are called *implicit* curves and surfaces, because they are implied by the point classification. They are also referred to as half-spaces, because they divide the coordinate space up into two halves: points classified as positive and points classified as negative. The two halves are not in any sense equal, of course, and one may be bounded and the other not (as in the case of a circle) or both unbounded (as in the case of a plane).

However, because half-spaces divide up space, they must have a dimensionality that is one lower than the space in which they are embedded. So, in the plane, all implicit equations describe curves; in three-dimensional space, they all describe surfaces; and in four dimensions they describe volumes (visualize that if you can).



2(ii)—Curves as *mappings*: an implicit plane curve maps from two to one dimensions; the parametric version maps the other way, from one to two dimensions.

Parametric equations

Parametric equations are obtained by introducing one or more extra variables, or *parameters*, and calculating x, y—and z etc.—as functions of them:

$$\begin{aligned} x &= \phi_1(t, u, v, \ldots) \\ y &= \phi_2(t, u, v, \ldots) \\ z &= \phi_3(\ldots) \\ \vdots &= \vdots \end{aligned}$$

You can think of the parameters as another set of coordinates. (If the parameters were the *same* x, y, z coordinates, then these would be *transform* equations—see Chapter 9.) A parametric curve or surface can also be seen as a mapping in the opposite sense to an implicit one: in the case of a curve—look at Illustration 2(ii) again—going *from* a one-dimensional straight line *to* a two- or three-dimensional space.

However, because we can determine the number of coordinates and the number of parameters independently, there is no fixed relationship between the sort of geometrical element we can describe with a parametric equation and the space in which we are working. We can perform mappings which embed a two-dimensional space in a four-dimensional space, or whatever else we fancy. But by far the most useful parametric equations are functions of a single parameter in two and three dimensions (planar and space curves) and functions of two parameters in space (surfaces).

To summarize:

Implicit equations	*Classify points in the space*	*Fixed dimensionality relative to the space*
Parametric equations	*Generate points on the element*	*Any dimensionality*

In general, it is not easy to convert between the implicit and parametric equations of a geometric element (see Chapter 11). However, simpler shapes do have both implicit and parametric equations, and we shall spend the rest of this section looking at the simplest ones: the straight line, plane, circle and sphere. If you want more details,

financial considerations prompt us to recommend that other effort of ours: *A Programmer's Geometry*.

The straight line and plane

The implicit straight line is $ax + by + c = 0$; the parametric straight line is:

$$\begin{aligned} x &= x_0 + ft \\ y &= y_0 + gt. \end{aligned}$$

By extension the plane is $ax + by + cz + d = 0$ and

$$\begin{aligned} x &= x_0 + f_1 t + f_2 u \\ y &= y_0 + g_1 t + g_2 u \\ z &= z_0 + h_1 t + h_2 u. \end{aligned}$$

The circle and sphere

We have already seen that the implicit equation of the circle is

$$(x - x_0)^2 + (y - y_0)^2 - r^2 = 0,$$

which is easily extended to the sphere:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2 = 0.$$

The centre is (x_0, y_0, z_0) and the radius is r. The classic parametric equation of the circle is:

$$\begin{aligned} x &= x_0 + r\cos\theta \\ y &= y_0 + r\sin\theta \end{aligned}$$

and the classic, but not-too-useful, sphere is:

$$\begin{aligned} x &= x_0 + r\cos\theta\cos\psi \\ y &= y_0 + r\sin\theta\cos\psi \\ z &= z_0 + r\sin\psi. \end{aligned}$$

The angles θ and ψ are latitude and longitude respectively. There are two problems worth mentioning here. First, for computational

reasons we don't like trig functions (they take too long), so commonly replace cos and sin with the half-angle formulae:

$$\begin{aligned} \sin\theta &= \frac{2t}{1+t^2} \\ \cos\theta &= \frac{1-t}{1+t^2} \\ \text{where} \quad t &= \tan\frac{\theta}{2}. \end{aligned}$$

These have their own problems (see *A Programmer's Geometry*); they only do 90° worth of the circle. Second, the sphere is the first example of a nasty parameterization [2]. Geographers before and after Mercator have struggled with this well-known problem; even for such a simple shape there is—horrors—*no perfect solution*: nor even a universally acceptable best effort.

Bounding geometry

In practice, we seldom want an infinite curve; we've got to clip it to get it into a picture, if for nothing else. We want straight-line segments, circular arcs, and pieces of other curves and of surfaces too. The *bounding* that generates these pieces is a process of selecting *part* of something, and throwing the rest away. Implicit geometry classifies things, and so is the obvious tool for this job.

An element of implicit geometry classifies a space into two parts. The shape of the boundary between the parts is determined by the equation of the geometric object. When such equations are constructed from the usual algebraic operators, the result is—except in certain special and complicated cases—a smooth curve, surface etc. To get a shape with sharp corners—such as a rectangle in the plane—we need to introduce operators which can combine the regions classified by several 'ordinary' algebraic equations. If we consider an implicit piece of geometry as a set of points, we can see

[2] Note that the word *parameterization* has two meanings: "choice of parameters" (as here) and "conversion from implicit to parametric form" (the opposite of implicitization). There seems no easy way around this terminological trap.

that we combine these sets using the operators which already exist in set theory; here we shall need only the intersection operator, $\cap$.

For example, if the four sides of a rectangle aligned with the coordinate axes are:

$$\begin{aligned} x &= x_0, \\ x &= x_1 \quad (x_1 > x_0), \\ y &= y_0, \\ \text{and} \quad y &= y_1 \quad (y_1 > y_0), \end{aligned}$$

we can generate four sets of points from the inequalities:

$$\begin{aligned} x &\geq x_0 \\ x &\leq x_1 \\ y &\geq y_0 \\ y &\leq y_1 \end{aligned}$$

and combine them:

$$(x \geq x_0) \cap (x \leq x_1) \cap (y \geq y_0) \cap (y \leq y_1).$$

Any point which satisfies that equality is inside (okay, purists—or on the edges or corners of) the rectangle.

Just to make things more complicated, note that we could have achieved the same bounds by replacing $\cap$ with a minimum function:

$$\min\left((x - x_0), (x_1 - x), (y - y_0), (y_1 - y)\right) \geq 0.$$

Although this is not a continuously differentiable function, it is an implicit equation in its own right. When we come on to talk about penalty functions (Chapter 5) it will become apparent how this way of formulating set operations might be useful. Combining implicit functions with set-theoretic operators is a catching disease, and is in fact the foundation of set-theoretic (or constructive solid geometry—CSG) solid modelling, which we resist the temptation to expand into here....

What about parametric equations? They can easily be bounded by implicit equations *in terms of the parameters* (t, u, and so on are now the coordinates—so we say that these are implicit equations *in*

parameter space). As regards curves, they have a one-dimensional parameter space, so this bounding isn't usually thought of as geometric at all. We just specify a beginning and an ending parameter for the curve—say t_0 and t_1. But of course we are really using the intersection of the two one-dimensional implicit functions:

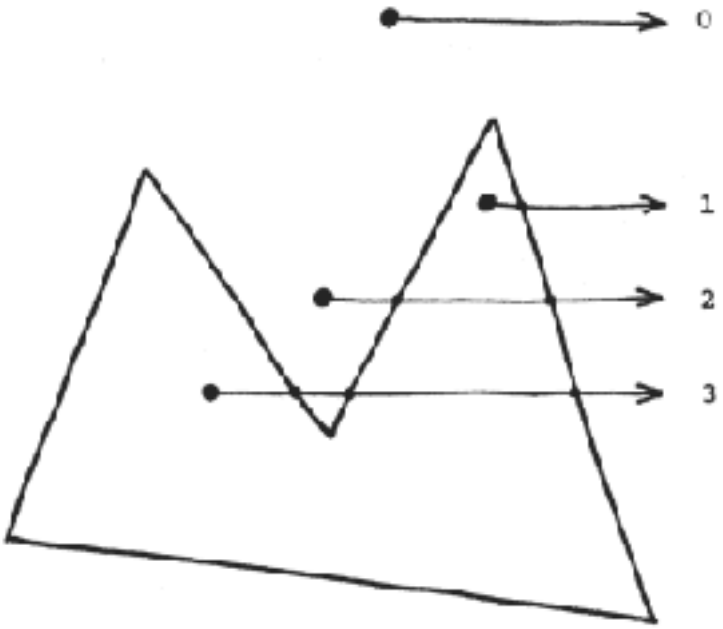
$$\begin{aligned} t &\geq t_0 \\ t &\leq t_1. \end{aligned}$$

In two dimensions, it becomes clearer what's going on. Typically, parametric surfaces are arranged as *patches*, and these are pieces of surface delineated by a rectangle (and that's usually a square) in parameter space. Again, in practice we just specify the limiting values of the two parameters (typically 0 and 1), and don't do a song and dance about the *geometry* of the bounds.

We could bound any region in parameter space, and triangular patches are quite common[3]. These areas of parametric surface are then matched—by interpolation processes we will look at—to curves and other patches in three-dimensional 'real' space. This takes us a certain distance in many applications, but eventually we will have, say, another surface that intersects a patch, and we will want to represent the result of that operation on the patch by bounding the pieces that are created. At this point, we often have no obvious bounding equations, either in parametric or real space. We can choose to work in either space, whichever is more convenient ("convenient" is an overstatement in this context, as you will see in Chapter 10).

Parametric equations can also bound things, but this is a much trickier business; now, an implicit equation, or a set-theoretic combination of implicit equations, cuts space into two parts. If we can make a curve or surface that cuts space into two parts out of other (i.e. parametric) elements, then that is a *de facto* half-space. A simple example is a polygon made up of parametric straight line segments. *As long as the polygon is complete,* we know that it defines an inside and an outside. The problem is classifying a point,

[3]They are needed for the corners of objects with rounded edges; they are not, in fact, usually done by setting up a triangular boundary in the space of the two parameters, but by setting up a parametric system *in the triangle itself.*



2(iii)—Using a 'ray-test' to find out whether a point is inside a polygon: if the number of intersections is odd, it is.

since we have no implicit equation. We've got to do that in a most oblique way, by shooting out a straight line from the point of interest to infinity, and counting how many times it crosses the polygon (see Illustration 2(iii)); the Jordan curve theorem tells us that whether the final count is odd or even determines whether we're inside the polygon or not. This is a classic area of activity (especially in three dimensions), and if we went any deeper into the matter we'd be talking about another type of solid representation, the boundary model....

3

Parametric curves and surfaces

We have seen that in two dimensions the straight line and circle can be expressed by equations for x and y, in terms of an auxiliary parameter t. As the value of t changes, new values of x and y are generated. In three dimensions, we need only add another equation for z. Thus a straight line in space is:

$$\begin{aligned} x &= x_0 + ft \\ y &= y_0 + gt \\ z &= z_0 + ht. \end{aligned}$$

You may (or may not) find it helpful to think of these parametric equations as mappings from a one-dimensional coordinate system with the single coordinate t into our usual two- or three-dimensional space. Thus, the equation of the straight line[1] provides a way to get from any value of t to a value of x, y or z (but not a way to get from values of x, y, z to a value of t, as only points on the curve *have* a corresponding value of t).

If we introduce a second parameter into the equations above, then we are mapping from a two-dimensional space (we'll use t and u for its two dimensions) into real space. But if we drop an equation and map from the t, u space into another two-dimensional space, then what we have is a two-dimensional transform: or, with three equations and three parameters, a three-dimensional transform. You can think of Chapter 9 when you get there in terms of parametric equations if it makes you happier. It makes most people quite a lot less happy, so we won't pursue that avenue.

[1] Or curve, if we have t^2, t^3 and so on in the equation.

Let's back up a bit here; if we map from t, u space into a three-dimensional coordinate system (i.e. x, y, z as usual), we create a surface. The parametric equation of a plane

$$\begin{aligned} x &= x_0 + f_1 t + f_2 u \\ y &= y_0 + g_1 t + g_2 u \\ z &= z_0 + h_1 t + h_2 u \end{aligned}$$

has already appeared.

It's pretty tempting to try more adventurous expressions in terms of t or t and u in the above equations. It is especially attractive because the whole essence of the parametric form is that it makes it easy to *generate points* on the curve or surface, and to bound them to a particular range of parameter values. If we can evaluate the expressions we have created, then we can get points; and we can get them within a particular range (interval) of values of t, or within a rectangle in the t, u plane. Of course that doesn't mean that other operations are so easy (more of that later).

Arguably the simplest way to extend parametric equations is to make the functions of the parameter(s) an arbitrary polynomial:

$$\begin{aligned} x &= a_0 + a_1 t + a_2 t^2 + \cdots \\ y &= \cdots \\ \vdots &= \vdots \end{aligned}$$

or, with two parameters,

$$\begin{aligned} x &= a_0 + a_1 t + a_2 u + a_3 tu + a_4 t^2 + a_5 u^2 + a_6 t^2 u + a_7 t u^2 + a_8 t^3 + \cdots \\ y &= \cdots \\ \vdots &= \vdots \end{aligned}$$

Alternatively we might consider rational polynomials; going back to one parameter—to keep things a little bit simpler—we have:

$$\begin{aligned} x &= \frac{a_0 + a_1 t + a_2 t^2 + \cdots}{b_0 + b_1 t + b_2 t^2 + \cdots} \\ y &= \frac{\cdots}{\cdots} \\ \vdots &= \vdots \end{aligned}$$

(Wonderful thing, the ellipsis...) Rationals give us the opportunity to represent conic sections exactly, using the $t = \tan\frac{\theta}{2}$ parameterization we saw in the last chapter. The alternative is to approximate them with a single high-degree polynomial. It can be shown (the Weierstrass theorem) that we can do this to arbitrary precision, but in general high-degree equations are not attractive. However, while rationals allow us to do more with lower-degree equations, the existence of the denominator causes an obvious problem: what happens if it is allowed to come close to, or to cross, zero? We can adopt a polynomial that is guaranteed to remain positive, such as the $1+t^2$ term in that circle parameterization, or simply ensure that the range of parameters within which we are working does not cause trouble. It makes things less straightforward, though, and for the rest of this chapter we shall ignore rational equations.

A more immediate problem is, what values are we going to use for all these constants a, b etc.? We will start to answer that question below, commencing with curves, and moving on to discuss surfaces.

Interpolation

When we looked at the straight line and the plane, we used notation of this sort: $x = x_0+ft$; now we have changed to $x = a_1+a_2t+a_3t^2+\cdots$. You may think that we've just run out of letters, and that's true, but the main point is that, in the equations for the straight line and plane, each of the coefficients had a meaning. Take the straight line in space, for instance; (x_0, y_0, z_0) is the point where $t = 0$ and (f, g, h) is a vector in the direction of the line.

When we start adding terms in higher powers of t, *the coefficients have no intuitive meaning*. We must therefore control curves and surfaces in a less direct way. The most common method is to make the curve or surface obey certain constraints, and the most common constraints are position and tangent direction, although other constraints, such as higher derivatives and curvature, are also used. Constructing a geometric element to obey constraints of this sort is called *interpolation*.

In general, the more coefficients there are in the equation of a curve or surface, the more constraints it is able to meet. That

means two things:

> We need enough coefficients for the job in hand (unless we're going to use more than one curve or surface—see later).
>
> We don't want too many coefficients, as the extra degrees of freedom that these provide for us (or encumber us with) have to be mopped up some other way.

So, let's look at the sort of interpolations we can do with points, straight lines and circles. A straight line is able to fulfil two constraints: it can go through two points, or go through a point and be tangent to a circle. A circle has three degrees of freedom, so it can go through three points, go through two and be tangent to a straight line, be tangent to two circles and a line, and loads more; or we can fix its radius, and it has two degrees of freedom like a line. Although the geometry in point, straight-line and circle constructions is simple, the constructions are interesting because there are independent position and tangent constraints in many different combinations.

With more general parametric polynomials, we are normally restricted to specifying position and tangent values at particular values of the parameter.

Lagrange interpolation

Lagrangian interpolation makes the curve or surface pass through a number of points. It can pass through a point for every coefficient in each equation. For instance, a quadratic will go through three. We decide what the actual values of the coefficients will be by solving a set of simultaneous equations in the coefficients. These are obtained by substituting the coordinates of each point— $(x, y, \ldots)$—and the value that we want the parameter(s) to have at that point— $t, \ldots$— into the curve or surface equations.

Hermite interpolation

In Hermite interpolation, we differentiate the equations of the curve or surface, and solve simultaneous equations for both position and tangent value at each of the points being interpolated. Thus twice as many coefficients are required as in the Lagrange case. A particularly important case is constructing a curve between two end

3(i)—Lagrange and Hermite interpolation used to construct a parametric cubic curve segment.

points, with known tangent values at each. That requires a curve with four coefficients in each equation, which are *cubics*; there is also an equivalent patch which runs between four corner points, and has 16 coefficients. Cubics are frequent sightings in computing with geometry: see Illustration 3(i).

The problem of parameterization

Interpolating parametric curves, deciding what the parameter values at each point will be—the issue of parameterization—is crucial. (That is a problem that does not occur with explicit, single-valued, curves and surfaces: and so we can see why these are preferred for drawing graphs and so on. And techniques from 'graphing' applications usually exploit this limitation, which is why we should be wary of trying to transplant them to more general geometric problems.) So, the problem with interpolating parametric curves is that, while the positions and tangent directions may be provided, we have to estimate the parameter values that the curve 'should' have when it passes each point, and the magnitude as well as direction of derivatives. In the case of Lagrange interpolation, the simplest choice is to space parametric values equally between point data. This works if the points are themselves quite evenly spaced; otherwise something better is needed. Since parameterization is related to curve

length, we would like to know what the length of the curve will be between each data point; but that is putting the cart before the horse, because we haven't got the curve yet. One could implement a technique of successive refinement—set up one curve, get the curve lengths from it, and thus obtain new parameter values at the data points, and repeat the exercise—but this rigmarole is not usually attempted; it would probably be difficult even to prove that it would converge.

The usual solution is chord-length parameterization, where the parameter values at the points are based on the lengths of the straight-line segments connecting them. This is a good workhorse, giving trouble only when there are abrupt 'corners' implied by the data, and changes of spacing. Further refinements involve taking the angle between successive spans into account (see Farin's book *Curves and Surfaces for Computer Aided Geometric Design* for more detail).

With Hermite interpolation, similar problems occur; and it must be remembered that magnitudes of derivatives of the form $\frac{dx}{dt}$ etc., *are* related to the actual size of the curve in the units of length being used. Thus, if we scale a curve by scaling the values of its Hermite coefficients, we must scale the derivatives explicitly. That's easy enough for a simple scaling, but what about a shear transform?

All these remarks have been addressed to the problem of interpolation, but also apply to curve fitting. Again, this is a process that works well with explicit geometry, and fairly well with implicits (except that normalization causes a problem). With parametric geometry, we again have to decide in advance what parameter value each point will correspond to. But if the points are at all dense, this is difficult: chord-length parameterization is certainly useless.

We conclude this section with C code for Lagrange and Hermite interpolation. The first procedure works out the Lagrangian interpolating cubic parametric polynomial through four points in three dimensions. The points will be supplied in `px`, `py`, and `pz`. The parameter on the curve at the first point will be 0, and the parameter at the last point 1; the coefficients of the polynomial will be returned in `polyx`, `polyy`, and `polyz`; `polyx[3]` is the coefficient of

t^3 in x and so on. The parameter values at the middle two points on the curve will be returned in `t1` and `t2`.

```
#include <math.h>
#include <stdio.h>

/* Absolute value macro */

#define fabs(a) (((a) < 0.0) ? (-(a)) : (a))

/* Almost 0 - adjust for your application */

#define ACCY (1.0e-6)

int lagrange(px,py,pz,t1,t2,polyx,polyy,polyz)
float      px[4],polyx[4];
float      py[4],polyy[4];
float      pz[4],polyz[4];
float      *t1,*t2;
{
     float xd,yd,zd,dfl;
     int i;

/*
     Sum the distances between the points to use to
     scale t1 and t2.  Note the extremely tiresome
     casting that needs to be done because all of the
     standard C maths library is in doubles.
*/

     dfl = 0.0;
     for(i = 1; i < 4; i++)
     {
        xd = px[i] - px[i-1];
        yd = py[i] - py[i-1];
        zd = pz[i] - pz[i-1];
        dfl = dfl + (float)sqrt((double)
                    (xd*xd + yd*yd + zd*zd));
```

```
        if (i == 1) *t1 = dfl;
        if (i == 2) *t2 = dfl;
     }

     if (dfl < ACCY)
     {
        fprintf(stderr,
               "Lagrange: curve too short: %f\n",dfl);
        return(1);
     }

     *t1 = *t1/dfl;
     *t2 = *t2/dfl;

/*
     Call the procedure to compute the coefficients in
     each coordinate.
*/

     if(lagrange_coeffs(px,polyx,t1,t2)) return(2);
     if(lagrange_coeffs(py,polyy,t1,t2)) return(3);
     if(lagrange_coeffs(pz,polyz,t1,t2)) return(4);

     return(0);

} /* lagrange */


/*
     Procedure to compute the coefficients in one
     dimension of the Lagrangian cubic through four
     points.  The code reflects the algebra.
*/

int lagrange_coeffs(p,poly,t1,t2)
float      p[4],poly[4];
float      *t1,*t2;
{
     float d1,d2,d3,t1s,t2s,t1c,t2c,denom,tt;
```

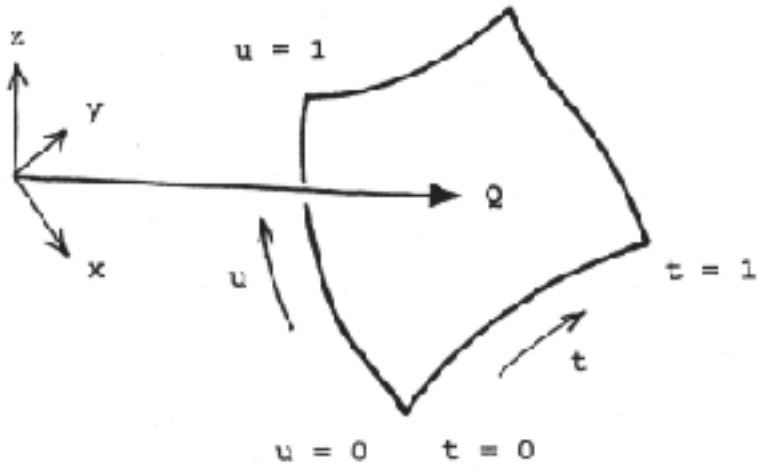
```
    d1 = p[1] - p[0];
    d2 = p[2] - p[0];
    d3 = p[3] - p[0];
    t1s = (*t1)*(*t1);
    t2s = (*t2)*(*t2);
    t1c = t1s*(*t1);
    t2c = t2s*(*t2);

    denom = (t2s - (*t2))*t1c;

    if (fabs(denom) < ACCY)
    {
       fprintf(stderr,
        "Lagrange_coeffs: increments too short: %f\n",
         denom);
       return(1);
    }

    tt = (-t2c + (*t2))*t1s + (t2c - t2s)*(*t1);

    poly[3] = (d3*(*t2) - d2)*t1s +
              (-d3*t2s + d2)*(*t1) +
              d1*t2s - d1*(*t2)/denom + tt;

    poly[2] = (-d3*(*t2) + d2)*t1c +
              (d3*t2c - d2)*(*t1) +
              d1*t2c + d1*(*t2)/denom + tt;

    poly[1] = (d3*t2s - d2)*t1c + (-d3*t2c + d2)*t1s +
              d1*t2c - d1*t2s/denom + tt;

    poly[0] = p[0];

    return(0);

} /* lagrange_coeffs */
```

The second procedure computes the Hermite interpolant in three

dimensions through two points with two gradient vectors at the ends. The points are `p0` and `p1`, and the gradients are `g0` and `g1`. The coefficients of the interpolating polynomial are returned in `polyx`, `polyy`, and `polyz`. The algebra in this case is much simpler than that for Lagrangian interpolation.

```
void hermite(p0,p1,g0,g1,polyx,polyy,polyz)
float p0[3],p1[3],g0[3],g1[3];
float polyx[4],polyy[4],polyz[4];
{
      void hermite_coeffs();

      hermite_coeffs(p0[0],p1[0],g0[0],g1[0],polyx);
      hermite_coeffs(p0[1],p1[1],g0[1],g1[1],polyy);
      hermite_coeffs(p0[2],p1[2],g0[2],g1[2],polyz);

} /* hermite */

void hermite_coeffs(p0,p1,g0,g1,poly)
float p0,p1,g0,g1;
float poly[4];
{
      float d,g;

      d = p1 - p0 - g0;
      g = g1 - g0;

      poly[0] = p0;
      poly[1] = g0;
      poly[2] = 3.0*d - g;
      poly[3] = -2.0*d + g;

} /* hermite_coeffs */
```

Surface patches



3(ii)—A parametric patch $\mathbf{Q} = \mathbf{F}(t, u)$ defined over the interval $0 \leq t \leq 1, 0 \leq u \leq 1$.

Surface patches are parametric surfaces of the form

$$\begin{aligned} x &= f_1(t, u) \\ y &= f_2(t, u) \\ z &= f_3(t, u) \end{aligned}$$

(which we can also write with *vector coefficients*, as $\mathbf{Q} = \mathbf{F}(t, u)$ a paper-saving measure that will be increasingly used in this chapter). A patch can be defined over any parametric portion of the (t, u) *parameter space*, but is easiest to deal with over the two-dimensional interval:

$$\begin{aligned} 0 \leq \quad t \quad &\leq 1 \\ 0 \leq \quad u \quad &\leq 1. \end{aligned}$$

The primary constraint on these square areas of surface is that their boundary should match the boundaries of adjacent patches. This can be met in two ways:

Define the patch in terms of its boundaries; this is the approach taken in the Coons patch (see Coons' 1967 paper), where the

interior of the patch is the result of a blending operation performed on its four boundary curves.

Explicitly choose a patch equation that gives known types of curves at the boundaries.

The *Cartesian product* patch has equations in which the terms are products of powers of t from 0 to 3 (i.e. $1, t, t^2, t^3$) and the same powers of u (i.e. $1, u, u^2, u^3$). There are 16 such terms in the equation corresponding to each coordinate:

$$\begin{aligned} x &= a_0t^3u^3 + a_1t^3u^2 + a_2t^3u + a_3t^3 \\ &+ a_4t^2u^3 + a_5t^2u^2 + a_6t^2u + a_7t^2 \\ &+ a_8tu^3 + a_9tu^2 + a_{10}tu + a_{11}t \\ &+ a_{12}u^3 + a_{13}u^2 + a_{14}u + a_{15}. \end{aligned}$$

Of course there are corresponding equations in y and z, making 48 terms in all. It's easy to see what curves we will get at the edges of the patch. At the edge $u = 0$, for instance, the equation degenerates to:

$$x = a_3t^3 + a_7t^2 + a_{11}t + a_{15}.$$

In fact, for any *fixed* value of t or u, we get out a cubic *iso-parametric* curve in the other parameter.

As well as knowing the position of the boundary curves, we need to match tangents—and maybe higher derivatives, radius of curvature etc.—*across* the boundaries. Let's find an expression for the tangent across the $u = 0$ edge of a Cartesian product patch. First differentiate with respect to u:

$$\begin{aligned} \frac{\partial x}{\partial u} &= 3a_0t^3u^2 + 2a_1t^3u + a_2t^3 \\ &+ 3a_4t^2u^2 + 2a_5t^2u + a_6t^2 \\ &+ 3a_8tu^2 + 2a_9tu + a_{10}t \\ &+ 3a_{12}u^2 + 2a_{13}u + a_{14}. \end{aligned}$$

Then set $u = 0$ again:

$$\left.\frac{\partial x}{\partial u}\right|_{u=0} = a_2t^3 + a_6t^2 + a_{10}t + a_{14}.$$

Any other patch which has the same boundary curve and derivative polynomial[2] at its edge will match this patch at its $u = 0$ edge; similar constraints apply at the other edges[3].

In a common case, we have a network of space curves ready-designed. Annoyingly, it works out that bicubic patches have just one too many degrees of freedom (in each dimension) to surface such a network without the supply of additional data. (Higher-degree patches have lots of extra degrees of freedom, quadratics don't have enough.) If the patches are being determined by a Hermite technique, or as a geometric relationship between the allowable positions of the internal points in adjacent patches (or—looking ahead—the corresponding vertices of a Bézier control mesh), then the extra degrees of freedom emerge as so-called twist vectors at the patch corners:

$$\frac{\partial^2 \mathbf{Q}(t,u)}{\partial t \partial u}.$$

See Illustration 3(iii) for a sketch of both of these cases. Suggested solutions to this problem have padded out many a thesis. (That's why they're called higher-degree patches....)

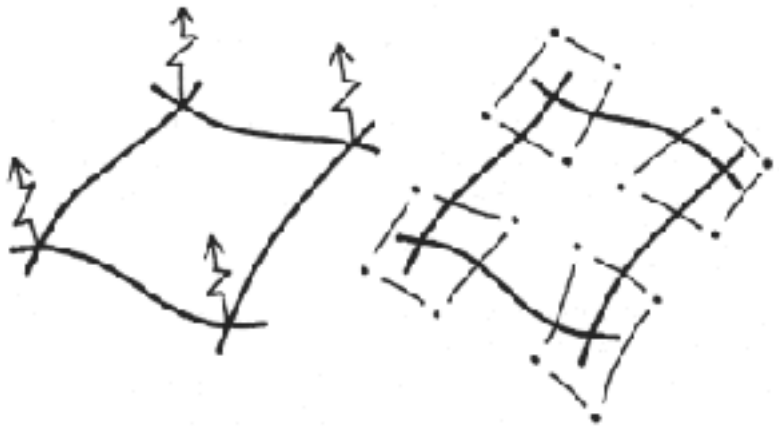
For now, let's look at something simple. How do we draw a patch? The simplest way is to make a line drawing of an iso-parametric grid. Let's look at some code to do that. We could simply write a routine to evaluate x, y and z in the obvious way, and keep calling that; but it's not very efficient because, along iso-parametric curves, either t or u is fixed, and so we would be doing a lot of recalculation. The code that follows draws an iso-parametric straight line at a specified value of u, varying t in n steps. We assume that the coefficients are available as three sets of variables `ax[16], ay[16], az[16]`, which correspond to the subscripts in the equations above and the three coordinate axes.

Here's the result; note the nested or Horner forms[4] of the equa-

[2]For some applications, continuity on higher derivatives is required.

[3]Continuity of parametric derivative is not an essential condition for smoothness; patches might also join smoothly if the tangent derivatives at the mutual edge were in the same directions but had different magnitudes; or the derivatives might not match at all, but there could still be a common normal (i.e. the surfaces could be locally coplanar) at the joint.

[4]The *Horner form* of a polynomial $a_0 + a_1 t + a_2 t^2 + a_3 t^3 + \cdots$ is $a_0 + t(a_1 +$



3(iii)—The extra degree of freedom of a cubic patch in interpolating over a network of curves: twist vectors in the Hermite case, and constraints on inner point positions in the Lagrange interpolation. Each constraint is shared with four adjacent patches, thus yielding an average of one per patch, except at the edge of a patched surface.

tions appear again. In this case they save pre-calculation of u^2 and u^3 at the beginning of the loop and of t^2 and t^3 within it.

Plotting is performed by two somewhat notional routines called `move(x,y,z)` and `plot(x,y,z)`, which move the 'pen' in 'pen up' and 'pen down' mode respectively. We assume that they are kindly going to deal with projection, clipping and so on for us.

```
   float ax[16], ay[16], az[16];

/*
 * The next coefficients are to be used in the inner
 * loop, so we don't want lots of array-subscript
 * arithmetic; hence no cx[4] etc.
 */

   float cx_0,cx_1,cx_2,cx_3, cy_0,cy_1,cy_2,cy_3,
               cz_0,cz_1,cz_2,cz_3;

   float t, u, dt, x, y, z;
   int i, n;

   cx_0 = ax[3]  + u*(ax[2]  + u*(ax[1]  + u*ax[0] ));
   cx_1 = ax[7]  + u*(ax[6]  + u*(ax[5]  + u*ax[4] ));
   cx_2 = ax[11] + u*(ax[10] + u*(ax[9]  + u*ax[8] ));
   cx_3 = ax[15] + u*(ax[14] + u*(ax[13] + u*ax[12]));

   cy_0 = ay[3]  + u*(ay[2]  + u*(ay[1]  + u*ay[0] ));
   cy_1 = ay[7]  + u*(ay[6]  + u*(ay[5]  + u*ay[4] ));
   cy_2 = ay[11] + u*(ay[10] + u*(ay[9]  + u*ay[8] ));
   cy_3 = ay[15] + u*(ay[14] + u*(ay[13] + u*ay[12]));

   cz_0 = az[3]  + u*(az[2]  + u*(az[1]  + u*az[0] ));
   cz_1 = az[7]  + u*(az[6]  + u*(az[5]  + u*az[4] ));
   cz_2 = az[11] + u*(az[10] + u*(az[9]  + u*az[8] ));
   cz_3 = az[15] + u*(az[14] + u*(az[13] + u*az[12]));
```

$t(a_2 + t(a_3 + \cdots)))$. It saves arithmetic when the polynomial is being evaluated.

```
t = 0.0;
dt = 1.0/(float)(n-1);

move(cx_3,cy_3,cz_3);

for (i=1; i<n; i++) /* We want to loop n-1 times */
{
   t = t + dt;

   x = cx_3 + t*(cx_2 + t*(cx_1 + t*cx_0));
   y = cy_3 + t*(cy_2 + t*(cy_1 + t*cy_0));
   z = cz_3 + t*(cz_2 + t*(cz_1 + t*cz_0));

   plot(x,y,z);
}
```

Splines

It is not feasible to make a long curve or complicated surface with a single high-degree polynomial. One problem is that of parameterization; poorly chosen parameter values at the data points make the curve more and more wiggly as the degree of the curve increases. Also, high-degree polynomials (this category usually starts somewhere between degree 6 and 10) are expensive to compute—because of the number of terms—and extremely sensitive to inaccuracies in their coefficients. (There is more about this in the next chapter.)

The obvious solution (well, fairly obvious solution) is to *knot* a lot of simple pieces of curve or surface together, and a lot of energy has been frittered away over this very matter. Why? It is, you might think, as easy to make ten curves as one, if the conditions that each one is to meet are precisely and individually defined; but usually they are not. More frequently we have some data specified for each span, typically its position; and other requirements, typically tangent or curvature continuity, are specified over the whole curve or surface. So we need to invent values for local data that satisfy the global constraint, and possibly are also optimal in some defined way. Perhaps we want to minimize or maximize the integral of some

quantity over the curve or surface, or perhaps we'll be satisfied with a curve that is optimal in the designer's opinion.

The word *spline* covers all such piecewise curve and surface techniques, and we'll classify splines into three types:

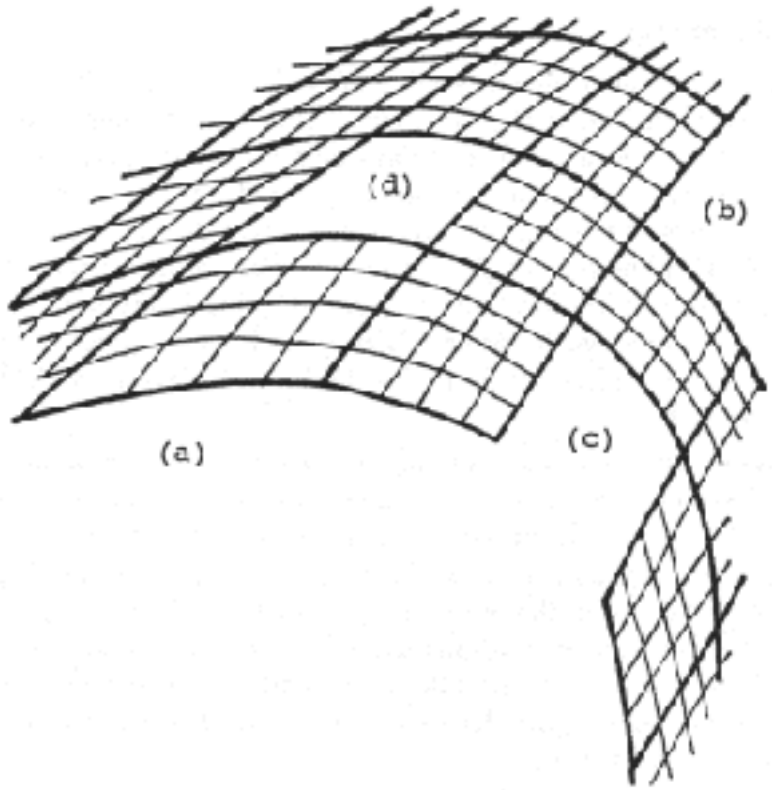
Sequential splining

This is a term that we've just made up: a phrase to describe the simplest approach to the generation of piecewise curves and surfaces. We start with one piece of the curve or surface—usually the largest and most important—and then join further pieces to it. These extra pieces will need at least enough degrees of freedom to meet the constraints imposed by what's already in place, and you'd better have some more in hand, otherwise the new geometry is totally determined, and will probably start to oscillate wildly a few curve or surface elements out from the original piece of geometry.

This is rather a faint-hearted way to go about designing long curves, but it's very simple; each new curve segment meets no, one, or two defined end-conditions, and can be created by Hermite interpolation or similar processes. For surface patches, this method of construction is a common one; adding new patches is more complicated, as the number of conditions to be met grows with the number of edges that are shared with patches that have already been created (see Illustration 3(iv)). Thus, knowing the order in which to create a patched surface is a significant piece of design expertise, and so the method is difficult to automate.

Local splining

To avoid order-dependence, we should like to determine all the spans or pieces of surface at the same time. One way of doing that is to take into account a number of data points that are near—but not on or adjoining—a particular span or piece of surface. The B-spline we shall meet in the next chapter does exactly this, although in the case of a B-spline surface, *just* within patches, so the overall scheme is a hybrid between local and sequential spline interpolation. But the sequential stage is easier because the B-spline patches, being assembled from pieces of simpler surface, can be larger: well, that's the sales talk.



3(iv)—When constructing a large patched surface, it is not possible to avoid patches that join others at one `(a)` and two `(b)` edges. Patches that meet others at three `(c)` and four `(d)` edges are encountered in editing a surface.

It is easy to invent (conceptually) simpler local splines: a favourite is the Overhauser curve (see our other effort *A Programmer's Geometry*). The idea is to fit some subset of the data available with a simple curve segment. In effect we are generating tangent or higher-derivative data from the position data supplied. Though dependent upon these data, these tangents and so on are non-unique—we could just as easily use others of differing magnitude, for example. Once the tangents and higher derivatives are defined, each span can be interpolated separately.

Global splining

This is the real McCoy. Now we try to determine a whole curve [5] in a single process. The tangents, or other criteria that must be matched at the knots between the spans, are equated together in appropriate pairs, and the large set of equations that results is solved, yielding the end conditions from which the spans can be constructed. The benefit of this approach is at least the opportunity to get a 'better' curve; whatever we are optimizing will be optimized over the whole curve, not just some segments. The disadvantage is that changes to the data defining a global spline proliferate—at least to some extent—along the whole curve, so it is possible that an improvement to a curve in one place will wreck it elsewhere. There is also that large set of simultaneous equations to solve, for which matrix methods are preferred; the matrices are strongly diagonalized and relatively easy to deal with (see the book *Numerical Recipes*).

Types of continuity

At the knots of spline curves (where two pieces—or spans—are joined) we do not usually attempt to match all the non-zero derivatives that the spans possess. Cubic spans, for instance, have a non-zero third derivative; but commonly only position and first derivative—and possibly second too—are enforced across the knots. These degrees of continuity are commonly written $\mathbf{C}^0$ (position),

[5] Surfaces are not globally splined, but some techniques do exist for splining a mesh of curves, where the knots in the two sets are coincident and share tangent planes.

$\mathbf{C}^1$, $\mathbf{C}^2$ etc.

We should bear in mind that splines were originally conceived as explicit curves (reminder: $y = f(x)$) and the parameterization—which we need to get geometric flexibility—is a liability. Thus, continuity in parameter derivative is not the same as continuity in a geometric sense; this is a variant of the parameterization problems we saw in the last section. First- and second-degree geometric continuity are continuity of unit tangent vector, and continuity of curvature. They would be the same as parametric continuity if we had a true arc-length parameterization; since we never do, that's not such an insight (see Farouki and Sakkalis' paper on the impressively named *Pythagorean hodographs*, which have at least a rational polynomial expression for arc length).

The unit tangent is easily written down:

$$\frac{\frac{d\mathbf{Q}}{dt}}{\left|\frac{d\mathbf{Q}}{dt}\right|}.$$

The curvature of a parametric curve is given by the following formula, which involves a vector product between the first and second derivatives:

$$\kappa = \frac{\left|\frac{d\mathbf{Q}}{dt} \times \frac{d^2\mathbf{Q}}{dt^2}\right|}{\left|\frac{d\mathbf{Q}}{dt}\right|^3}.$$

You can see why derivatives of parameter are more commonly used for matching between curve segments, although the result is a more constrained—and hence higher-degree—curve than would otherwise be necessary. In three dimensions, the matching of position and first derivative are not much different. Second derivatives must now be matched in direction as well as magnitude. In fact, the first, second and third derivatives of a curve in three dimensions form a local coordinate axis called the *Frenet frame*, which changes its orientation along the curve. As well as curvature, we now have the concept of torsion: the rate at which the Frenet frame rotates 'around' the curve. Spline curves can be made to match any or all of this stuff in a profusion of combinations.

4

Bernstein-basis curves and surfaces

Bernstein-basis functions generate Bézier and B-spline curves and surfaces. There is more literature about these functions than about any other topic in computing with geometry: very likely, more than about every other topic put together. Much of it is highly mathematical, with complicated—and variable—notation. There are also a number of books that provide introductions to the research literature (see Rogers and Adams' book, or Farin's, for example).

What can we hope to accomplish in this present volume? Let's try to do just two things:

> Firstly, we introduce the ideas of Bernstein functions, placing the emphasis on comprehensibility, rather than generality; it is much more difficult to obtain an intuitive appreciation of their properties than to perform at least the simpler algebraic manipulations of them.
>
> Secondly, we give a checklist of the advantages and disadvantages of Bernstein-basis curves and surfaces, compared to more 'obvious' methods; this is an area where popular technical—and promotional—literature is often hyperbolic [1], to say the least.

Introducing Bernstein-Bézier curves

You may think of a parametric equation $\mathbf{Q} = \mathbf{A}_0 + \mathbf{A}_1 t + \mathbf{A}_2 t^2 + \ldots$ as an ordinary polynomial; but it can also be seen as a weighted

[1] Hyperbole, not -a: not geometric, for once.

combination of a set of powers of the variable, t in this case; these powers are the *basis* of an ordinary polynomial, which is going to get airs by calling itself a *power basis* polynomial in this chapter.

If we look at the way that these basis functions vary over an interval, say $[0, 1]$ as in Illustration 4(i), they don't look intuitively symmetrical or anything nice. They don't look much better over a symmetrical interval $[-0.5, 0.5]$. Further, as already mentioned, the coefficients beyond A_1 really have no apparent geometrical significance.

However the 'straight-line bit' is both symmetrical and comprehensible. It's even better when we rewrite it as $\mathbf{Q} = (1-t)\mathbf{P}_0 + t\mathbf{P}_1$. In that case, we can see that $\mathbf{P}_0$ is the point we start from, $\mathbf{P}_1$ is the point we go to, and we make steady progress in between. But how can we make curves this way? The de Casteljau construction is a way of doing just that. Add a new point, $\mathbf{P}_2$, and set up two straight-line segments (see Figure 4(ii)):

$$\begin{aligned} \mathbf{P'}_0 &= (1-t)\mathbf{P}_0 + t\mathbf{P}_1 \\ \mathbf{P'}_1 &= (1-t)\mathbf{P}_1 + t\mathbf{P}_2 \end{aligned}$$

Then create a new straight-line segment between the moving points on the first two; as we see in Illustration 4(ii), this is actually a point on the curve:

$$\mathbf{Q} = \mathbf{P''}_0 = (1-t)\mathbf{P'}_0 + t\mathbf{P'}_1$$

In the old power basis, we've actually created the polynomial

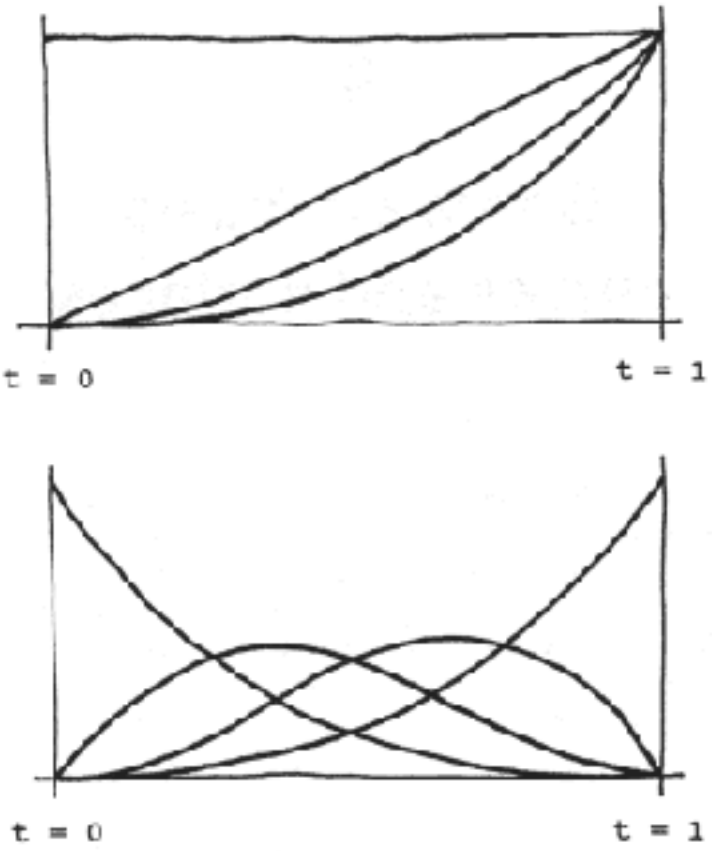
$$\mathbf{Q} = \mathbf{P}_0 + 2(\mathbf{P}_1 - \mathbf{P}_0)t + (\mathbf{P}_0 - 2\mathbf{P}_1 + \mathbf{P}_2)t^2,$$

which is once again less than obvious; but then our new basis functions are actually $(1-t)^2$, $2t(1-t)$ and t^2, which are symmetrical in the interval [0–1], as shown in Illustration 4(i).

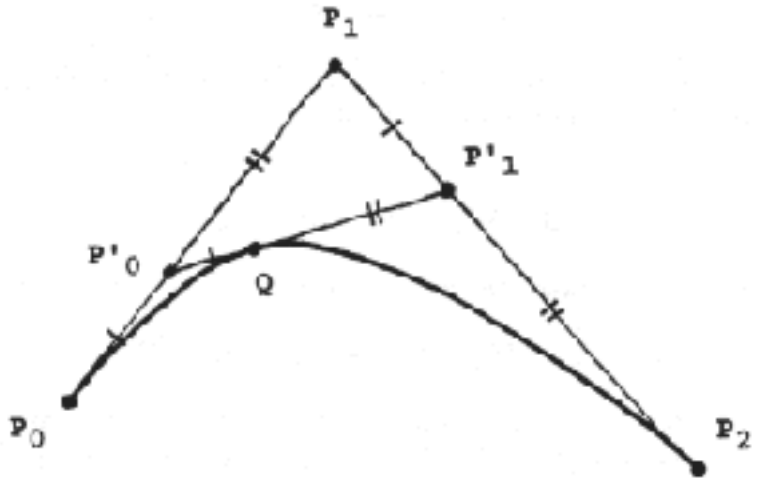
We can generalize the de Casteljau construction, but it's quicker to jump in and say that the polynomials we are creating are:

$$\mathbf{Q}(t) = \sum_{i=0}^{i=m} \frac{m!}{(m-i)!i!} t^i (1-t)^{(m-i)} \mathbf{P}_i,$$

where the term $\frac{m!}{(m-i)!i!}$ reflects the fact that the points near the middle of the curve are used more often in formulating the straight-line segments, so have a greater influence in the quadratics and the



4(i)—Power and Bernstein basis functions for a cubic polynomial.



4(ii)—A quadratic Bézier curve constructed out of two fixed and one varying straight-line segment. The terms t and $1 - t$ in the equation correspond geometrically to equal ratios being maintained between each single- and double-ticked part of each straight-line segment.

higher-degree curves, and the final curve. The degree of the last curve is m, which is one fewer than the number of points which control it (because we started with $\mathbf{P}_0$).

The weights $\frac{m!}{(m-i)!i!}t^i(1-t)^{(m-i)}$ that we are using to multiply $\mathbf{P}$ are the *Bernstein basis*. Together, they (merely?) comprise a very fancy way of writing down the number 1:

$$1 = [t + (1-t)]^m = \sum_{i=0}^{i=m} \frac{m!}{(m-i)!i!} t^i (1-t)^{(m-i)}.$$

There are many interesting (well, quite interesting) relations that we can derive from the Bernstein basis. Converting to a power basis is relatively straightforward:

$$\mathbf{A}_k = \sum_{j=0}^{k} (-1)^{(k-j)} \frac{m!}{j!(j-k)!(m-k)!} \mathbf{P}_j.$$

The reverse operation is much less easy to formulate, and it has only fairly recently become apparent that there is a closed form for this; we remember using Gaussian elimination for the purpose in the early Eighties (maybe that was just ignorance). The closed form (see Farouki and Rajan's 1987 paper) is:

$$\mathbf{P}_k = \sum_{j=0}^{k} \frac{k!(m-j)!}{m!(k-j)!} \mathbf{A}_j.$$

B-splines

B-spline curves are just pieces of Bézier curve ingeniously knotted together; whatever the hype, don't forget this. They are normally defined by a recursive de Casteljau-like formulation:

$$\mathbf{Q}(t) = \sum_{i=0}^{i=m} B_{i,k}(t) \mathbf{P}_i$$

where $\mathbf{P}_i$ are a set of points on what is called a *control track*, like the Bézier ones, and the interpolation is achieved by the B terms:

$$B_{i,k}(t) = \frac{t - t_i}{t_{i+k-1} - t_i} B_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} B_{i+1,k-1}(t).$$

This recursive definition terminates in the way you would expect, at $k = 1$:

$$\begin{aligned} B_{i,1}(t) &= 1, (t_i \leq t \leq t_{i+1}) \\ &= 0, (t < t_i) \\ &= 0, (t > t_{i+1}). \end{aligned}$$

The constant values of t, which are $t_i \ldots (i = 0, m+k)$, are the knots where the curves join, and the whole list of them is called the knot vector.

While the recursive definition is the simplest to understand, it is apparent that each span has a closed form either as a Bézier curve or in the power basis, and these may well lead to more efficient evaluation, because de Casteljau-type recursion formulae have an $O(n^2)$ performance. Note that the control-track points corresponding to a Bézier curve segment will not correspond to those for the B-spline as a whole, except for certain knot vectors, such as 00001111, where the whole B-spline is actually a (single cubic) Bézier curve.

Advantages of the Bernstein basis

There is no doubt that Bézier and B-spline techniques are an academic bandwagon of considerable horsepower; many people have had papers published, obtained professorships, and sundry other honours, from work on these topics. Nor can it be denied that they are the basis of most recently written curve and surface design software. But it would be surprising if that were not the case, given the strength of the literature. There are some discussions of advantages and disadvantages to be found in the literature (e.g. in the back of Farin's book); here is our own list.

Advantage 1: interacting using the control track

The original argument in favour of Bézier curves was a simple one. "You cannot design polynomial curves from their coefficients. If you try to interpolate through any but a very short series of points, interpolation techniques produce wavy curves which are useless; furthermore, you have to solve simultaneous equations to get them.

A Bézier curve can be designed by sketching its control track on a graphics screen; as you move the control track about, it pulls the curve in an intuitively acceptable way. Oh, and there are no equations to solve." It is on this basis that Bézier curves came to popularity during the Seventies. The advantage is still valid, but there are counter-arguments, which will follow.

Advantage 2: transforms

Bernstein-basis curves and surfaces have the useful property of being *invariant* under *affine* transforms[2]. This offers a considerable simplicity of system organization; it is only necessary to have one transform in the system: for points. And, although Bernstein-basis curves and surfaces are not invariant under the transforms corresponding to perspective projection, there is the possibility of a useful cheat (sorry, approximation) here, which avoids the necessity of working with the exact (necessarily rational) representations.

Against that, we note that there may be more work required to transform the control-track points of a B-spline curve than the coefficients of the same curve on a power basis. The Bézier form of our favourite, the cubic, has four control-track points, and the power basis four coefficients in each dimension:

$$\mathbf{Q} = \mathbf{A}_0 + \mathbf{A}_1 t + \mathbf{A}_2 t^2 + \mathbf{A}_3 t^3.$$

To rotate the cubic in either form requires two multiplications and an addition for every control-track point coordinate or coefficient: 16 multiplications and 8 additions, either way. However, to translate the cubic requires that every control-track point be translated: 8 additions; while only the coefficient $\mathbf{A}_0$ of the power basis ($\mathbf{A}_0$ is clearly the point on the curve at $t = 0$) needs to be modified to translate the curve to somewhere else.

Advantage 3: convex hull property

[2]Affine transforms are rigid-body motions (translation and rotation), scaling in one or more coordinates, and shearing. Invariant means that it doesn't matter whether you transform the points and then regenerate the curve or surface, or transform the curve or surface directly—you get the same resulting curve either way.

If a two-dimensional point (x, y) is defined as a weighted combination of a number of other points (x_i, y_i):

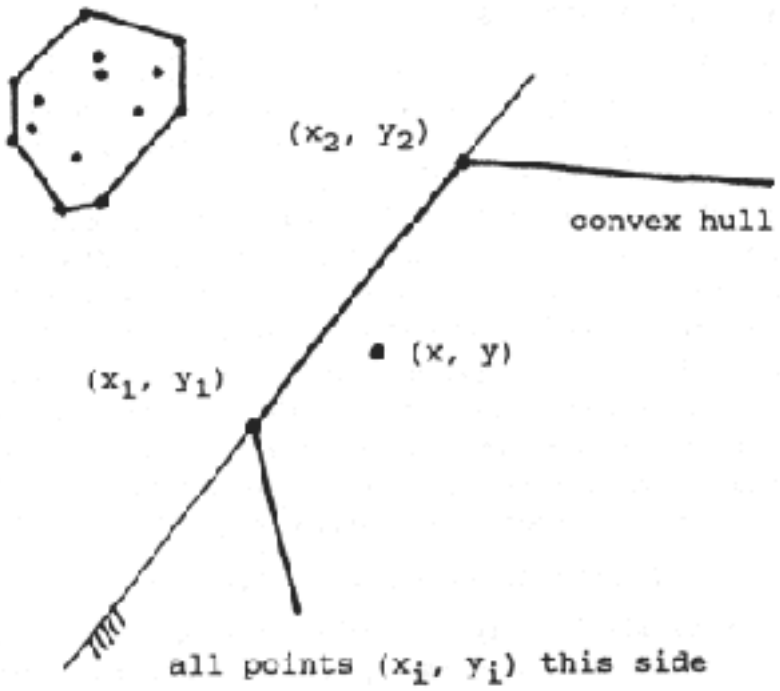
$$\begin{aligned} x &= w_1x_1 + w_2x_2 + w_3x_3 \ldots w_ix_i, \\ y &= w_1y_1 + w_2y_2 + w_3y_3 \ldots w_iy_i, \end{aligned}$$

where all the weights are positive, and they add up to one, then the point (x, y) must lie in the convex hull of the other points. What is the convex hull? Join all the points and take the outermost polygon; that's it. How do we know that (x, y) must lie within the convex hull? Suppose it lies on one of the edges of the hull, formed by the straight line joining (x_1, y_1) and (x_2, y_2). Then w_1 and w_2 will be the only non-zero weights. If we reduce w_1 or w_2, or both of them, and increase another weight, (x, y) must be pulled towards another point (x_i, y_i). But they are all on one side of the straight line (x_1, y_1)–(x_2, y_2), and so the new (x, y) is inside the convex hull; it can never get out of it, as shown in Illustration 4(iii).

This argument is slightly simplified; can you tell how? Answer: another point, such as (x_3, y_3), might be collinear with (x_1, y_1) and (x_2, y_2). That would allow more than two weights to be non-zero when (x, y) is on the edge of the hull. Indeed, any number of the points (x_i, y_i) might lie on one edge of the hull. That would mean that more weights might be non-zero, but it doesn't let (x, y) outside.

The coefficients of a Bernstein polynomial are all positive, and they add up to one. Therefore, we know that *the curve is constrained to remain in the convex hull of the control-track points* . This is extraordinarily useful, because in many applications we don't want to charge all over space looking for a tiny segment of curve or surface. We can use the convex hull as a containing volume, or box, or we can take the maximum and minimum coordinates of each control-track point and form a coarser but rectangular box, aligned with the coordinate axes. Even when our test fails, and the box is too large, we can continue to play the same trick by dividing the curve into two pieces, and forming a convex hull for each. Here is the formula for the control-track points in a new curve spanning the parametric interval $[0, t_s]$:

$$\mathbf{P'}_j = \sum_{i=0}^{i=j} \frac{j!}{(j-i)!i!} t_s^i (1-t_s)^{j-i} \mathbf{P}_i.$$



4(iii)—The point (x, y) cannot escape from the convex hull formed by the points (x_i, y_i), because of the way its coordinates are defined as a sum of theirs. (An example *convex hull* is inset top left.)

We can continue subdividing and such recursive subdivision algorithms are common: especially when the numerical alternatives are tough. Although the convex-hull property is well-known and attractive, it is not immune from criticism:

> There are other convex combinations of points that can have smaller convex hulls, for some curves, than the B-spline control track. We mention this just to emphasize that this property is not exclusive to B-spline curves and surfaces. In other respects alternative convex combinations are not so attractive.
>
> Other schemes for bounding curves and surfaces—both parametric and implicit—are available; we will mention them later. They may or may not be either tighter or easier to compute with than convex hulls of control-track points.

Advantage 4: numerical stability

A proper analysis of the numerical stability of the Bernstein basis (see Farouki and Rajan's 1987 and 1988 papers) is (miles) beyond the present scope. In general, it suffices to say that the Bernstein basis is less prone to magnify errors in its coefficients, both in evaluating points on a curve or surface, and in finding roots of the polynomial, which is required when we come to try to calculate intersections etc. However, this advantage is rather obviously lost if a conversion from the power basis is necessary. To preserve it, all algorithms must be performed in the Bernstein basis; this is not yet a perfected art.

Let's try to give an inkling *why* the Bernstein form is more stable. A simple measure of numerical stability is to see what happens when the coefficients of two representations of the same polynomial are perturbed; this will certainly happen in practice, either due to the inaccuracies inherent in the floating-point representation itself, or a series of floating-point operations performed on them, such as a transform. Suppose the Bernstein coefficients (of a single polynomial—forget vector values for a moment) are p_i and those of the power form are a_i. Given a perturbation of size ϵ to each coefficient, then the changes in the polynomials are simply:

$$|\delta q| = \sum_{i=0}^{i=m} \left| \frac{m!}{(m-i)!i!} t^i (1-t)^{(m-i)} p_i \right| \epsilon$$

in the Bernstein form, and:

$$|\delta q'| = \sum_{i=0}^{i=m} |t^i a_i| \epsilon$$

in the power form. Our ability to show the superiority of the Bernstein form essentially derives from the formula for converting from Bernstein to power coefficients, given at the beginning of this section. The Bernstein coefficients can all be obtained by summing *positive* factors multiplied by the power form coefficients, but not vice versa. So, if we substitute into the Bernstein form above, we are able to show:

$$|\delta q'| = |C \sum_{i=1}^{i=m} a_i|$$

while

$$|\delta q| = C \sum_{i=0}^{i=m} |a_i|,$$

(where the big Cs hide a lot of detail.)

It should be intuitively obvious that the sum of the moduli of a lot of quantities is always as big or bigger than the modulus of the sum of the quantities. This is called the triangle inequality [3]. Using it here, we can show that $|\delta q| \leq |\delta q'|$. The practical effect of this observation clearly depends on how large the coefficients of opposite sign are in a power from equation. That can vary (i.e. get worse and worse) as we move away from the origin. In practice, the accuracy of all geometry diminishes away from the origin, as any error becomes bigger and bigger along with the coordinate values. A power-basis polynomial exacerbates this problem by taking the difference between functions of these numbers.

Advantage 5: built-in approximations

Degree elevation is adding extra terms to a polynomial which have a higher degree than the original. In the power basis, the idea of degree elevation does not really exist; there is neither a reason for, nor a problem in, adding a term $0t^{n+1}$ to a polynomial of degree

[3] So called because of the observation that any two sides of a triangle are together longer than the third.

n. In the Bernstein basis, degree elevation is of interest, because all the coefficients change. In the case of a curve, the new control track corresponding to the elevated polynomial is a closer approximation to the curve—which itself remains unchanged. The degree elevation formula is:

$$\mathbf{P'}_i = \frac{i}{n+1}\mathbf{P}_{i-1} + \left(1 - \frac{i}{n+1}\right)\mathbf{P}_i.$$

This has a nice geometric significance, as each new $\mathbf{P'}_i$ lies on one of the edges of the original control track. Such artificially elevated polynomials can of course have their degrees reduced without changing their value (although not without loss of accuracy). More generally, of course, degree *reduction* cannot be performed exactly. In the power basis, there is no obvious way to do this. We can endeavour to invert the degree elevation process (see Farin's book—yet again—for more detail: page 55), but the results are unpredictable. The geometric interpretation of the Bernstein coefficients seems little help; other, more sophisticated techniques involve conversion to another base (see Watkins and Worsey's 1988 paper). This is therefore a minor advantage.

Advantage 6: a de facto standard

One of the problems with standards is that their existence can blight the development of alternative techniques. With the tropical riot of new parametric techniques that keep springing up in the literature, this does not seem to be happening.

Whether it's desirable or not, Bernstein polynomial techniques—and particularly the rational B-spline[4]—are now the standard building blocks for representing 'sculptured' geometry in computer-aided design systems.

---

[4] A rational B-spline is one B-spline divided by another—see Chapter 8 for the formula. A NURBS is a non-uniform rational B-spline; *non-uniform* means that the knots are not spread at integral parametric intervals. Like any acronym ending in 'S', NURBS sounds like a plural, but works both as a singular and plural noun, and as an adjective—you get "this is a NURBS", "these are NURBS", and "this is a NURBS curve". As with most jargon, this has the regrettable effect of excluding the non-cognoscenti.

Disadvantages of the Bernstein basis

Disadvantage 1: expensive to compute

The de Casteljau construction for evaluating Bernstein polynomials is elegant, and has excellent numerical properties. Unfortunately it shows $O(n^2)$ growth in running time with degree of polynomial. Even using Horner's method (as mentioned in Farin's book: page 47), evaluating Bernstein polynomials directly is still not as fast as can be achieved with the power basis. And the B-spline formulae are themselves overly complicated for simple curves and surfaces such as circles or spheres. We'll say no more here; there are some concrete examples in Chapter 13.

Disadvantage 2: difficult to understand

You should be able to make up your own mind about this by now! In general, maybe it is the major problem with the Bernstein approach, especially among non-mathematicians (join the club). Most people start learning about Cartesian geometry with explicit formulae, come across implicit equations, and finally meet parametric geometry when it becomes necessary to describe curves in three dimensions. The Bernstein basis itself is an over-elaborate way to describe a straight line (see Chapter 13), but if you wanted to change the world, you could make sure that the first equation for a straight line that anybody learned was $\mathbf{P} = (1-t)\mathbf{P}_0 + t\mathbf{P}_1$, rather than $y = mx + c$, $ax + by + c = 0$, or even $x = x_0 + ft, y = y_0 + gt$.

Disadvantage 3: unacceptable control technique

The control of Bézier and B-spline curves and surfaces using control-track and control-mesh points—however influential in the initial development of the method—is not now the primary reason for their continuing use. Why should the Bézier control track be a particularly intuitive 'handle' on a curve? It just drops out of the mathematics, rather than being designed with any 'ergonomic' insight or experiment. It is efficient in computation, but 25 years on from Bézier's original paper, that argument is as strong as the one that says we should always program in assembly code "for efficiency". So, today, shouldn't we expect to sketch a curve, or mould a surface

with a data glove, and get the computer to make the *actual* curve or surface that we've described? The answer must be yes.

Of course, you can say that the Bézier control track is now a traditional tool, which many people have grown to understand. But computers have a way of eating up traditional tools, and their own progeny are no exception. If the development of user-interface techniques is arrested by an outdated linkage to a particular algebraic representation, then that is one of its disadvantages.

This chapter is concluded with a C procedure to compute a point on a B-spline at a parameter value t. The control track is held in the array `track[m]`, and the knot vector in the array `knots[]`. The `set_weights` procedure is called once to fill the array `weights[m]`, then `b_spline` is called for each coordinate direction. These procedures do not check if there are enough knot values stored in `knots[]`. There should be $m+k+1$ of them where there are $m+1$ control-track points (first is 0), and the degree of the curves is to be k.

```
float b_spline(track,weights,m)
float      track[],weights[];
int  m;
{
     float p;
     int   i;

     p = 0.0;

     for (i = 0; i <= m; i++)
          p = p + track[i]*weights[i];

     return(p);

} /* b_spline */


/*
     Procedure to set up a list of weights, one for
     each track point.
*/

void set_weights(t,k,m,knots,weights)
float      t;
int        k,m;
float      knots[],weights[];
{
     float wt();
     int   i;

     for (i = 0; i <= m; i++)
          weights[i] = wt(t,i,k,knots);

} /* set_weights */

/*
     Recursive procedure to compute a weight value for
     one control-track point.
*/
```

```
float wt(t,i,k,knots)
float      t;
int        i,k;
float      knots[];
{
     float a,b,c,d;

     if(k == 1)
     {
           if((t >= knots[i]) && (t < knots[i+1]))
                 return(1.0);
                 else return(0.0);
     } else
     {
           a = (t - knots[i])*wt(t,i,k-1,knots);
           b = knots[i+k-1] - knots[i];
           c = (knots[i+k] - t)*wt(t,i+1,k-1,knots);
           d = knots[i+k] - knots[i+1];

           /* Avoid division by zero */

           if (b == 0.0) a = 1.0; else a = a/b;
           if (d == 0.0) c = 1.0; else c = c/d;

           return(a+c);
     }

} /* wt */
```

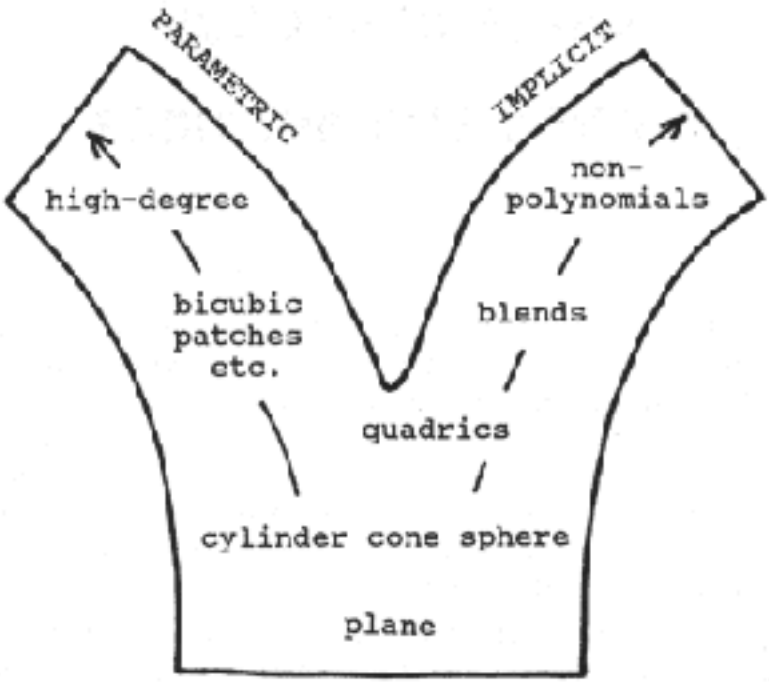
5

General implicit curves and surfaces

The great advantage of parametric curves and surfaces is that we can bound them simply by putting limits on parameter values; their great disadvantage is that we cannot easily find out on which side of a parametric curve or surface a point lies. The situation for implicits is reversed. It is not easy to bound them; we must in general provide additional geometric elements. On the other hand they are naturally half-spaces, and finding out on which side of implicits an arbitrary point lies falls out of their equations.

So far, the only implicit equations we have seen are those of the straight line and the conics in two dimensions, and of the plane and quadrics in three dimensions. The value of these curves and surfaces hardly needs elaborating; and anyway they all have parametric forms (albeit rational ones in some cases). So where do more complicated implicit curves and surfaces come from?

> Some shape representations are based exclusively on high-degree implicit equations.
>
> We may wish to have pieces of complicated surface in a shape model in which inside/outside calculations are very important (e.g. a solid model). If all the simple surfaces are manipulated as implicits, it makes sense to use implicit forms for the other bits, if we can. (Note that the 'engineering' surfaces are quadrics, and so with implicits we do not need to get involved with rational equations. This is a considerable simplification: and just as well, since it is more difficult to guarantee a non-zero denominator when the function is not localized to a parametric interval.)



5(i)—This is an informal illustration of the way in which parametric and implicit geometric elements cease to be interchangeable as complexity increases.

Implicits are useful in the context of *volume data* (from computer tomography (CT) scans and so on) where an object is defined as a density function at a number of points; this is a *de facto* implicit form, and thus it makes sense to fit implicit equations to this sort of input, or conversely to derive point data of this sort from an implicit equation.

Implicit curves and surfaces may fall out of calculations; for instance, the curve of intersection between a parametric patch and an implicit surface is naturally an implicit curve *in patch-parameter space*. Further, the degree of this particular curve is the product of the degrees of the original surfaces, which can be high enough to make it complicated, even if the original surfaces were simple.

The one place from which—in practice—we rarely get high-degree implicits is from implicitization of parametric surfaces [1]. Attempted implicitization (sounds indictable) leads to very large equations indeed, *and* curves and surface may intersect themselves: as you see in Illustration 5(ii). That's very difficult to deal with when the bounding provided by a parametric interval is taken away.

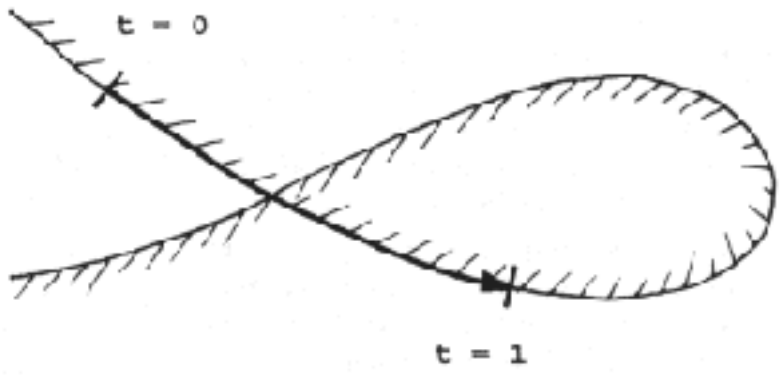
Blends

Many high-degree implicits may be viewed as *blends*: that is, a combination between a number of simpler elements—see, for example, Illustration 5(iii). The first of these blending ideas was introduced by Ricci (in his 1973 paper). He took a number of implicit equations $f_i(x, y, z)$, where each of these might be a sphere, cone, or whatever, and combined them using the formula

$$f' = (f_1^{-w} + f_2^{-w} + \cdots + f_n^{-w})^w.$$

This equation generates an approximation to the set-theoretic *union* of the original shapes: in fact a new surface that is a 'bag' containing all the original shapes. As the value of w is increased, the bag shrinks

[1] Though it's always possible to do this; the reverse operation, parameterization (going from implicit to parametric), is not always possible.



5(ii)—A parametric curve segment (the thick line) which itself is perfectly well-behaved may nevertheless contain a self-intersection in the corresponding implicit curve.

on to the objects. (Ricci also gave a similar formula to compute an approximation to the region shared by all the shapes—their set-theoretic intersection—but this is beyond our present scope.) In this formula, the effect of each f_i is felt over all space, albeit diminishing quickly.

Later (in 1982) Blinn came out with a variation on this technique for molecular modelling, where each f_i is a sphere equation. Blinn used exponentials, and he pragmatically neglected very small values as the exponentials decayed away from each spherical 'atom'.

A more structured way of dealing with the same problem is the version of Ricci's blend due to Rockwood and Owen; in essence, their blend is:

$$\begin{aligned} f' = \quad & [\quad \max((1-f_1),0)^w \\ & + \quad \max((1-f_2),0)^w \\ & + \quad \vdots \\ & + \quad \max((1-f_n),0)^w \;]^{\frac{1}{w}} . \end{aligned}$$

The Wyvills' 'soft object' system (see Wyvill, McPheeters and Wyvill's 1986 paper, for instance, which is only one of a number of their publications on the subject) avoids variable exponents. The

functions which they blend together are points (i.e. spheres of zero radius $f_i = (x - x_i)^2 + (y - y_i)^2 + (z - z_i)^2$) and they combine them with the equation:

$$f' = \sum_1^n \left(-0.44 f_i^3 + 1.89 f_i^2 - 2.44 f_i + 1\right).$$

This polynomial is an approximation to a cubic in distance (rather than squared distance), and becomes zero at $f_i = 1$.

There is a good analogy with splines in these developments: the search for localized effect of each data value (each implicit polynomial being blended together, in this case), together with economy in computation. Another interesting observation is that these formulations actually behave quite well, despite being potentially very high degree (indeed, only the Wyvills' formula is actually a polynomial). Nor is there a high storage requirement; there is no necessity to store all the terms that would be associated with a general implicit of the same degree.

Instead of controlling the blend surface by range, we can use yet more implicits as bounding geometry. In the Liming formulation,

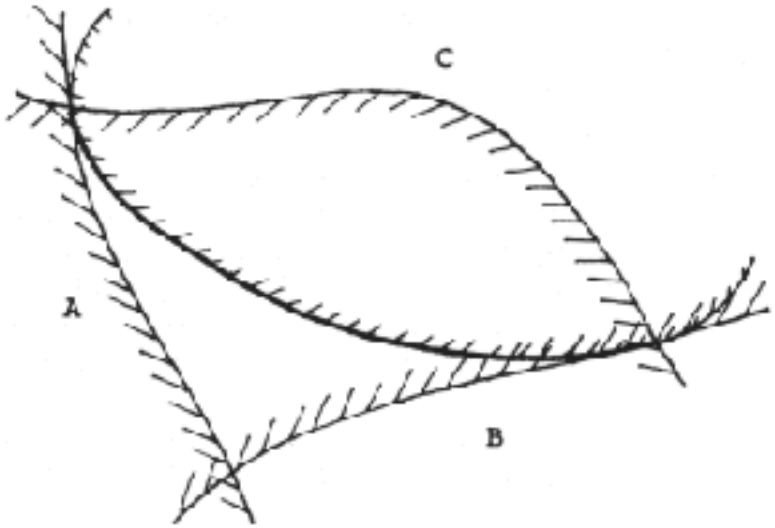
$$f' = (1 - u)(f_1 f_2 \cdots f_m) + u(g_1^2 g_2^2 \cdots g_n^2),$$

the resulting surface f' runs between all the curves where the surfaces f_i meet the surfaces g_j. What's more, because the terms g_j are squared, the surface generated is tangent to the surfaces f_j at those curves. As you see in Illustration 5(iii), the set-theoretic intersection of half-spaces corresponding to the surfaces f_i and g_j can be used to bound the blend:

$$(f_1 \leq 0) \cap (f_2 \leq 0) \cap \ldots \cap (g_n \leq 0).$$

By swapping any or all of the '$\leq$'s for '$\geq$'s, the sets that the inequalities define are complemented, and the region in which the blend is confined is changed. This line of thought leads (yet again) to solid modelling topics beyond our scope.

There are quite a few more blends which will produce high-degree implicit equations. Furthermore, you will notice that not very much has been said about the terms f_i which we have been gaily blending together. In fact we can blend blends, and indeed produce as complicated equations as anyone might fancy.



5(iii)—A two-dimensional Liming blend (the thick curve) is formulated as $(1-u)AB+uC^2$, so that it blends curves A and B, and is controlled by C. The constant u has a value between 0 and 1 and is used to control the shape of the blend. As u approaches 0 the blend gets closer to A and B; as u approaches 1 the blend gets closer to C. The extent of the blend can conveniently be bounded by the set-theoretic intersection of half-spaces formed from A, B and C:

$$(\mathrm{A} \geq 0) \cap (\mathrm{B} \leq 0) \cap (\mathrm{C} \leq 0).$$

Some properties of implicit equations

It is in theory easy to detect when a point lies on an implicit curve or surface; if the point coordinates are substituted into the equation, the result is zero. In practice, of course, some fudge factor is unfortunately inevitable, and choosing its value is not necessarily easy.

We can also find the normal to the curve or surface by partial differentiation. If $f = 0$ is a implicit surface, the normal (not a unit vector) is the *grad* of the surface:

$$\bigtriangledown f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right).$$

As will be hammered home in Chapter 11, only the straight line and plane yield distance if a point is substituted into their equations. The circle and sphere give squared distance; we can think of them as nothing more than particular contours of distance from a point. Although the result of substituting the coordinates of a point into an implicit equation is not the distance to that point, it *is* a value that *in general* increases as we get further from the curve. We call this a *penalty function*. The relationship between penalty function and distance is a slippery one. Just by thinking of the penalty function inside an ellipse, you can see that you can stay on a curve of constant penalty function, and yet get nearer to and further from the actual surface. If we evaluate the partial derivative of the equation for a point off the surface, we get a normal to the penalty function contour. Following that normal produces a curve of 'steepest descent'. However the normal to the penalty function is a local property, and so the curve of steepest descent is not quickest route to the curve or surface: which is hardly surprising, since it's a curve.

Using hill-climbing methods[2] with penalty functions is often an effective way to deal with implicit curves and surfaces, but it can be difficult to show that methods will *always* work. For instance,

[2]One might imagine that 'methods of steepest descent' were techniques of a diametrically opposite sort; in fact they are the same; the choice between 'ascent' and 'descent' is notional, just involving one change of sign.

Illustration 5(iv) shows a situation in which moving uphill can lead to a local maximum of the penalty function.

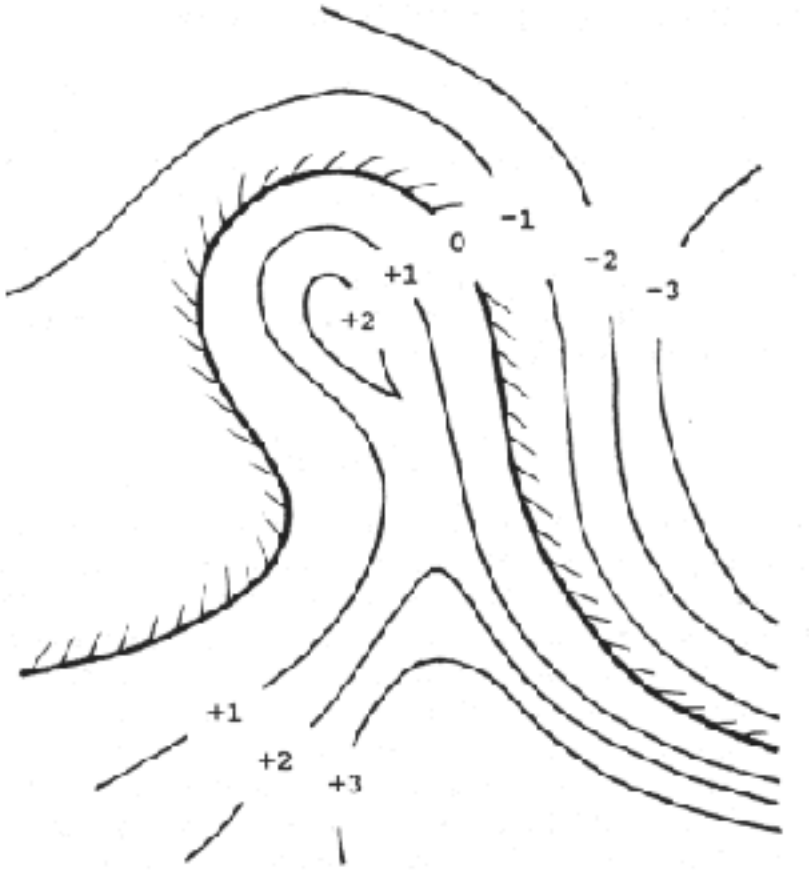
In some cases, we want the penalty function to behave like true distance in a particular locality (e.g. the region in which the curve of surface is actually forming part of the model of an object). Obviously, we can multiply the whole equation through by a constant, so that the evaluation of the product *at the point of interest* really does match true distance (if we know what that should be). We can further improve the resemblance of the penalty function at that point to a genuine distance function by making more Draconian modifications to the original function. The crude measure of taking the nth root of a degree n polynomial, for instance, can reduce the magnitude of the derivative nearer to unity.

We can also multiply through by an auxiliary polynomial (e.g. a plane) and in this way get both magnitude and direction of the slope to match a true distance function *locally*.

But be *very careful* about this sort of thing; it tends to cause other problems, especially in regions away from the magic point. Taking a root, for instance, means considerably more calculation, negative values need special handling, and in some regions the penalty function may become less like distance. For instance, if the polynomial is $x^3 - y$, and we take its cube root, obviously values near the y axis will become more like distance, but values near the x axis will become less like true distance. And if we multiply by another polynomial, we are actually introducing another piece of surface; the function is zero on the auxiliary polynomial as well as on the original curve or surface. Usually, we will need to be very sure that the auxiliary polynomial does not enter any region of space in which we are actually interested.

Bernstein-basis implicit curves and surfaces

When we *are* only working within a particular region, one option that we have is to use the Bernstein form of the curve or surface. It's often not appreciated that implicit polynomials *have* a Bernstein form. They do (see Patrikalakis and Kreizis' 1989 paper); it is defined within a rectangular or cuboidal region, orthogonal to the axes, and is formulated in terms of the corner vertices and a set of



5(iv)—A local maximum in the penalty function near an implicit curve; the contours correspond to values of $f(x, y)$.

weights. Here is the two-dimensional version, for the region $x_0 \leq x \leq x_1, y_0 \leq y \leq y_1$:

$$f = \sum_{i=0}^{m} \sum_{j=0}^{n} w_{ij} B_{i,m}(x) B_{j,n}(y),$$

where

$$B_{i,m}(x) = \frac{m!}{(n-i)!i!} \left(\frac{x - x_0}{x_1 - x_0} \right)^i \left(\frac{x_1 - x}{x_1 - x_0} \right)^{m-i}$$

and similarly for y.

We can even manipulate the curve (or surface) using the weights w_{ij}; but it's not really the same interactive tool as the Bézier control track; further, it doesn't have the same advantage in terms of transforming the curve or surface. For, if we rotate it, the cuboid comes out of alignment with the axes, and we have to choose a new region of interest.

6

Tessellations

Suppose you had 100,000 different numbers stored in a computer, and you wished to find the one closest to 5.736. To be sure of getting the right answer you would have to compare all 100,000 of them with 5.736, which would be tedious. If you had a million numbers like 5.736, and had to find the closest numbers to all of them, that would be a million times as tedious. What you would do then is to *sort* the 100,000 numbers into order (using a fast algorithm like Shellsort or Quicksort—see Knuth's *The Art of Computer Programming*), and use binary searches (Knuth again) to find the million numbers. This would only be $\log(100,000)$ tedious.

The vast bulk of the geometry that we want to do is in two or three dimensions. If we are dealing with complicated geometrical constructions involving many simpler elements, we often want to search amongst those elements to find, say, the nearest ones to a given collection of points. As with the numbers, we obviously want to avoid examining all the elements for every one of our points. Is there any way that we can 'sort' geometry to allow us quickly to find an element of it near to a given position?

At first sight, it seems that the answer to this question must be no. The reason for this is that there is a natural ordering of numbers along a one-dimensional axis which makes the idea of sorting sensible, but that there is no natural ordering of, for example, points dotted about in a two-dimensional plane. We could, of course, attempt to sort our points in x, which is, in effect, to project the problem down from two dimensions to one; but if there are a lot of coincidental x values (as there will be if our data points are on

some sort of grid, or come from an engineering drawing) this won't help much. We could then do a subsequent y sort (and, in three dimensions, a z sort) and, for some applications, such as data-driven divide-and-conquer algorithms (see Preparata and Shamos' *Computational Geometry*), this is the approach adopted.

However, it is also possible to use one of a variety of *tessellations*, and it is these that this chapter is about.

The word *tessellation* comes directly from the Latin *tessellare* which means 'to form from many *tesserae*'. A *tessera* is a small tile, such as might be used in a mosaic. A tessellation, therefore, is a pattern of polygons that all fit together without gaps and fill an area of the plane. The idea extends up into any number of dimensions easily; in three dimensions, for example, a tessellation becomes a pattern of packed polyhedra filling a region of space.

In fact, the tiles do not even need to be polygons or polyhedra; we can have shapes with curved sides, as long as they all fit and leave no gaps.

Regular tessellations

Rectangular grids

The simplest tessellation is a regular grid of boxes like a sheet of graph paper. This is easy to keep in a computer; all you need to store are the coordinates of any arbitrary *vertex* (that is, box corner) of the grid, and its *pitches* (that is the length of the sides of the boxes in the x, y and, if needed, z directions). This presumes that the axes of the grid are to be aligned with these coordinate directions which, for the vast majority of applications, they will be. If they aren't, for some reason, the angle that the grid's 'x' direction makes with the real x direction must also be recorded.

In the case of a grid, our tiles are all congruent rectangles, and the structure, once defined, extends to infinity in all directions. Usually, of course, we are only concerned with a bit of it.

We now have a method of attacking the problem posed in the Introduction: how to find the geometrical element nearest to a given

point if we have a big collection of elements and don't want to look at each one to get the answer. This method of attack has been used by cartographers for centuries, and is best exemplified by a street map such as the *London A–Z*. The map-maker divides the map up using a grid and, for each rectangle in the grid, constructs a list of the streets that intersect it. The map-maker then prepares an alphabetical index of streets and, against each, lists the grid rectangles in which it appears. This is why you can find Oxford Street without starting in Hampstead and working slowly and methodically south and east looking at every street on the map.

The map in the example uses not one tessellation, but two. The pages form a coarse grid, and each page is divided into a finer one. If, instead of a street name, you had the x, y coordinates of a point, you could do a simple arithmetical calculation to find which grid rectangle in the *A–Z* it lay in, look on the right page at the right rectangle, and find what was nearest to your point by examining just that rectangle and, possibly, its immediate neighbours (if your point happened to fall near the edge of the rectangle). This was exactly the problem that we wished to solve. In fact, you may have to search the neighbours of neighbours, and so on. The terminating condition for the search is that it can ignore any tile the nearest corner of which is further from the search point than the nearest geometric entity already found. This still means that, in general, the vast majority of the tiles and entities will never be examined.

In a computer program the grid (or grids) would be defined as outlined above, and a big bounding rectangle-of-interest would also be defined (a big box round London, in our example). All the rectangular grid tiles in the big box would have a unique index in the x and y directions. Each tile would have a list associated with it (probably kept on a *heap*—see Knuth's book again), and in that list would be a set of pointers to all the geometrical elements that intersected that particular tile. Not forgetting the need occasionally to examine tiles that border on one we are interested in, we could then find the nearest geometrical element to a given point without examining all the elements. The way that the lists would be constructed would depend upon how the elements were generated. In general, the best idea would be, whenever a new element was added, to calculate the grid tiles which it cut, and then to amend their list

appropriately. Depending on the application, it may be a good idea to keep lists for each geometrical element of the tiles they intersect. This makes things easy if, for example, an element is to be removed from the structure.

What has been achieved by the grid tessellation is not analogous to a true sort in one dimension, but is like a bucket-sort of the type needed to construct a histogram. The rectangular tiles are the buckets.

Non-rectangular regular tessellations

There are many regular tessellations other than those composed of congruent rectangles. Their use for the problem of data-ordering and data-searching is less common but, when the data are of a highly ordered but non-rectangular form, they are sometimes useful. The simplest non-rectangular regular tessellation is a tiling of identical equilateral triangles. Here only one side length and vertex position needs to be stored (along with an angle if the pattern is to be inclined to the axes). Embedded within this tessellation is one of congruent regular hexagons. It is not possible to tile the plane with regular polygons of more than six sides, as their internal angles are greater than 120°. This means that it is not possible to cram three (let alone more) such angled corners together, as the (non-)tessellation would contain gaps or overlaps.

It is also possible to tile the plane with any scalene triangle, as long as you allow yourself to turn it upside down half the time. This is a consequence of the more obvious fact that you can rule parallel straight lines across the plane in two directions, thus constructing a tessellation of any size and shape of congruent parallelograms. Cut all the parallelograms in half along a diagonal (a third set of parallel lines) and you've got your triangles.

If you allow yourself non-convex shapes, or two (or more) congruent shapes acting in tandem, or both, you can make yourself all sorts of regular Escher wallpaper tessellations. The way to experiment with these is to start with an easy tessellation and then to divide up all its elements in an identical way to make a more complicated one. This is also the way to store it in the computer.

Adaptive tessellations: quad-trees and oct-trees

The *London A–Z* works efficiently because in London the streets are about the same distance from one another. Suppose you extended the idea to cover the whole of the British Isles. The grid squares (which, in London, each contain perhaps forty or fifty streets) would now mostly be in open countryside. Indeed lots of them would contain little more than the odd rock—covered at high tide—because significant fractions of the area covered are sea. This might not matter; we could compile a list of the tiles that contained useful data and the list would be much smaller than all the tiles in a rectangle surrounding the country. But suppose we wanted to go down to a finer level than the streets in cities, and record lamp-posts, litter bins, and phone boxes? The city grid would have to be smaller, and even more grid is wasted in sea and moor and lake. Things are beginning to get out of hand.

In fact, the first sentence of that paragraph is a fib. Streets in Central London are much denser than in the suburbs. To accommodate this the *A–Z* has two scales of grid, one for the West End, and one for the rest of Greater London. The central one is much finer, and the maps there are plotted to a bigger scale.

There are many situations in which we would like to use our grid idea to bucket-sort geometrical elements in two or three dimensions, but are put off by the uneven distribution of the elements in space. Consider the straight lines, arcs, and so on that are the geometrical elements of an engineering drawing in a draughting system. Most real engineering products have small volumes of intricate complexity surrounded by comparatively large regions where not much is going on at all. What is needed is a grid which, like the two-tier *A–Z, adapts* itself to the local density of the geometrical elements that are to be put into the buckets. The usual tessellation employed for this is a tree of recursively-divided rectangles.

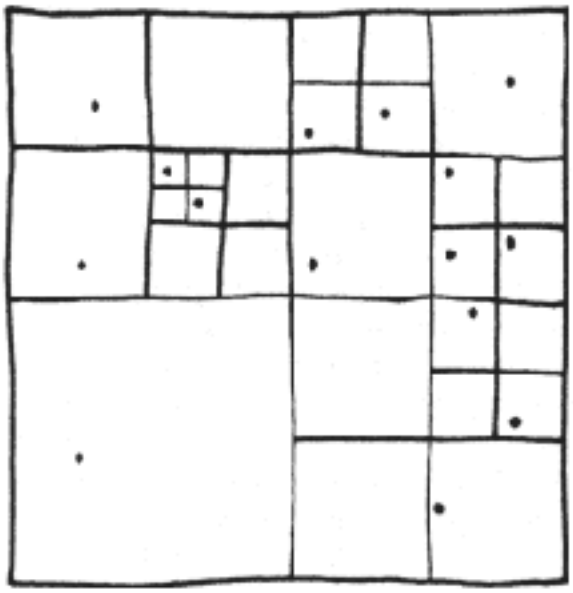
Suppose, to make things explicit, that we are concerned with phone boxes all over the British Isles, and that there are a million of them. What we might do is to start with a single square surrounding the whole of Britain and Ireland a thousand miles along its side. We would then write a recursive procedure that would take a square of side length l containing n phone boxes and do one of two things.

If n were less than some predetermined small value (say 10), the procedure would do nothing but return. Otherwise, it would divide the square into four smaller squares of side length $\frac{l}{2}$, and bucket the phone boxes into those four smaller squares; it would then call itself recursively for each of those four squares and their contents.

If the phone boxes were roughly evenly distributed, then each $\frac{l}{2}$ square would get about $\frac{n}{4}$ phone boxes. But if one square contained a city with many phone boxes, and the other three squares were in countryside and contained less than our minimum ten phone boxes each, only the city square would get further divided. In this way the division would adapt itself to the local density of the geometrical elements (point coordinates of phone boxes in this case) that we wished to bucket. Cities would be finely divided, and the countryside and sea would have a few big squares covering them. Each parent square that is further divided has four child squares that compose it; the natural data structure to represent this is a tree. The undivided squares containing ten or fewer phone boxes would be its leaves; the big square round the British Isles would be the root. The whole structure is known as a *quad-tree*; Illustration 6(i) shows one.

If we label the four child squares of each parent in a consistent order (top left, top right, bottom right, bottom left, say), then all we need to specify the size and position of any given square is the path down the tree to it from the root, and the size and position of the root square. As we walk down the tree we keep a representation of the square we are currently in, and update it each time we move down into a smaller square; two of the smaller square's sides will be new, two will derive from the parent.

This is also the way in which we search for the square containing a given arbitrary point. The tree is walked as described, the child square containing the point of interest being chosen at each four-way branch. Note that if we are, for example, searching for the phone box nearest to the arbitrary point, then things are a bit more tricky than they were with the regular grid. Searching the square in which the arbitrary point lies is trivial, of course. But the nearest phone box may lie in an adjacent square. To find these we have to go back up the tree and walk down at least five more branches to find all the immediately adjacent squares. There is no guarantee that we won't



6(i)—A quad-tree divided down until each square contains at most one data point.

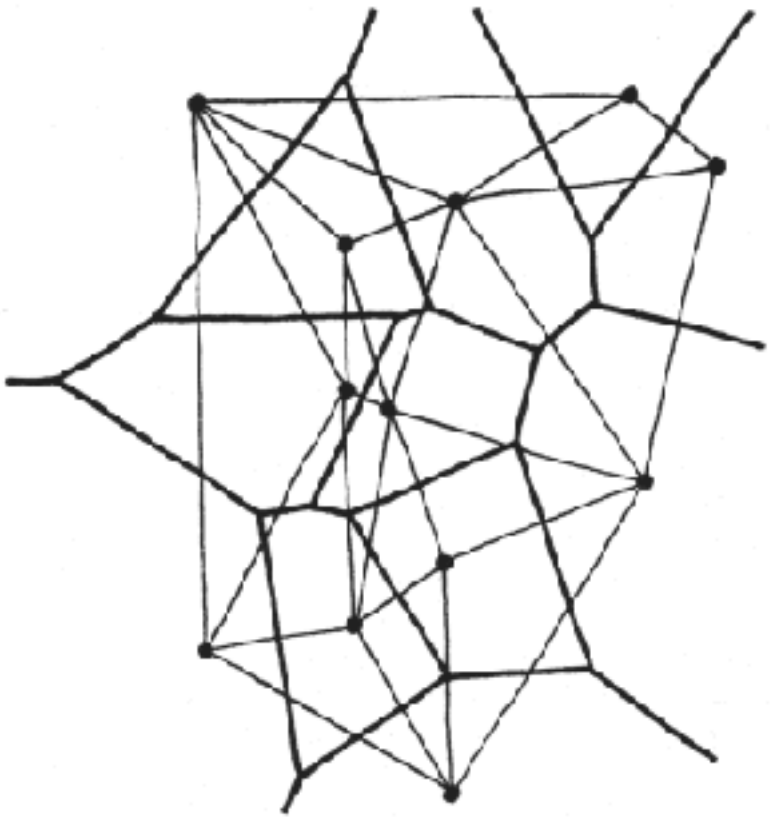
have to search further afield yet, as with the regular grid, and the same rules apply about finding a phone box nearer than the nearest corner of a square.

The quad-tree idea can be employed in three dimensions, where the squares become cubes, and are divided into eight. This is an *oct-tree* (see Samet's blockbuster books for full coverage of these tessellations).

One virtue of a quad-tree is that all its elements are similar. If we are prepared to abandon this we can do a binary (rather than a four-way) division. Here we start with an arbitrary rectangle and cut it in half at each stage of the division. The cut can be either length-wise or, more commonly as it prevents very long thin rectangles being built, width-wise. Indeed, there is no reason to cut the rectangle into two equal halves. In the case of the phone box example, if we knew that the majority of the phone boxes in a rectangle were at its right-hand end, it might be sensible to cut it three-quarters (or whatever) of the way along. This results in a pattern of rectangles that are all different; but it can be highly efficient as the freedom obtained by relaxing symmetry has allowed a tighter adaptation to the data. Of course, the coordinates of all the division planes have to be recorded, and whether the division is in the x or y direction, but this does not result in an inordinate amount of tree information to store.

A division scheme that is half-way between the quad-tree and the free-for-all just described is that used for the European standard 'A' paper sizes; the sides of standard sheets are in the ratio $1 : \sqrt{2}$, and smaller sizes are formed by dividing them recursively divided in half along their longer ($\sqrt{2}$) edges. The result is a sequence of similar rectangles divided alternately horizontally and then vertically. It allows the regularity of the quad-tree together with the simplicity of a binary tree (with only two children per parent).

All these schemes work just as well in three (or indeed any number of) dimensions as they do in two. They tend to take a time that is proportional to $n \log(n)$ (where n is the number of geometric elements that need to be bucketed) to do the division. Subsequent searching is proportional to $\log(n)$, as is the depth of the tree.



6(ii)—The Dirichlet tessellation and Delaunay triangulation of 12 data sites.

Dirichlet tessellations and Delaunay triangulations

The regular rectangular grid tessellation is *imposed* upon our data, the quad-tree and its derivatives *adapt* themselves to it, but the Dirichlet tessellation is entirely *derived from* the data.

Suppose the data consist of a map of some robins' nests. Suppose further that the robins (highly territorial birds) are all equally fierce in defending their nest sites. What shape would we expect their territories to have?

The answer is the Dirichlet tessellation (sometimes called the Voronoi diagram) of their nest locations. This gives each data site

(nest) a territory that is *the region of the plane nearer to it than to any other site*. Illustration 6(ii)[1] shows the pattern (bold lines) for twelve data sites. The fine lines in the figure are the *Delaunay triangulation*. This triangulation results from joining each pair of data sites that share a common tile boundary (that is, are territorial neighbours).

The territorial tiles are convex polygons. Their edges are the perpendicular bisectors of the Delaunay triangle edges joining neighbouring sites. The vertices of the tiles (where three meet) are the circumcentres of the Delaunay triangles. The Delaunay triangles completely cover the convex hull of the data sites; the Dirichlet tessellation extends to infinity—sites within that convex hull have finite territories, those on it infinite ones.

These two geometrical constructions have a wide variety of applications. It is not difficult to compute them, and many programs have been written to do so (see Green and Sibson's 1978 paper, and Bowyer's 1981 paper for efficient algorithms and data structures for calculating and storing them). The two constructions form a mathematical *dual*; if you know one, you can completely deduce the other—no extra information is needed.

Let us consider some properties of these two intimately related tilings (for proofs of some of the properties listed below, you are recommended to see Preparata and Shamos' book).

Nearest neighbours: the nearest-neighbour of each data site is always one of its Delaunay neighbours. This means that the tessellation makes it trivial to construct a nearest-neighbour list.

Circumcircles: the circumcircle round a Delaunay triangle contains no data sites other than the three at the triangle's corners.

Minimum spanning trees: the *minimum spanning tree* (MST) of the data sites is embedded in the Delaunay triangulation. It is the tree with the data sites at its nodes in which the sum of the lengths of the branches is as small as possible. It is the basis for some heuristic solutions to the well-known *travelling salesman* problem.

Optimum triangulation: The Delaunay triangulation is *locally equiangular*, by which is meant that, given the data sites, it is

[1] After Green and Sibson's 1978 paper.

> the nearest that one can get to a pattern of equilateral triangles using the sites as vertices. This is important for many applications, as calculations done on triangles (such as those in finite-element analysis) that are close to equilateral tend to be more numerically stable than those done on long thin triangles. In fact, triangles don't make good finite elements—numerical analysts prefer quadrilaterals—but they are often used for things like interpolation, so the property is still important.

Like the other tessellations that we have considered, the Dirichlet tessellation imposes an ordering of a kind on the data. It makes it easy to search for the nearest data site to an arbitrary point (the data site in whose territory the point lies is the one we want; we get there by walking along the Delaunay links from any data site, always choosing a link that takes us nearer to the point we're interested in). It *localizes* the data in a way that is entirely controlled by the data themselves.

Just as quad-trees lead to oct-trees, all the things that have been said about Dirichlet tessellations and Delaunay triangulations generalize into any number of dimensions. In three dimensions the territorial tiles become convex polyhedra and the triangles tetrahedra, and so on.

It is also possible properly to define territorial tessellations of geometrical entities other than sets of points. A pattern of infinite straight lines, for example, will have territories that are collections of triangles formed by the bisectors of the angles at the points where the straight lines meet. If we have straight-line segments, things get more complicated (in the limit, a line segment is a point so, if we can solve that problem, we get the Dirichlet tessellation of points more or less free). Things are more complicated because the territorial boundary that is equidistant from a point and a straight line is a hyperbola—our Dirichlet tessellation must now have territories with curved boundaries. The calculations needed to find these are difficult; essentially they are the same as those needed to compute offset curves (and, in three dimensions, surfaces); they are covered in Chapter 11.

7

Approximations

Approximations are necessary just about all over. To start with, many geometric problems have solutions that are algebraically infeasible. Even if we do have a notional exact solution:

It may be unacceptably slow.

It may require too much intermediate storage to implement (the 'intermediate expression swell' problem in algebra systems).

The results may be geometric entities in forms that cause problems in subsequent processes; i.e. we end up approximating later anyhow (a simple example is an algebraic solution which gives an exact result which is a high-degree polynomial that later we have to evaluate using floating-point arithmetic).

A floating-point implementation of the 'solution' is actually so inexact that an explicit approximation would be more accurate (we will deal with that sort of approximation in Chapter 12).

So, rather than continue fighting with the algebra, we often decide to approximate the geometry to elements which we *can* operate on exactly (or at least very accurately). Here are some questions to ask yourself when you are choosing an approximation:

What is the accuracy requirement of the application? In some robotics applications, calculations start from highly inaccurate sensor data; you will see exact algebraic solutions to what purport to be robotics problems discussed in the literature (like the *piano-mover's* problem—how to shift an irregular object through an obstacle course), but they are often wildly inappropriate for real applications.

If the approximation involves an increased number of simple elements, will the simpler calculations more than offset the growth in number of things to be processed? (This is a common problem with polylines and facets—see below.)

At what stage in the calculation shall we approximate? The macho approach is to delay approximations as long as possible; more often than not (e.g. in intersection calculations) early geometric approximation is more fruitful than late approximation of B-I-G algebra.

Will there be special cases for which the approximation is, or is not, required? Examples are natural quadrics in the sorts of special positions in which they frequently occur in mechanical parts etc., and for which simple solutions exist (e.g. a cone intersecting with an orthogonal plane to give an exactly circular intersection curve).

Shall we try to reconstitute a more complicated geometric structure from the approximation after processing? (For instance, putting a spline curve through a set of points.)

Shall we retain the exact structure within the approximate one?

This is a terrific list of questions; *some* of the answers are in the following sections, where we will look at a few of the tricks of the approximator's trade. Of course, some (many? most?) approximations are so widely used that they have become representations in their own right. Approximation is a blanket with many threads that need to be followed—a long way.

There are really two sorts of approximation. One is the obvious sort, in which we replace something by something simpler which more-or-less has the same geometric form. We may be able to control, or at least to understand, how great the inaccuracy is, or we may trust to luck and test-cases. In the other sort of approximation, we construct a bounding region, or enclosure, in which the original piece of geometry is *known to lie*, do some operations on it, and generate a region in which the result is *known to lie*. These approaches have their own characteristics:

Enclosures get more brownie-points from the mathematicians, who hate committing to approximations of unprovable validity.

It is easier to carry around the original geometry if we are using enclosures. If we simply approximate, say, a surface, we may find that something we have done to the approximation (e.g. offsetting it) cannot be reflected back on to the original geometry.

Enclosures are often slower than pure approximations, because we have to deal with regions rather than curves, surfaces etc.

The vagaries of floating-point arithmetic can chip away at the validity of the enclosure approach, but we may be able to remedy this by inflating the sizes of the enclosures; see the section on accuracy.

About the worst thing that happens with enclosures is that they grow so large as to be practically useless. If the enclosures contain other geometry, and if the code is correct, programs won't fail; they just run grindingly slow. This can be a difficult behaviour to predict and control.

The best enclosures are the tightest, and often something clever can be arranged for particular pieces of geometry. There is one general technique that we will now discuss.

Intervals

Rectangular boxes, or cuboids, are a popular sort of enclosure, and tests using enclosures are often called *boxing tests*, even when the boxes are not in fact cuboids. The box aligned with the coordinate axes is particularly popular:

It's easy to use, because it's bounded by planes of the form $x = x_{min}$, $y = y_{min}$ etc. The plane has a parametric and an implicit equation which are both simple enough to substitute easily into many other more complicated equations, giving a head start on intersection calculations.

One such box is easily unioned with another, even though the result (i.e. a box which encloses them both) may be rather large.

Axially-aligned boxes may be considered as *intervals* in the three coordinates.

The only disadvantage of rectangular boxes is that they do not conform well to certain sorts of geometry, and so:

They can easily grow rather large.

An interval is simply a range of values of something; so $[3.0, 4.0]$ means all the infinitely many real numbers between 3.0 and 4.0. (There is a fussy distinction in real mathematics about whether an interval contains its end-points or not, but since we are usually over a floating-point barrel, this is not normally significant.) So an interval in both x and y, i.e.

$$\begin{aligned} x &= [x_{min}, x_{max}] \\ y &= [y_{min}, y_{max}], \end{aligned}$$

defines the sort of axially-aligned box we've been talking about, in two dimensions. We'll keep everything in two dimensions, to make an example drawable—but one of the great flexibilities of intervals is that they're applicable to any number of dimensions, as well as all sorts of different types of equation.

So how do we use it? Well, if we have an interval of two variables, and the variables are combined algebraically in some way, we can work out the interval on the result. (In draft, the operations below were written out using mathematical notation. Showing how the operators might be implemented in PROLOG is a little more interesting, and actually doesn't look too different. The layout follows Moore's book, in case you don't care for PROLOG.)

```
/* Interval addition: int_add(A,B,A+B) */

int_add((Alo,Ahi),(Blo,Bhi),(Alo+Blo,Ahi+Bhi)).

/* Interval subtraction: int_sub(A,B,A-B) */

int_sub((Alo,Ahi),(Blo,Bhi),(Alo-Bhi,Ahi-Blo)).

/* Interval multiplication: int_mul(A,B,A*B) */

int_mul((Alo,Ahi),(Blo,Bhi),(Alo*Blo,Ahi*Bhi))
     :- 0 <= Alo, 0 <= Blo.
```

```
int_mul((Alo,Ahi),(Blo,Bhi),(Alo*Bhi,Ahi*Bhi))
     :- Alo < 0, 0 < Ahi, 0 <= Blo.
int_mul((Alo,Ahi),(Blo,Bhi),(Alo*Bhi,Ahi*Blo))
     :- Ahi <= 0, 0 <= Ylo.
int_mul((Alo,Ahi),(Blo,Bhi),(Ahi*Blo,Ahi*Bhi))
     :- 0 <= Alo, Blo < 0 , 0 < Bhi.
int_mul((Alo,Ahi),(Blo,Bhi),(Alo*Bhi,Alo*Blo))
     :- Ahi <= 0, Blo < 0, 0 < Bhi.
int_mul((Alo,Ahi),(Blo,Bhi),(Ahi*Blo,Alo*Bhi))
     :- 0 <= Alo, Bhi <= 0.
int_mul((Alo,Ahi),(Blo,Bhi),(Ahi*Blo,Alo*Blo))
     :- Alo < 0, 0 < Ahi, Y <= 0.
int_mul((Alo,Ahi),(Blo,Bhi),(Ahi*Bhi,Alo*Blo))
     :- Ahi <= 0, Bhi <= 0.
int_mul((Alo,Ahi),(Blo,Bhi),(Clo,Chi)
     :- Alo < 0, 0 < Ahi, Blo < 0, 0 < Bhi,
        min(Alo*Bhi,Ahi*Blo,Clo),
        max(Alo*Blo,Ahi,Bhi,Chi).
/* Purely a precaution:- */
int_mul(_,_,_) :- write('int_mul: unknown error'),
                  !, fail.

/* Exponentiate: positive integer N only */

int_exp(A,N,C)
     :- N < 0, write ('int_exp: negative exponent'),
               !, fail.
int_exp(A,N,C)
     :- N mod 1 < 0, write ('int_exp: real exponent'),
                     !, fail.
int_exp((Alo,Ahi),N,(Clo,Chi))
     :- N mod 2 = 1, exp(Alo,N,Clo), exp(Ahi,N,Chi).
int_exp((Alo,Ahi),N,(Clo,Chi))
     :- Alo > 0, exp(Alo,N,Clo), exp(Ahi,N,Chi).
int_exp((Alo,Ahi),N,(Clo,Chi))
     :- Ahi < 0, N mod 2 = 0, exp(Alo,N,Chi),
                              exp(Ahi,N,Chi).
int_exp((Alo,Ahi),N,(0,Chi))
     :- Alo <= 0, Ahi >= 0, N mod 2 = 0
```

```
        Aloabs = -Alo, max(Aloabs,Ahi,Chi).
/* Purely a precaution */
int_exp(_,_,_) :- write('int_exp: unknown error'),
                  !, fail.
```

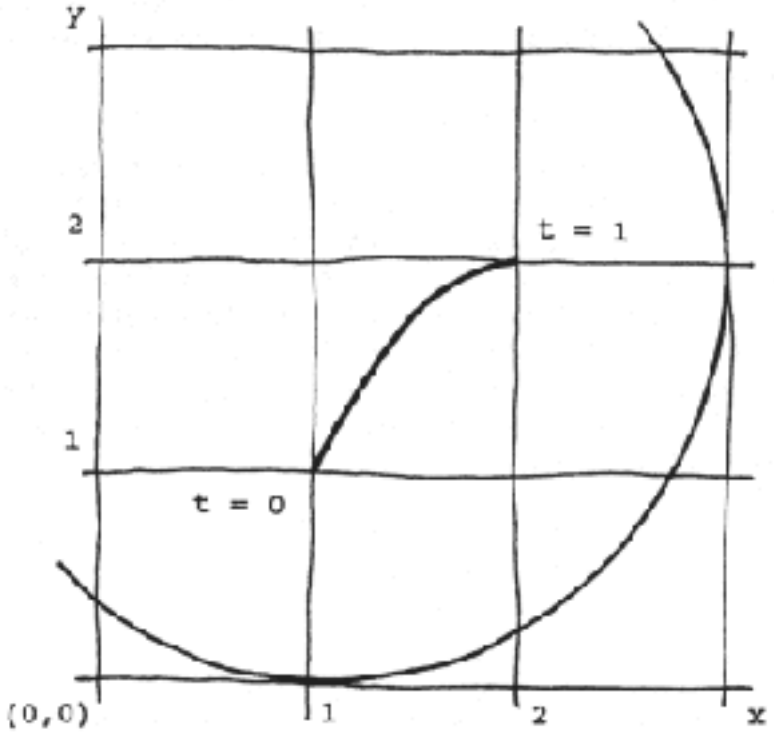
The utilities used follow, in case you haven't got them; everything else is according to the gospel of Clocksin and Mellish (see the References), except that we use a rather tidier `:=` for arithmetic operations which, of course, we assume to support floating-point numbers.

```
/* Min and max: min(A,B,min(A,B)), max(A,B,max(A,B)) */

min(A,B,A) :- A<=B.
min(A,B,B).
max(A,B,A) :- B<=A.
max(A,B,B).

/* Exponentiation (by recursion): exp(A,N,A**N) */

exp(A,N,B) :-  Ndec := N - 1, exp(A,Ndec,Bdec),
                  B := A * Bdec.
exp(A,1,A).
```

We could survive without explicit exponentiation, by using multiplication, but we can immediately see that this broadens intervals unnecessarily: $[-1,1]^2$, for instance, would be calculated as $[-1,1]$ rather than the tighter $[0,1]$.That's all we need for dealing with polynomials[1], so let's rush ahead and look at a geometric problem. Supposing we want to find the intersection between the parametric quadratic segment

$$\begin{aligned} x &= 1+t \\ y &= 1+2t-t^2 \end{aligned}$$

[1]In fact, all we've left out is the inverse of an interval, from which division immediately follows, and roots, which would allow fractional powers. Inversion involves ∞ (think of an interval which contains zero), but is perfectly well-behaved—see Milne's thesis in the References. Fractional powers smear us all over the complex plane, of course. Oh, and there's raising things to the power of an interval....



7(i)—The parametric quadratic $x = 1 + t, y = 1 + 2t - t^2$, and the circle $(x-1)^2 + (y-2)^2 - 4 = 0$.

where $0 \leq t \leq 1$; and a circle, centre $(1, 2)$, radius 2:

$$(x-1)^2 + (y-2)^2 - 4 = 0.$$

They're both sketched in Illustration 7(i).

Well, we could substitute the quadratic into the circle, in the usual way, and try to solve the resulting quartic. (We could even use interval arithmetic to try to see if that had any roots between $t = 0$ and $t = 1$.) Instead, we'll employ a more geometric use of intervals that illustrates how they define boxes. Take the quadratic; it is defined over an interval $[0, 1]$ in t. So, let's substitute that interval into the equations for x and y, and thus get intervals in x

and y out:

$$\begin{array}{rcll} x_{[0,1]} & = & [1,1]+[0,1] & = & [1,2] \\ y_{[0,1]} & = & [1,1]+[0,2]-[0,1] & = & [0,3]. \end{array}$$

How do we compare the box with the circle? We substitute the intervals $[1,2]$ and $[0,3]$ for x and y into the polynomial part of the circle equation:

$$(x-1)^2+(y-2)^2-4.$$

We know that any points *on* the circle will give 0 if substituted into that equation; any points *inside* will give a negative result; any points *outside*, a positive result. So if the interval we get out is negative (i.e. $[a,b]$, $a<0$ and $b<0$), then the box that the interval represents is all within the circle; if it's positive then the box is completely outside the circle. If the interval contains 0 (i.e. $a \leq 0$ and $b \geq 0$) then we have learned nothing, and must either give up, or try some more precise approach. So here goes with the substitution into the polynomial above:

$$\begin{array}{rl} & ([1,2]-1])^2+([0,3]-2)^2-4 \\ = & [0,1]^2+[-2,1]^2-4 \\ = & [0,1]+[0,4]-4 \\ = & [-4,1]. \end{array}$$

So, the interval contains zero, which is hardly surprising if we look at the figure; we can see that the circle intersects the interval on x and y we obtained from the curve. In fact, that interval is rather gross. Let's try another version of the equation of the quadratic—the Horner form. Obviously this only affects the equation for y:

$$y = 1+t(2-t).$$

Again substituting $[0,1]$ for t, we get:

$$1+[0,1](2-[0,1]) = 1+[0,1][1,2] = [1,3].$$

You can see from the figure that the new interval misses the circle, and indeed if we substitute it into the circle polynomial we get the result $[-4,-2]$; correctly confirming that the parametric quadratic lies inside the implicit circle.

The fact that a rearrangement of the quadratic equation produced a better result is actually rather worrying; in fact, any equation in which the interval variable *appears more than once* will in general give an over-pessimistic result: i.e. a larger interval than necessary to contain the geometry. Note that the circle equation and the parametric equation for x, *as written above* contain only one reference to x and y, and t, respectively. In such cases, the result is therefore as tight as possible.

However, even the rearranged quadratic for y did not produce a very satisfactory result; the peak value of y is actually 3, which occurs at $t = 1$, the end of the span. So the tightest interval obtainable is $[1, 2]$.

So let's try a couple more tricks on that quadratic for y. The first thing to note is that the growth in the interval caused by the $t - t^2$ term is sure to get worse the further we get from zero; we are subtracting two larger and larger numbers to get a small one, and the pessimistic nature of interval arithmetic makes this bad news. So we would do better to work with an interval *centred* around zero. In this case, we need to re-parameterize the curve with a new parameter $u = t - \frac{1}{2}$; so that u will run between $-\frac{1}{2}$ and $\frac{1}{2}$. Now, the new equation is:

$$y = 1\tfrac{3}{4} + u - u^2$$

and, substituting in the intervals, we get:

$$1\tfrac{3}{4} + [-\tfrac{1}{2}, \tfrac{1}{2}] - [-\tfrac{1}{2}, \tfrac{1}{2}]^2 = [1, 2\tfrac{1}{4}].$$

That's better than both the original and Horner versions over the interval $[0, 1]$.

The two ideas we've tried so far—rearranging the equation and centering the interval—can only be done once; supposing we want to try to make the interval tighter still? The last technique we shall see here—taking sub-intervals—can be applied as many times as necessary.

Let us yet again go back to the original form of the quadratic above, and divide the interval in t into two parts: $[0, \frac{1}{2}]$ and $[\frac{1}{2}, 1]$. Substituting the first half-interval into the quadratic, we get:

$$1 + 2[0, \tfrac{1}{2}] - [0, \tfrac{1}{2}]^2 = 1 + [0, 1] - [0, \tfrac{1}{4}] = [\tfrac{3}{4}, 2];$$

and substituting in the second:

$$1 + 2[\tfrac{1}{2}, 1] - [\tfrac{1}{2}, 1]^2 = 1 + [1, 2] - [\tfrac{1}{4}, 1] = [1, 2\tfrac{3}{4}].$$

That's better than the Horner form, but not as good as the centred interval, but that's not really important; we can combine all three techniques, *and repeat the last*. This splitting of intervals features in the textbooks on the subject, and there is an obvious relationship with quad-tree and oct-tree structures that we can exploit.

As an exercise, try applying all the interval reduction techniques described at once to our pet quadratic; the result should be $[1, 2\frac{1}{8}]$.

Other enclosures

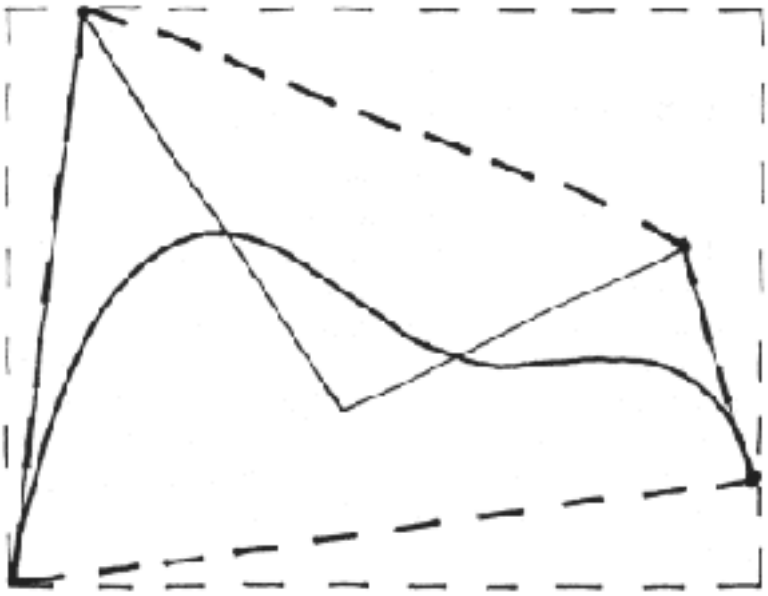
A lot was said about intervals, because they're not well-covered elsewhere. In this short sub-section we will run through some of the other enclosures available. In every case, the enclosure can be a permanent replacement for what it contains, but it is more often a dynamic structure that is refined in regions of interest.

Non-orthogonal boxes

A very significant improvement in the bounding of intervals can be achieved by having boxes which are not aligned with the axes. In two dimensions, a curve may be contained within a series of rectangular boxes (the strip tree). In three dimensions, enclosures may be associated with a primitive (e.g. a box around a cone) or generated from local properties of the surface (e.g. aligned with a coordinate system based on the surface normal).

Convex hulls

Bounding volumes need not be rectangular boxes; *convex hulls* are most commonly associated with Bernstein-basis curves and surfaces. However, in three dimensions, the precise convex hull of a Bézier control track is complicated to create and use; a box based on the extreme control-track points (optionally aligned with the surface normal) is often more practical (see Illustration 7(ii)). This is a typical example of a structure that is usually used dynamically,



7(ii)—A quartic Bézier curve, its control track, the convex hull of the control track (thick dotted), and an *interval* in x and y containing the hull (thin dotted). In this particular case, it is interesting to note that, while there are five points on the control track, only four contribute to the hull, and only three to the two-dimensional interval (the surrounding rectangle).

shrinking down upon regions of interest during a recursive division process.

Bounding volumes don't even have to have straight edges or flat surfaces. In two dimensions, circular regions can be used to enclose curves (see Sederberg and colleagues' paper on *fat arcs*). In three dimensions, spheres and ellipsoids are sometimes used, although they can be very cumbersome.

Approximate geometry

There are many well-known genuine approximations, rather than enclosures. They almost all involve replacement of an element by several elements of lower degree. The analogies with *splines* and with *degree reduction* of Bernstein-basis curves and surfaces are apparent. It is possible to have approximate geometry that is refined in regions of interest, or even replaced by the original exact geometry; but that approach doesn't work as well as it does with enclosures. The initial approximation can mean that a 'region of interest' (e.g. an intersection) is missed completely; in that case, the program never finds out that it should refine the representation, or restore a more accurate one.

Pixels

The picture on a raster-scan display is just a square array of dots, or *pixels*. A lot has been written about getting curves into pixels (i.e. rendering them), and recovering geometry from pixels. This is a specialized subject, usually only of use in low-level graphics and image processing.

Polylines

Polylines are a sequence of straight-line segments, common both as an approximation and as the data obtained from an input device such as a tablet or mouse. It's moderately easy to convert a parametric curve to a polyline (but watch the parameterization); less easy to convert an implicit curve. Conversion back is splining. A big problem with polylines is that the slopes of the straight-line segments essentially never match and so there is always some degree of magnification that makes them look tacky.

Facets

Facets are the three-dimensional equivalent of polylines. They make many three-dimensional operations much simpler, because all intersections are straight lines and they create no horizons away from their edges etc. etc. The well-known smooth-shading techniques of Gouraud and Phong have given facets an enormous following in the

graphics world. Beware of refining faceted representations too far; you are trading algebraic for combinatorial complexity and (in the case of doubly-curved surfaces such as spheres) the bargain may be very bad.

Biarcs

Biarcs and biquadratics are really a sort of splining technique, but may be used to replace higher-degree curves, rather than to approximate data. The reason for the prefix 'bi' is that arcs and quadratics are unable to meet Hermite end-conditions on their own, but can if two are used across a span. After approximation the lower-degree equations make many calculations faster, but there is real tangent continuity, unlike polylines and facets. Biarcs are commonly used for numerical control applications, to take advantage of functions available in machine-tool controllers. Biquadratics are related to the patches of the same name, which offer simpler-than-usual intersection calculations.

Arc length, surface area, and volume

Although integral formulae for calculating the length of curves, areas of surfaces and volumes of solids may sometimes be written down easily (well, fairly easily), solving them is another matter: approximate techniques are usually used.

The length of an arc of a curve can be found by summing an equivalent polyline, but this is always an underestimate. Simpson's rule—and similar but better quadrature formulae (see Guenter and Parent's paper)—give a much more accurate result for fewer steps. Surface areas can be calculated in the same way, as the approximation of a double integral, and volumes (of simple solids) by a triple integral.

Estimating the length of curve segments is not difficult, because the ends of the curve provide natural integration limits. Computing the areas of pieces of surfaces, and the volume of solids bounded by many faces, is much more difficult. We often end up creating a two or three-dimensional grid and—in effect—counting squares. This can be erratic, because of accidental alignments between the grid

and the object, and is therefore combined with a random sampling technique within the grid boxes.

Sampling points for volume is quite easy, but sampling area is more difficult. It is necessary to generate a pattern of random lines, and then count the intersections between the lines and the surface to be measured. Here are two questions for further thought:

> What is a computationally tractable way to generate a genuinely *unbiased* set of random straight lines?
>
> How is the number of 'hits' converted to a measure of surface area?

8

Storing geometry

There is much, much more in the literature about the different types of geometric element than there is about where to put them when you've got them. However, storing geometry is not a trivial problem. The first consideration is the operations in which the element will be involved, which will determine a choice, where possible, between the implicit and parametric forms. With simple elements, we may even indulge in the luxury of storing both forms; in that case they must of course be kept consistent.

Then there are different forms of the same polynomial; such as the power, Horner and Bernstein forms. The algebraic or algorithmic reasons for a choice are often overriding; here we will just consider some characteristics of a 'geometry bin'. *Compactness* is probably the most obvious consideration. To take a simple example, we have already seen that the equation of a plane is

$$ax + by + cz + d = 0$$

in the implicit form, while the parametric form is:

$$\begin{aligned} x &= x_0 + f_1 s + f_2 t \\ y &= y_0 + g_1 s + g_2 t \\ z &= z_0 + h_1 s + h_2 t. \end{aligned}$$

So, in this case, we can save five coefficients by using the implicit equation. It follows that there must be some redundancy in the parametric version. Of course, there is a redundant degree of freedom in the implicit equation too; we can normalize it to eliminate that freedom, by dividing through by $\sqrt{a^2 + b^2 + c^2}$.

We could eliminate redundancy altogether; suppose we just stored a, b and d, and recomputed c from the formula $c = \sqrt{1 - a^2 - b^2}$, when we needed it. This makes a 25% saving, but is not usually done because:

A square root operation is required to reconstitute c.

The sign of c has to be stored anyway.

The reconstitution will be ill-conditioned when c is small.

(Although we could be really obsessive and get around this last difficulty by storing the two smaller of a, b, and c, but you'd need a flag somewhere to indicate which one was omitted. Good luck.)

So, in storing planes, we put up with the redundancy, but the normalization is very important, because:

It makes the magnitudes of the numbers more predictable, and so reduces the likelihood of numerical problems.

It makes it easier to determine when two planes are the same, or nearly the same, and can share storage.

It avoids a normalization step in algorithms which get the plane equation from this source.

The parametric form of a plane equation can be normalized in a similar way, this time by dividing f_1, g_1, and h_1 through by $\sqrt{f_1^2 + g_1^2 + h_1^2}$, and similarly for f_2, g_2 and h_2. The extra degrees of freedom in the parameterized equation arise because the point (x_0, y_0, z_0)) can lie anywhere on its surface (i.e. anywhere in its parameter space), and that is two-dimensional. We can anchor (x_0, y_0, z_0) by putting it into a defined position (the point nearest the origin is usual); there is no natural choice for the orientation of the t, u coordinate system, but consistent choices are possible (see *A Programmer's Geometry*).

So we see that, even for a simple surface like the plane, there are a number of possible approaches to normalization. More complicated equations are more difficult again to normalize satisfactorily. For general implicit equations, we can adopt some more or less arbitrary but consistent scaling of the coefficients. Dividing through by the sum of the squares of all the coefficients, except the constant term, is the usual device, and is sometimes called supernormalization; it

gives consistency with the system just adopted for the plane, in case the polynomial degenerates to that form.

With general parametrics, we are usually interested in a particular curve segment or surface patch, and it is most convenient to have a parameterization that runs from 0 to 1 along the curve, or $(0, 0)$ to $(1, 1)$ across the patch. So the coefficients in the parametric equations will vary depending on where the piece of curve or surface we are interested in starts and finishes. Add to that the possibility of reparameterization, with degree elevation (see Chapter 4), and it is not possible to identify two segments as being part of the same parametric curve or bi-parametric surface without a considerable amount of arithmetic. This is not usually a problem; coincidental free-form curves and sculptured surfaces are much less likely to occur than, for instance, coincidental quadrics in a solid model of a machined component.

Equations

As we start to move down from the rarefied heights of algebra towards mundane questions of programming, we need to consider what coefficients we're actually going to store to represent a given piece of geometry, and (later) where we're going to put them. If we are, say, going to 'support' quadric surfaces, what should we do about planes? And cones, cylinders and spheres for that matter? The possibilities would appear to be:

Store everything as a general quadric, and treat it as such. So a plane equation is:

$$0x^2 + 0y^2 + 0z^2 + 0xy + 0yz + 0zx + ax + by + cz + d = 0.$$

Not too efficient: and do we really want to try to find the equation of the intersection of two planes by tracing intersection curves of two degenerate quadrics?

Store everything as a full-length general quadric equation, but tag planes, cylinders etc. so that appropriate pieces of code can be used: simple, but crude.

Economize on those shapes—planes, spheres, ellipsoids—that always have a number of coefficients set to zero, by allocating

them their own bits of storage: more economical, but we might run out of storage for planes while there were unused spaces for spheres. This is not a problem for languages that allow dynamic storage allocation, although storage allocation may still be a practical problem.

Do not store the quadric representing the actual size and position of natural quadrics at all. Store a minimalist representation of each element and the transforms needed to move it into final position. This could be as little as a code for the quadric's name (e.g. a 'sphere' is taken to be a unit radius sphere located at the origin). This just transfers the storage burden from the quadrics to the transform matrices. Further, some calculation is required before shapes are even ready to be used in operations at their correct sizes, positions and orientations.

Object-oriented languages, such as C++, provide an environment that strongly encourages the partitioning of geometric data such as natural quadrics into as many useful different types as possible. Types, or *classes*, of data can be arranged in a hierarchical fashion, with a lower-level class inheriting characteristics from a more general class above it. Thus the class of "cylinders" might have a sub-class "axially-aligned cylinders". Data, or attributes, applicable to all cylinders—such as a value for radius—would be defined for cylinders and *inherited* by axially-aligned cylinders.

Additionally, code—or 'methods' in object-orientspeak—that are appropriate to each class may usually be associated with it in convenient ways. For instance, some object-oriented languages provide constructs for checking that classes are properly maintained by routines; these entry and exit conditions can be explicitly programmed, and appropriate action to avoid corrupting data is taken if they are not met. Thus, a procedure to offset a sphere might subsequently check that the sign of the coefficient corresponding to r^2 in the sphere equation $(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 - r^2$ never becomes positive (or zero): which would indicate that the sphere was degenerate. Of course you could do this in any language, but object-oriented languages provide built-in ways to include these conditions, and to deal with the fall-out when they are *not* met.

An example: storing NURBS

So far we have focused on implicit equations. Many of the same considerations apply to parametric curves and surfaces, but there are also differences in storage strategy. For instance, we would not normally consider storing separate transforms for parametric equations. They already have so many degrees of freedom that separating out a transform is notional.

A more pressing question is how to deal with variable degree. Do we want to represent everything in the system as a degree-five Bézier surface, or whatever? You may have seen brochures for computer-aided design systems which say something like "all the surfaces in this system are universally represented as NURBS". Is this good or bad? What do they mean by it anyway? Well, it *may* mean simply that all the geometries supported are representable as NURBS; or it may mean what it says, that all the equations are stored internally as NURBS. What would this do to a plane?

Now, the rational B-spline patch equation is:

$$\mathbf{Q}(t,u) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{i,j}B_{i,k}(t)C_{j,l}(u)\mathbf{P}_{i,j}}{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{i,j}B_{i,k}(t)C_{j,l}(u)},$$

where the terms $B_{i,k}$ and $C_{j,l}$ are each defined in terms of their own knot vector $t_i(i = 0, m + k)$ and $t_j(j = 0, n + l)$. (The recursive de Boor definition of the knot vector appears in Chapter 4.) To define a plane in this elephantine way, we require the following pieces of data:

The degree k, which is 2.

Four control-mesh points $\mathbf{P}_{i,j}$. In this particular case, with the values we are supplying, the B-spline equation actually combines the control-track points as follows:

$$\mathbf{Q}(t,u) = tu\mathbf{P}_{0,0} + t(1-u)\mathbf{P}_{0,1} + (1-t)u\mathbf{P}_{1,0} + (1-t)(1-u)\mathbf{P}_{1,1}.$$

Thus the control-mesh points $\mathbf{P}_{i,j}$ must themselves be coplanar, or we will generate an arbitrary doubly-ruled surface instead of a plane. Given three (non-collinear) points in the plane, we can generate a fourth as a combination of them.

Four weights $w_{0,0}$, $w_{0,1}$, $w_{1,0}$ and $w_{1,1}$. Assuming the points $\mathbf{P}_{i,j}$ *are* coplanar, then the values of these weight will not make any difference to the plane defined, but it will affect the shape of the t, u parametric coordinate system that lies in it. The weights should be positive, however: not for any geometrical reason, but to avoid division by zero. Obviously sensible values (e.g. 1.0) will reduce the possibility of other numerical problems.

Two knot vectors $t_i(i = 0, m + k)$ and $t_j(j = 0, n + l)$: to define the plane over the parametric interval $(0, 0)$ to $(1, 1)$. These can conveniently both be set to 0011. Since we are talking about *non-uniform* rational B-splines, the knots *need* not be integers, and thus require to be stored as floating-point numbers.

In all then, we require 25 numbers to define the plane, and that assumes that there is no overhead because (see below) storage is automatically reserved for some higher degree of surface; in that case the requirement climbs giddily as nm, because of the number of storage locations we will need to reserve for vertices.

As well as the storage requirement, there is further bad news:

We have actually created a parametrically rectangular piece of plane; the points corresponding to values of t and u outside the range specified by the knot vectors are undefined.

Even a simple computation—for instance generating points on the plane—is likely to be very slow. Looking at that operation as an example, we see that evaluating each $B_{i,2}$ and $C_{j,2}$ costs between 9 and 11 floating-point operations (one addition, four subtractions, two divisions, and two to four comparisons): say 10 on average. Assuming that the computations are not wastefully repeated for numerator and denominator, there are four sets of subscripts for each. Thus the cost of evaluating $w_{i,j}B_{i,k}(t)C_{j,l}(u)$ four times is: 80 floating-point operations, plus 8 more to combine the results; *and* 11 additions to sum the denominator, 12 multiplications with $\mathbf{P}_{i,j}$, and 12 divisions by the denominator. In all (we make it) a princely 123 operations! (Compare just 12 operations for the boring old $x = x_0 + f_1t + f_2u$ etc. parametric equation.)

But that's just generating points on the plane; finding (say) intersections with all those other NURBS is going to be spectac-

ularly inefficient if we continue to regard our plane as one of them, and ignore the advantages of the implicit form.

So watch out for that "all NURBS" system; either it's not real or it's not efficient. Regrettably, geometry is not like currency conversion; changing all our dollars into pounds, or vice versa, often makes our life at our destination harder rather than easier.

Coefficients

So, having decided which equations to go for, where are we actually going to put the coefficients? Probably the worst answer is a matrix. The array form of a parametric equation shown here (or, worse, the tensor representation of an implicit equation) looks mathematically respectable:

$$\begin{bmatrix} x \\ y \\ z \\ \vdots \end{bmatrix} = \begin{bmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,n} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,n} \\ \vdots & \vdots & & \vdots \\ c_{n,0} & c_{n,1} & \cdots & c_{n,n} \end{bmatrix} \begin{bmatrix} 1 \\ t \\ \vdots \\ t^n \end{bmatrix}.$$

It is difficult to find anything good to say about this last nasty equation; anything that has a subscript for dimension (i.e. $x = 1$, $y = 2$, $z = 3$) is inefficient, unless we *really* want to vary the number of dimensions we're working in. And what happens if we require a lot of matrices? Do we add a third subscript? And a fourth for patches, to allow for the two parameters? This is likely to be both ugly and slow. Indeed many languages will not support arrays this wide; three cheers for the language designers.

Having a separate array for each of x, y and z reduces these requirements by one index. Arrays of coefficients of increasing powers of t can be a good way to feed a single parametric equation into a subroutine, because we can alter the degree of the equation by incrementing the index. But, to identify the data elements, we need an extra index. Further, the storage wasted becomes significant over many equations, all using the same amount of space as the highest-degree equations in the system.

A somewhat better scheme is obviously to create *records* in those languages that permit it. This has the advantage of keeping coordi-

nate values together, and thus usually making access faster. It also allows the whole geometric element to be passed around by a single data pointer. We can keep these records themselves in an array, or try to solve the problem of wasted storage through any of the usual list or heap datastructures that are widely used throughout programming.

Usually, the relationship between the coefficients of the equation of a geometric entity, and position, orientation etc., is not easily deduced. So there is not much point in trying to structure data so that it can be retrieved using its algebraic coefficients as keys. Firstly, these coefficients often refer to the whole geometric entity, not just the bit in which we're actually interested; secondly, they may be difficult to relate to geometrically useful data. These problems are addressed in the next two sections of this chapter. Even where the information stored about an entity does refer to simple geometry, such as the control track of a Bézier curve, and we are using all of that curve, the values—point coordinates in this case—cannot usually be accessed by direct match, and the specifically geometric structures we are going to look at must be used. They are usually auxiliary to the primary copy of the geometric data, and thus do not affect how this is stored.

Bounded geometry

As we have already observed, bounded geometry is required in most applications, rather than infinite planes etc. (although these are very cheap per acre). To bound geometry, we need to partition the space in which it lies (which may be real or parameter space) in one of two ways:

> We can introduce half-spaces (i.e. like implicit curves and surfaces) which allow points on the space to be classified directly. The actual boundary that is produced is—as usual—the barrier between the two classifications of space.
>
> This sounds simple, but we do need to find and store a piece of set-theory that relates the half-spaces in a way that describes the region of interest.

Alternatively, we can use geometry of lower dimensionality [1] which directly models the boundary itself (i.e. like parametric curves and surfaces), and rely on something like a ray-test (as we have already mentioned) to decide on which side of this boundary we are.

Where the boundary between inside and outside is made up of many small geometric elements, they *can* all be stored separately (as in the classic graphics *face model*). However, it is much better to store some connectivity information, ensuring that the pieces really do make up a contiguous loop, or a closed surface, and there are no missing or separated curve segments or faces.

This second alternative sounds less promising, but parametric elements have very tractable shapes, and finding the correct set-theory to combine elements which are half-spaces has often proved very difficult, so the boundary model approach is very commonly used. We will briefly discuss the storage implications of each.

Set-theoretic bounds

Set-theoretic bounding information includes simple things like intervals which are rather trivially incorporated into data structures. We *could* write the interval $[0, 1]$ as $(t \geq 0) \cap (t \leq 1)$ and store it as a piece of set-theoretic algebra—which it is. However, in general we know that a single one-dimensional interval is intended and allocate storage to suit. Indeed, more often than not, when curves are defined between 0 and 1 this information is never represented explicitly at all!

On the other hand, a large constructive solid geometry model may require many primitive solids related by a very complicated set-theoretic expression, such as a tree. This is stored in the same way as other types of algebra (i.e. a tree, or some traversal, such as reverse Polish notation). Because set-theoretic relationships are algebraic statements, their complexity does not increase with the dimensionality of the sets being combined.

[1] Only *one* fewer dimensions: bounding elements that fail to partition the space do not work; thus, wire-frames have been shown to be highly ambiguous representations of polyhedral solids. See Markowsky and Wesley's milestone paper of 1980: something of a tombstone for the wire-frame (although it has recently been receiving the attention of grave-robbers).

Connectivity

As far as stitching up elements of lower dimensionality is concerned, a variety of complicated structures is commonly used to represent the topological relationship—or connectivity—between them. In the case of a polygon, this may only be a list of the edges. In three dimensions, the faces of a boundary model have to be linked by pointers, because they can't be arranged as a list (the usual problem of no natural ordering). Also, each face must itself be bounded. This could be done by half-spaces in the face's parameter space, but it is much more usual to bound each face by another set of yet-lower-dimensional entities i.e. curves. Each face can be a free-standing polygon, but this leads to duplication and possible error. So, in the better class of model, faces, edges and vertices (vertices = bounds on edges!) are bound together into a single spaghetti Bolognese—sorry, data structure—such as the well-known winged-edge data structure.

These 'topological' structures guarantee that a polygon will be a loop, and a polyhedron will be isomorphic to a sphere, or whatever 'sphere-with-handles' shape that the genus [2] dictates; faces and edges cannot be missing. However, this does not stop a polygon being a figure-of-eight, or a polyhedron being self-intersecting. Some other agency must assure that. For instance, a boundary model that is created from a set-theoretic model will bring a guarantee of solidity with it. But, in many systems, the user (who is, after all, responsible for the *utility* of the shape) is also given responsibility to ensure that an area or a volume has actually been created.

Connectivity representations require a hierarchy of bounds which have different dimensionalities (e.g. zero-dimensional vertices, one-dimensional edges and three-dimensional faces bounding a three-dimensional object), and so the complexity of the pointering goes up dramatically as dimensionality increases (cf. a polygon and a boundary model). Double-ended pointers increase speed at the cost of storage.

Exploiting locality

[2] Genus is the topological measure of the number of holes through an object.

Neither of the two sorts of bounding structure outlined above responds at all well to being asked the question "Is this point inside the structure?". In both cases, a naïve algorithm has to access the entire structure (although the algorithms for doing inside-outside tests on topologically linked and set-theoretic structures are of course quite different).

The same applies to other essential geometric queries addressed to stored geometric data. The solution, as has already been seen, is to superimpose a simpler auxiliary structure, which increases *locality* by providing easily accessible regions into which the the more complicated shape data may be sorted. Localizing structures are therefore commonly stored with shape models of all sorts, and provide efficient access to them.

The suitability of a localizing structure for this purpose may easily be assessed against a small number of criteria:

> How well does the localizing structure fit the exact geometry? Are we going to have to break up quite simple exact elements into penny packets; are we going to find that some complicated regions are not localized at all?
>
> How easy is it to access the localizing structure? Obviously, it should be a lot easier than accessing the exact geometry, and the performance against complexity should be attractive.
>
> How easy is the localizing structure to implement and to maintain?

The enclosures mentioned in the last chapter are the 'classical' solution to this problem; in graphics, for instance, 'spheres round everything' was once the motto. The problem with enclosures is that it is not, of course, possible to get enclosures that will fit different sorts of data very well. When enclosures *don't* fit, trouble starts. Suppose we are trying to cover faces of polyhedra with spheres. If we have a long thin face that is a poor fit to a sphere, the obvious thing is to try and fit a lot of smaller spheres. But small spheres *don't* fit exactly inside a big one, so there is always complication in ensuring that a new set of spheres is chosen which doesn't leave some of the model out; and with so much overlapping a few levels of smaller and smaller spheres may be a very loose bit of localization indeed!

Adaptive tessellations are, of course, a better and more 'modern' answer to this problem; if a tile is a poor fit to part of the underlying data, it can be subdivided into pieces which it is known will fill the space *exactly*; that's what tessellations are about. So subdivision can continue cleanly until adequate localization is achieved.

Although covering the data efficiently is not a problem with tessellations, their shapes and access methods differ substantially. Here is a short review of some of the tessellations more commonly used for localization:

Grids

Grids are not of course adaptive; the only tailoring you can do is to choose a suitable pitch. But they are lightning-fast to access; you can get straight to the box that is holding a piece of geometry using the same simple access mechanism as a multi-dimensional array. Grids are easily implemented and widely used in 'simple' applications where different scales of data are not expected.

Quad- and oct-trees

Tree structures allow much better adaptation to regions of detail, but they are geometrically inflexible, as they are axially-aligned and with tiles of a single shape. The $O(\log n)$ time required to descend the tree can be a problem for large sets of data.

Excell

Tamminen's Excell structure (his 1980 report is worth the trouble of getting) is a grid of trees: a pragmatic hybrid that avoids time spent traversing the upper links of a quad- or oct-tree, on the assumption that, with real data of many sorts, no very large quads or octs will be present. Even if they are, the unnecessary work required to access them is usually of little concern; because they don't contain the interesting bits, it probably won't be necessary to access them very often.

Binary space partition

A tree-structured tessellation that was not mentioned in Chapter 6 is the binary space partition (BSP) (see Thibault's 1987 paper). It's

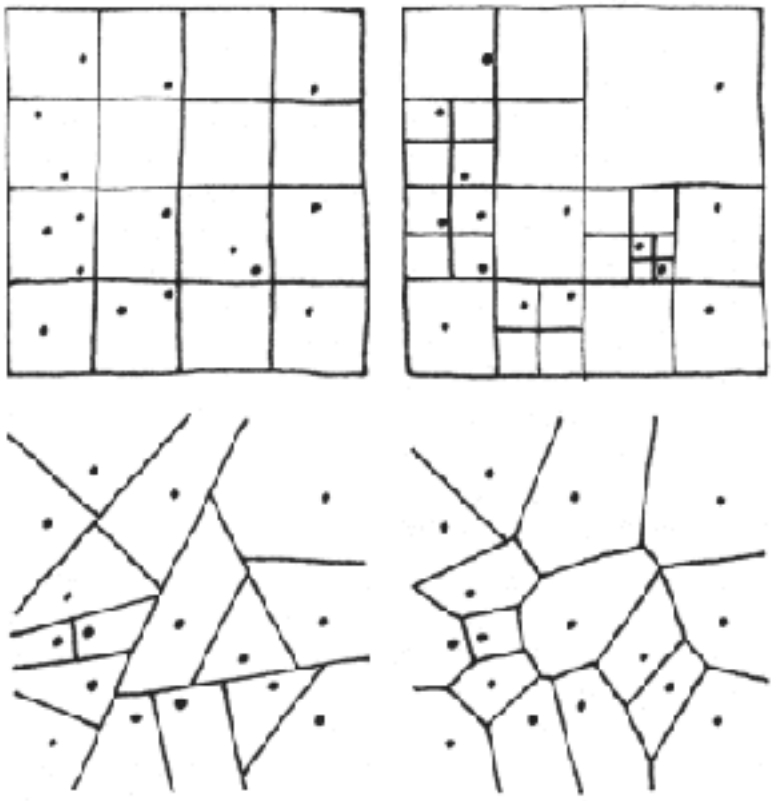
not theoretically highbrow but is widely used in graphics. It is constructed by segmenting the data using *arbitrary* cutting planes, chosen by some data-dependent heuristic. Because the planes are not axially aligned, they can be rotated to any position, or projected into perspective, at low cost and without any degradation of functionality (unlike grids, quad- and oct-trees). This attribute is highly prized for graphics applications, when scenes are to be rendered from a series of different viewpoints. The binary space partition is, however, virtually impossible to edit; it is a once-only localization process.

Dirichlet tessellations revisited

Localization based on the Dirichlet tessellation has attracted interest in recent years; Dirichlet tessellations have theoretical rigour, but are somewhat complicated and still the subject of research. As mentioned in Chapter 6, an exact Dirichlet tessellation of complex data (i.e. anything other than point sets) is algebraically very hard. Practical solutions can be obtained by incorporating heuristics to seed a complex shape model with points (e.g. take the vertices of a solid model) and tessellate them, rather than the underlying structure. If the density of these points corresponds reasonably well to the underlying data (the regions of detail, in the case of a model) then the resulting tessellation should localize the data reasonably efficiently.

Although a Dirichlet tessellation may fit the data well, access requires traversing the structure from one tile to another. However, unexpectedly, access to Dirichlet tessellations becomes more competitive as dimensionality increases.

An interesting alternative is to use the *logic* of a Dirichlet tessellation, but actually to compute another tessellation to approximate it, usually a quad-tree, or oct-tree. That keeps the geometry simple, makes access quicker, and is particularly useful for problems like those in robotics, where we need to know all the time how far we are from any piece of data—and thus from a collision with it.



8(i)—A point pattern localized by a grid, a quad-tree, a binary space partition and a Dirichlet tessellation. Observe that, using the grid, some of the boxes are empty, some hold more than one point; the quad-tree has some empty boxes, but none multiply-occupied. The binary space partition tree and Dirichlet tessellation have one point per region. The grid and quad-tree exhibit arbitrariness in choice of orientation and origin; the binary space partition is arbitrary in order of partition. The Dirichlet tessellation is unique, but it is also the most complicated of the structures.

9

Transforms

Transforms are really distortions of *space*; they allow us to generate geometric elements in convenient sizes, places and orientations, and then to move them to where we really want them. They are particularly significant in applications such as robotics and graphics, where the position of objects is often changing. Any mapping of the coordinate axes

$$\begin{aligned} x' &= f_1(x, y, z) \\ y' &= f_2(x, y, z) \\ z' &= f_3(x, y, z) \end{aligned}$$

is an acceptable transform. We are usually more interested in the rigid-body transforms which move geometry without changing its shape:

Translation (or shifting).

Rotation.

To these may be added the *shape-preserving* transforms, which allow change of size and handedness:

Scaling, which changes size only.

Mirroring, which produces right-hand and left-hand copies.

These are all *affine* (parallelism preserving) transforms and can be represented by the equations:

$$\begin{aligned} x' &= a_{11}x + a_{12}y + a_{13}z + d_1 \\ y' &= a_{21}x + a_{22}y + a_{23}z + d_2 \\ z' &= a_{31}x + a_{32}y + a_{33}z + d_3. \end{aligned}$$

The coefficients a are obviously the elements of a matrix, and so we arrive at the relationship between transforms and matrix algebra that is in *every* book on graphics. We reiterate it briefly here, using two-dimensional coordinates to save a bit of space.

A simple matrix multiplication:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

can represent movements in two dimensions such as a rotation θ about the origin

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

or a mirroring (in this case about the x-axis)

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Note that both the matrices which perform these rigid-body transforms have a determinant of 1. Scaling matrices do not have unit determinants; scaling by a factor s is achieved by the matrix

$$\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}.$$

But a matrix multiplication is unable to affect the origin $(0, 0)$, which is always mapped to itself. To obtain a shift, we need some way of adding values. This can be done as a separate operation, or we can increase the size of the matrices that we are using, and employ what are called *homogeneous coordinates*. Our new transforms look like:

$$\begin{bmatrix} x' \\ y' \\ s' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & d_1 \\ a_{21} & a_{22} & d_2 \\ p_1 & p_2 & s \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

After a homogeneous transform, the element at the bottom of the column matrix representing the result is restored to 1 by dividing the whole matrix through by s'. So, the mapping from x and y to x' and y' is:

$$\begin{aligned} x' &= \frac{a_{11}x + a_{12}y + d_1}{p_1x + p_2y + s} \\ y' &= \frac{a_{21}x + a_{22}y + d_2}{p_1x + p_2y + s}. \end{aligned}$$

The terms of the 2×2 sub-matrix, a_{11}–a_{22}, produce rotation, and shearing if we want it. It is convenient to keep the determinant of that sub-matrix at 1 (thus yielding a constant-area transform, in this two-dimensional case) as s does the scaling. The elements d_1 and d_2 give translation, while p_1 and p_2 produce perspective transform, which is used in graphics.

That completes a whistle-stop tour of the elementary ideas, which we pointed out can be (indeed are always) found elsewhere. In the remainder of this chapter we will look at some aspects of transforms, which are less commonly discussed, but inevitably emerge if you actually want to use them.

Implementing transforms

While matrices provide a great formalism for describing transforms, the fit is not as good as it is widely proclaimed to be:

> Matrices do not *per se* place an intuitively obvious constraint on the operations that they describe (as would be the case if—say—all matrix operations corresponded to rigid-body transforms); on the other hand they are not a fully general paradigm (suppose we want x' to have a term in x^2).
>
> Simple operations such as shifting and mirroring are unnecessarily complicated to explain, and to execute, using matrices.
>
> Even using homogeneous matrices, transforms do not map directly into matrix algebra; extra operations (subsequent normalizations) must be tacked on.

Against this:

> Matrices provide the best way to formalize the concatenation of transforms.
>
> Determinants provide an appropriate formalism to analyse certain (i.e. area- or volume-preserving) properties of matrices.
>
> Matrices are *de facto* compatible with graphics packages and matrix hardware (although the latter is still rather rare).

In practice, it is the concatenation property that is important. If we wish to shift some points once and once only, then some special code:

```
x = x + dx;
y = y + dy;
z = z + dz;
```

is both trivial and easy to understand. The alternative full homogeneous matrix operation involving 16 multiplications, 12 additions and three divisions is not to be thought of. On the other hand, the total amount of work required for a small number of simple transforms done sequentially soon becomes more than for the concatenated equivalent. The cost of the concatenation arithmetic can be quickly amortized over a few points, let alone a few thousand.

So, in general:

	Few points	*Lots of points*
Few transforms	*Bespoke code*	*Bespoke code*
Many transforms	*Bespoke code*	*Homogeneous matrices*

Sometimes, we may expect some of the transforms reaching a particular piece of code to be complicated, but others (most?) to be simple. What then? If we know that a transform matrix only represents—say—a translation, then we can pick out the data we need from a general matrix. There is some code that does concatenation in this way (see Cychosz' contribution to Glassner's *Graphics Gems:* pages 476–481) although it is rather spoiled by using double-subscript arrays. While these provide the best way to write matrix elements down, accessing two-dimensional elements requires a gratuitous multiplication that we can avoid, because we only need one size of matrix. For this reason, general matrix (worse, tensor manipulation packages) are best avoided.

So, as a trivial example, suppose we wish to transform a point `(XOLD, YOLD)` to a new point `(XNEW, YNEW)`, using two-dimensional homogeneous matrices that we know will often be shifts, rotations, or scalings. Presuming for now that the matrices came from somewhere else in our own program, it is easy to tag them with their type: `1`—*general,* `2`—*shift,* `3`—*rotation,* `4`—*scaling.* Here's what the code might look like. Note that, because of the good old `EQUIVALENCE` statement, at run-time we avoid having to deal with array subscripts completely in this piece of FORTRAN:

```
   REAL A(3,3)
   EQUIVALENCE (A(1,1),A11), (A(1,2),A12), (A(1,3),A13),
  +            (A(2,1),A21), (A(2,2),A22), (A(2,3),A23),
  +            (A(3,1),A31), (A(3,2),A32), (A(3,3),A33)
   DATA ACCY /1.0E-6/

   GOTO (ITYPE), 10, 20, 30, 40
```

... Unrecognized type code

```
C --- General case.

10 DENOM = XOLD * A31 + YOLD * A32 + A33
   IF(ABS(DENOM) .LT. ACCY)THEN
```

... New coordinate out of range

```
   ELSE
     DENINV = 1.0 / DENOM
     XNEW = (XOLD * A11 + YOLD * A12 + A13) * DENINV
     YNEW = (XOLD * A21 + YOLD * A22 + A23) * DENINV
   ENDIF
   GOTO 50

C --- Matrix is a shift.

20 XNEW = XOLD + A13
   YOLD = YOLD + A23
   GOTO 50

C --- Matrix is a rotation.

30 IF(ABS(A33) .LT. ACCY) THEN
```

... New coordinate out of range.

```
   ELSE
     DENINV = 1.0 / DENOM
     XNEW = XOLD * DENINV
     YNEW = YOLD * DENINV
```

```
   GOTO 50

C --- Matrix is a scaling.

40 XNEW = XOLD * A11 + YOLD * A12
   YNEW = YOLD * A21 + YOLD * A22

50 CONTINUE
```

Crude but effective: the accuracy constant obviously needs to be set according to the application. Note that the matrix type codes may also be used to make concatenations more efficient, by going to special code when, for instance, a shift is to be added to an existing matrix. The type codes can be made to survive certain concatenations (e.g. a series of shifts) and thus increase efficiency all along the line. Obviously, concatenations should be done by updating a running transform matrix, rather than creating a new one, to avoid copying. This is tricky in languages (e.g. PROLOG) that stop you writing back into a data structure, but otherwise simple. For instance, to apply a rotation to a matrix that is itself marked as a rotation matrix, looks like this when written out:

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{12} & a_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

(Plus a vestigial normalization step, if the code is really dumb.) But it comes down to the code

```
   REAL A(3,3)
   EQUIVALENCE (A(1,1),A11),(A(1,2),A12),(A(1,3),A13),
  +            (A(2,1),A21),(A(2,2),A22),(A(2,3),A23),
  +            (A(3,1),A31),(A(3,2),A32),(A(3,3),A33)
   GOTO (ITYPE), ... ,10, ...

   ...

10 ST = SIN(THETA)
   CT = SQRT(1.0 - CT)
```

```
TEMP = ST * A11 - CT * A21
A11  = CT * A11 + ST * A21
A21  = TEMP

TEMP = ST * A12 - CT * A22
A12  = CT * A12 + ST * A22
A22  = TEMP
```

In fact this piece of code works for any matrix for which:

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = 1.$$

Interpreting matrices

So far we have assumed (e.g. in the discussion of these notional type codes) that we know where matrices have come from. Sometimes, we don't. Maybe they come from another piece of software, or from some external input. We can see from the partitioning of the homogeneous two-dimensional transform matrix that we have already looked at—

$$\begin{bmatrix} a_{11} & a_{12} & d_1 \\ a_{21} & a_{22} & d_2 \\ p_1 & p_2 & s \end{bmatrix}$$

—that some bits are easy. If p_1 or p_2 are non-zero, then we've got a non-affine component in the transform; otherwise the shifts in d_1 and d_2, and the scaling factor in s, are easily extracted. But what about a_{11} to a_{22}? Supposing, as we well may, that we want to find out whether it's a rotation; we know that a rotation should be

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

so we can extract θ from one of the elements of the matrix (e.g. $\theta = \cos^{-1} a_{11}$) and then check whether the other three agree. In three dimensions, that calculation is a little less trivial, and worth looking at.

In space, a rotation can be about one of the primary axes, but it is more generally about an arbitrary axis. The matrix for arbitrary three-dimensional rotation is one of the more useful, but it is inconveniently omitted from most of the books (although it *is* in Rogers and Adams' book *Mathematical Elements for Computer Graphics*).

If the rotation axis is a unit vector (c_x, c_y, c_z), then the matrix is:

$$\begin{bmatrix} c_x^2 + (1 - c_x^2)\cos\theta & c_x c_y(1 - \cos\theta) + c_z \sin\theta & c_x c_z(1 - \cos\theta) - c_y \sin\theta \\ c_x c_y(1 - \cos\theta) + c_z \sin\theta & c_y^2 + (1 - c_y^2)\cos\theta & c_y c_z(1 - \cos\theta + c_x \sin\theta \\ c_x c_z(1 - \cos\theta) + c_y \sin\theta & c_y c_z(1 - \cos\theta) - c_x \sin\theta & c_z^2 + (1 - c_z^2)\cos\theta \end{bmatrix}$$

which looks complicated; as an aside, we see that it codes up nicely:

```
SINT = SIN(THETA)
COST = SQRT(1.0 - SINT * SINT)
D = 1.0 - COST

CXSQ = CX * CX
CYSQ = CY * CY
CZSQ = CZ * CZ

TEMP = CX * D
CXCYD = CY * TEMP
CYCZD = CY * CZ * D
CZCXD = CZ * TEMP

CXS = CX * SINT
CYS = CY * SINT
CZS = CZ * SINT

A11 = CXSQ + (CYSQ + CZSQ) * COST
A12 = CXCYD + CZS
A13 = CXCZD + CYS
A21 = CXCYD + CZS
A22 = CYSQ + (CXSQ + CZSQ) * COST
A23 = CYCZD + CXS
A31 = CXCZD + CYS
A32 = CYCZD - CXS
A33 = CZSQ + (CXSQ + CYSQ) * COST
```

But supposing we do want to go the other way, and

Find out whether a given matrix is a rotation.

Discover the axis and angle?

Equating all the terms in the matrix produces a non-linear, and very over-constrained, mess; we can easily spot a method of extracting $c_x \sin\theta$, $c_y \sin\theta$ and $c_z \sin\theta$. Since we know $c_x^2 + c_y^2 + c_z^2 = 1$, we can find $\sin\theta$: and hence c_x, c_y and c_z. We can then check all the other terms to whatever accuracy seems necessary.

Transformation of equations

Most texts restrict themselves to applying transforms to *points*. This is adequate, provided that the geometry to be transformed is constructed from a number of points, and can be reconstructed after transformation. This applies to wire-frames, polygons, Bernstein-basis curves and surfaces, and also curves and surfaces that we are prepared to reconstruct from point data by Lagrange interpolation.

Equations in the power basis are a different story. Parametric equations are relatively easy. If the polynomial is:

$$\begin{aligned} x &= x_0 + c_{11}t + c_{12}t^2 + c_{13}t^3 \ldots \\ y &= y_0 + c_{21}t + c_{22}t^2 + c_{23}t^3 \ldots \\ z &= z_0 + c_{31}t + c_{32}t^2 + c_{33}t^3 \ldots, \end{aligned}$$

and the transform is the affine:

$$\begin{aligned} x' &= a_{11}x + a_{12}y + a_{13}z + d_1 \\ y' &= a_{21}x + a_{22}y + a_{23}z + d_2 \\ z' &= a_{31}x + a_{32}y + a_{33}z + d_3, \end{aligned}$$

then the polynomial can readily be substituted into the transform, to give:

$$\begin{aligned} x' &= a_{11}x_0 + a_{12}y_0 + a_{22}z_0 + d_1 \\ &+ (a_{11}c_{11} + a_{12}c_{21} + a_{13}c_{31} + \ldots)t \\ &+ (a_{11}c_{21} + a_{11}c_{22} + a_{13}c_{32} + \ldots)t^2 \\ &+ (a_{11}c_{31} + a_{12}c_{32} + a_{13}c_{33} + \ldots)t^3 \end{aligned}$$

$\ldots.$

It's easy to see from this that the set of non-rational polynomial equations is not closed under perspective transforms (i.e. all polynomials don't transform to other polynomials); if the transforms are rational (either of the terms p_1 or p_2 are non-zero) then we must get a rational polynomial out.

Implicit equations are much harder. We have to invert the transforms, so that we can substitute them into the implicit equation, rather than vice versa. This gets complicated, so let's look at a simple two-dimensional example, the quadratic:

$$c_1x^2 + c_2y^2 + c_3xy + c_4x + c_5y + c_6 = 0$$

and the affine transform:

$$\begin{aligned} x' &= a_{11}x + a_{12}y + d_1 \\ y' &= a_{21}x + a_{22}y + d_2. \end{aligned}$$

Inverting the transforms gives:

$$\begin{aligned} x &= \frac{a_{22}x' - a_{21}y' + a_{12}d_2 - a_{22}d_1}{a_{11}a_{22} - a_{12}a_{21}} \\ y &= \frac{a_{11}y' - a_{21}x' + a_{21}d_1 - a_{11}d_2}{a_{11}a_{22} - a_{12}a_{21}}. \end{aligned}$$

And substituting into the quadratic gives a new set of coefficients:

$$\begin{aligned} c_1' &= c_1a_{22}^2 \\ &\quad +c_2a_{21}^2 \\ &\quad -c_3a_{22}a_{21} \end{aligned}$$

$$\begin{aligned} c_2' &= c_1a_{12}^2 \\ &\quad +c_2a_{11}^2 \\ &\quad -c_3a_{12}a_{11} \end{aligned}$$

$$\begin{aligned} c_3' &= -2c_1a_{22}a_{21} \\ &\quad -2c_2a_{11}a_{21} \\ &\quad +c_3(a_{11}a_{22} + a_{12}a_{21} \end{aligned}$$

$$
\begin{aligned}
c'_4 &= 2c_1(a_{12}a_{22}d_2 - a_{22}^2 d_1) \\
&\quad 2c_2(a_{11}a_{21}d_2 - a_{21}^2 d_1) \\
&\quad +c_3(2a_{21}a_{22}d_1 - (a_{11}a_{22} + a_{12}a_{21})d_2) \\
&\quad +c_4 a_{22}(a_{11}a_{22} + a_{12}a_{21}) \\
&\quad +c_5 a21(a_{11}a_{22} + a_{12}a_{21})
\end{aligned}
$$

$$
\begin{aligned}
c'_5 &= 2c_1(a_{12}a_{22}d_1 - a_{12}^2 d_2) \\
&\quad +2c_2(a_{11}a_{21}d_1 - a_{11}^2 d_2) \\
&\quad +c_3(2a_{11}a_{12}d_2 - (a_{11}a_{22} + a_{12}a_{21})d_1) \\
&\quad +c_4 a_{12}(a_{11}a_{22} - a_{12}a_{21}) \\
&\quad +c_5 a_{11}(a_{11}a_{22} - a_{12}a_{21})
\end{aligned}
$$

$$
\begin{aligned}
c'_6 &= c_1(a_{12}d_2 - a_{22}d_1)^2 \\
&\quad +c_2(a_{21}d_1 - a_{11}d_2)^2 \\
&\quad +c_3(a_{12}d_2 - a_{22}d_1)(a_{21}d_1 - a_{12}d_2) \\
&\quad +c_4(a_{11}a_{22} - a_{12}a_{21})(a_{11}a_{22} - a_{12}a_{21}) \\
&\quad +c_5(a_{21}d_1 - a_{11}d_2)(a_{11}a_{22} - a_{12}a_{21}) \\
&\quad +c_6(a_{11}a_{22} - a_{12}a_{21}).
\end{aligned}
$$

This is a lot of algebra, just for a quadratic: it shows how quickly things can get complicated. But there is no way around working this sort of thing out, and coding it up, if you want to transform implicit polynomials in the power form efficiently.

Again, a perspective transform will generate a rational equation, except for the plane, which is not distorted.

The advantages of geometric elements which can readily be generated from sets of points is obvious. We don't need special transform code for every different sort of geometry, and as an added bonus, if we require a non-linear transform, an effective cheat—transforming the control points and reconstructing willy-nilly to make an approximation—is readily available.

10

Intersections

The constructions that we have seen so far have allowed the poor curves and surfaces we've been generating some latitude to move, bulge, flatten, or loop-the-loop in order to meet our requirements. Let's recollect:

> Point, straight-line and circle constructions: so simple there are closed-form solutions for them (although some take a day or two to work out by hand—try the 'circle tangent to three circles' formula, for instance! It's worth using an algebra system for such things these days).
>
> Interpolating parametric curves through points and with given tangents; choosing the parameterization in advance makes things easier, but sometimes also bumpier.
>
> Interpolating parametric patches across sets of curves: again parametric positions are known; points on the surface can be derived from four boundary points only, using the Coons patch.
>
> Implicit blends: cannot be constructed on a point-wise basis, and degrees of blend equations soon become very large.

However, in all these cases, the way we formulated the construction was in some way favourable to the elements being constructed. It may be less easy to interpolate a Bézier curve through points on the curve than to generate it from points on the control track, but the algebra is relatively straightforward. In this chapter and the one that follows, we are going to look at a couple of constructions where the new geometry is constrained *in its entirety;* not being allowed

the fun of bulges and so on, the equations rebel and get very nasty; we will have to talk about approximations....

Intersections are places where two pieces of geometry meet; at points on an intersection, the equations of both pieces of geometry are satisfied. In general, an intersection poses the dual problems:

> Finding it.
>
> Representing it.

The algebraic difficulties are well known, and hinge on the types of equation we have available.

> If we have two sets of parametric equations, then we can easily *start* towards a solution by eliminating the coordinates. For instance, in three dimensions, a parametric surface $\mathbf{P}_1 = \mathbf{Q}_1(t, u)$ and a parametric curve $\mathbf{P}_2 = \mathbf{Q}_2(v)$ yield three equations of the form $\mathbf{Q}_1(t, u) = \mathbf{Q}_2(v)$. However, simultaneous non-linear equations like these are not promising.
>
> If we have two sets of implicit equations, there is no trivial progress we can make.
>
> If we have a parametric and an implicit equation, then we can substitute for x, y and z in the implicit equation, and we have only one equation to solve, in either one or two parametric variables.

For non-trivial cases, the simultaneous equations that pop out of the parametric-parametric and implicit-implicit cases will yield only to numerical methods, or to advanced algebraic methods such as Sylvester resultants (see Davenport, Siret and Tournier's book, and the section at the end of this chapter). This yielding is often reluctant, and a direct attack on these sets of equations is not to be undertaken lightly.

The easy(ish) pickings left are therefore:

> Elements which have both parametric and implicit representations (*sensible* ones, not the result of implicitization techniques, which generate huge equations); these run out at the quadratics and (in three dimensions) the quadrics. The rational parameterizations of the latter aren't much fun either.

> Unbalanced situations where one of the geometric elements has both representations, and so we can choose whichever one makes the computation easier.

The asymmetry of the latter approach may not be intellectually satisfying, but there are some rather significant practical applications.

For instance, a parametric surface $\mathbf{P} = \mathbf{Q}(t, u)$ and a plane (obviously in implicit form) $ax + by + cz + d = 0$, yield a single equation in t and u, which is the implicit equation of the intersection curve *in parametric space*. This approach is popular in constructing *scan-line* rendering algorithms for parametric surfaces, where the projection of a raster line on the screen is the plane.

Another commonly exploited example is the intersection of a parametric straight line with an implicit surface. This is the basis of many ray-tracing methods. A parametric straight line $\mathbf{P} = \mathbf{P}_0 + \mathbf{P}_1.(t)$ can be substituted into an implicit surface $f(x, y, z) = 0$ and the result is a polynomial in t in the maximum degree of the surface. Even when polynomials in one variable are of high degree, they are susceptible to a good collection of pretty reliable numerical techniques for root isolation and finding. Furthermore, quadratic and cubic equations have formula solutions. Let's look at the intersection of a straight line and a general quadric using the quadratic formula. The straight line looks like:

$$\begin{aligned} x &= x_0 + ft \\ y &= y_0 + gt \\ z &= z_0 + ht. \end{aligned}$$

The quadric looks like:

$$a_1x^2 + a_2y^2 + a_3z^2 + a_4yz + a_5zx + a_6xy + a_7x + a_8y + a_9z + a_{10} = 0.$$

Note that we use a symmetrical form of the equation, rather than exact *lexicographic order* (which would have the fourth to sixth terms $a_4xy + a_5xz + a_6$); that is what we would get out of an algebra system, but symmetry makes the code much easier to check.

So, the resulting quadratic in t looks like:

$$(a_1f^2 + a_2g^2 + a_3h^2 + a_4gh + a_5hf + a_6fg)t^2$$

$$+(2a_1fx_0 + 2a_2gy_0 + 2a_3hz_0$$
$$+a_4(hy_0 + gz_0) + a_5(fz_0 + hx_0)$$
$$+a_6(gx_0 + fy_0) + a_7f + a_8g + a_9h)t$$

$$+a_1x_0^2 + a_2y_0^2 + a_3z_0^2$$
$$+a_4y_0z_0 + a_5z_0a_0 + a_6x_0y_0 + a_7x_0 + a_8y_0 + a_9z_0 + a_{10}.$$

Considering it as $at^2+bt+c = 0$, we solve it with the usual formula:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

And here's the code for the internals of a procedure to do it:

```
#include <math.h>

#define ACCY (1.0e-6) /* The usual near-zero epsilon */

/* Absolute value macro follows */

#define abs(x) ( ((x) < 0) ? (-(x)) : (x) )
/*
 * Input variables
 */
     float x0,y0,z0,f,g,h;
     float a1,a2,a3,a4,a5,a6,a7,a8,a9,a10;
/*
 * The answers
 */
     float *t1,*t2;
     int *root_count;
/*
 * Intermediate results
 */
     float a,b,c;
     float a1_f,a2_g,a3_h,denom,den_inv,sub_fac,
               a4_y0,a5_z0,a6_x0,ac2,root;
```

```
         .
         .
         .

     a1_f = a1*f;
     a2_g = a2*g;
     a3_h = a3*h;

     a = f*(a1_f + a6*g)
       + g*(a2_g + a4*h)
       + h*(a3_h + a5*f);
     denom = a + a;

     if (abs(denom) < ACCY)
     {
/*
 * Asymptotically meets the quadric (e.g. a
 * hyperboloid)  at infinity;  not much use.
 */
        *root_count = 0;
        return;
     }

     sub_fac = x0*a1_f + y0*a2_g + z0*a3_h;
     a4_y0 = a4*y0;
     a5_z0 = a5*z0;
     a6_x0 = a6*x0;

     b = sub_fac + sub_fac +
          f*(a5_z0 + a6*y0 + a7) +
          g*(a6_x0 + a4*z0 + a8) +
          h*(a4_y0 + a5*x0 + a9);

     c = x0*(a1*x0 + a5_z0 + a7) +
          y0*(a2*y0 + a6_x0 + a8) +
          z0*(a3*z0 + a4_y0 + a9) +
          a10;

     ac2 = denom*c;
```

```
     root = b*b - ac2 -ac2;
     if (root < ACCY)
     {
/*
 * It would seem that we have missed the quadric;
 * but we may be hitting it at a tangent, and the
 * result has been depressed through zero by numerical
 * problems, so it's fudge-factor time.
 */
          if (root < -ACCY)
          {
               /* All right, all right,
                  we really have missed. */

               *root_count = 0;
               return;
          } else
          {
               *root_count = 1;
               *t1 = -b/denom;
               return;
          }
     }

     *root_count = 2;
     root = (float)sqrt((double)root);
     den_inv = 1.0/denom;
     *t1 = den_inv*(root - b);
     *t2 = den_inv*(-root - b);
     return;
```

We have tried not to calculate anything before we need it, so as not to waste effort in the event that there are no (useful) intersections. By a factorization scheme, we have saved nearly half the multiplications (now 27, was 52) over the 'natural' factorization of the original quadratic.

The results appear as up to two values of the parameter of the straight-line equation, with `*root_count` giving the number. We expressly avoid the nonsense of putting them in a two-element ar-

ray. In fact this is the start of the next problem to think about: representing the results of intersection calculations when we've done them.

Two-dimensional intersections, and intersections of curves with surfaces in three dimensions, have the enormous advantage that the results can be represented as points. Further, they are points on a one-dimensional curve, and so can be *sorted*. Thus, even if the product of the degrees of the surface and the straight line is large, and there can be many roots of the intersection equation and many intersection points, we can usually deal effectively with these by finding the first, last, the one lying between two particular parameter values, or whatever. In fact, in a naïve (i.e. slow) ray-tracing process, we might calculate the intersections with *all* the surfaces in the scene, even when these are different pieces of geometry. All the intersection points, of whatever their origin—one from this plane, that quadric etc.—can be sorted *en masse* and the nearest to the end of the straight line is the first surface that the ray strikes. So, in this case, we have:

> A representation: a list of parameter values (usually an array if more than two).
>
> A way of selecting the result we want (e.g. taking the first, the first after a particular parameter value, or whatever).

When it comes to surface-surface intersections, the problem is much more difficult; it's still a research topic. We offer a highly informal summary here; there is no particularly good single source of further information, although the paper by Patrakalakis and Prakash of 1989 is reasonably accessible. Let us look in turn at the problems of finding and representing the intersection.

Finding intersection curves

The algebraic problems we have already mentioned are severe, but we offer some insight in the final section of this chapter. However, rather than use approximations to the algebra or numerical methods, or both, most published algorithms attack the problem geometrically. We can identify four approaches:

> Find (somehow) a point on the intersection, and step along it by optimization methods, remaining within (an approximation

to) a certain distance of each surface (the distance approximation is more easily available for implicit surfaces, as we can use them as potential functions).

Compute (iso-parametric) curves on one surface and find their intersection with the other one (obviously one surface must be parametric).

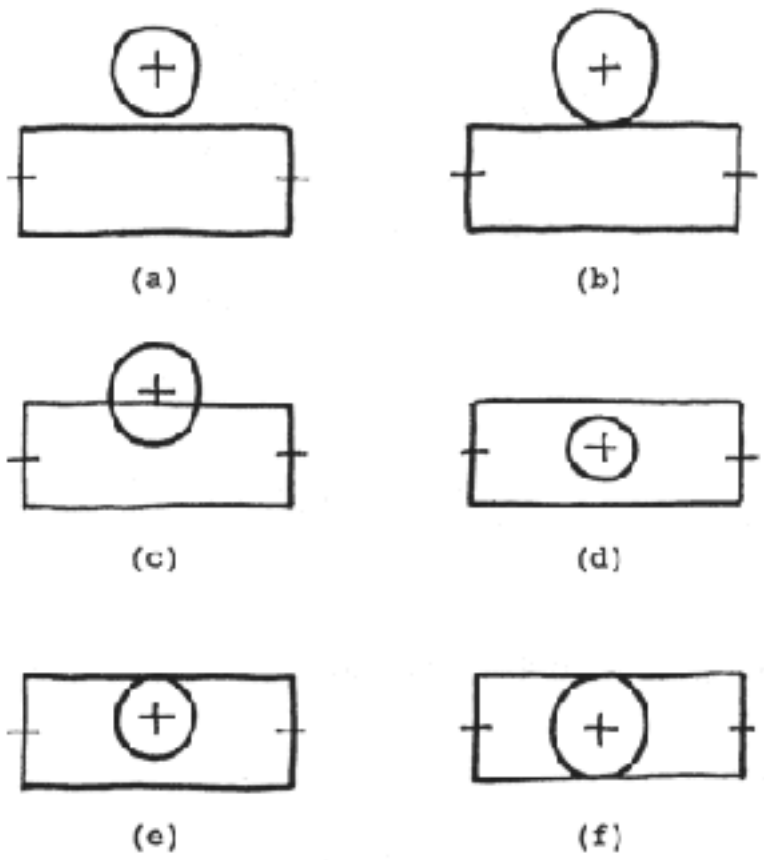
Recursively divide each surface (if they are both parametric) into smaller and smaller parametric intervals (sub-patches). Compare these using convex-hull properties, or ultimately as facets.

Recursively divide *space* into (e.g.) an oct-tree and use (e.g.) interval arithmetic to discard sub-spaces through which the surfaces don't pass. This works best with parametric surfaces, and in the leaf sub-spaces a planar approximation can be made to the surface, and a piece of polyline approximation to part of the intersection curve formed as a plane-plane intersection between these facets.

Intersection curves can be very complicated; for instance, even a simple cylinder-cylinder intersection can have six cases, as shown in Illustration 10(i). If the starting points are poorly selected, then the first method runs the risk of failing to find great chunks of curve. The second method also has this problem. The third and fourth methods will find everything (down to their resolving accuracy) but can be slow. A favourite (but not guaranteed) compromise is to start by a search and then trace the pieces of intersection curve found to get more detail.

Representing intersection curves

Representing the curves is another problem. As we saw above, when an implicit curve intersects with a parametric surface, the intersection comes out as an implicit two-dimensional curve in the parameter space of the parametric curve. Where the equation does not appear in such a nice way, we can still choose to represent the result as a numerical process in this form: as a polyline or spline in the parametric coordinate space of one surface (or both, but in this case they won't even be the same approximation to the real curve!). Representation in parametric coordinates is advantageous if we want to



10(i)—Six cases of a cylinder-cylinder intersection (the two cylinders are orthogonal, and drawn in parallel projection): `(a)` no intersection curve; `(b)` one-point contact; `(c)` one loop of intersection curve; `(d)` two loops; `(e)` two loops meeting at a point; `(f)` four segments meeting at two points.

represent pieces of surface (i.e. trimmed patches). If we are happy to have the intersection as a space curve, or if both the surfaces were implicit anyway, then we can represent the intersections as such. At least then we don't have to allow for distortion in a surface's parametric space which makes it difficult to decide what resolution (e.g. segment length in a polyline) is acceptable.

Resultants and discriminants

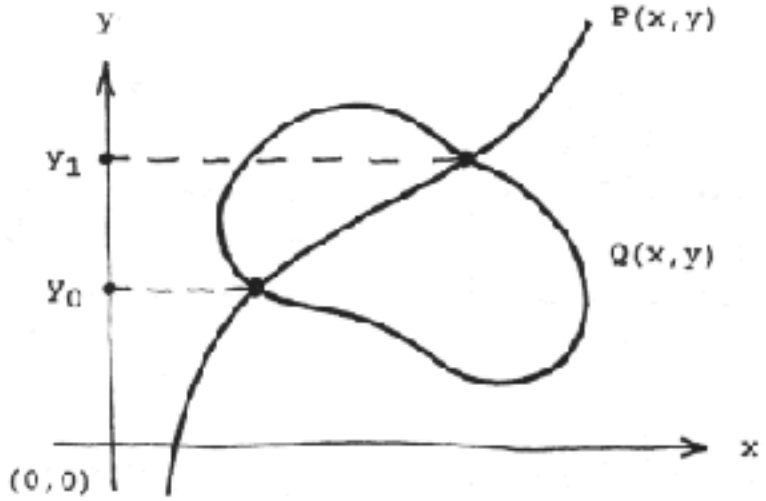
Resultants

Though we may not be able to find the intersection of, say, two multivariate implicit polynomials exactly, most algebra systems (see Davenport, Siret and Tournier's book) have routines to find the *resultants* of such polynomials. A resultant is a projection of an intersection. For example (see Illustration 10(ii)), suppose we have two polynomials $P(x, y) = 0$ and $Q(x, y) = 0$. We can eliminate x from them both (essentially by treating them as non-linear simultaneous equations). The answer is a polynomial which only contains terms in y, and the roots of this are the projections on to the y axis of where the original two polynomials intersected. Obviously we can do the same trick, but eliminating y instead, to get the projection on to the x axis.

This works in any number of dimensions; if we have $P(x, y, z) = 0$ and $Q(x, y, z) = 0$ we can eliminate z, say, from the pair and obtain a resultant, $R(x, y) = 0$ which will be the projection in the $x - y$ plane of the actual intersection curve(s) in space between P and Q. We are not restricted to projecting in a coordinate direction: the whole system can be rotated, and a projected resultant taken in any direction.

What follows is a transcript of an interactive session with a program called GAS (Geometric Algebra System) (see Milne's thesis) finding the x resultant of two polynomials in x, y and z:

```
[1] p := 2*x*y + 3*x*z - 5*z^2 - 2*x + 7;
(2*Y+3*Z-2)*X-5*Z^2+7
```



10(ii)—The resultant is a projection of an intersection; y_0 and y_1 are the resultant of **P** and **Q** obtained by eliminating x.

```
[2] q := -4*z*y + 7*x^2 - 2;
7*X^2-4*Z*Y-2

[3] resultant(p,q,'x);
-16*Z*Y^3+(-48*Z^2+32*Z-8)*Y^2
  +(-36*Z^3+48*Z^2-40*Z+16)*Y
  +175*Z^4-508*Z^2+24*Z+335
```

Note that, though the number of variables has been reduced, the degree of the resultant is higher than that of the parent polynomials.

The method of resultants is in essence the technique used for implicitization: the algebraic conversion of parametric polynomials to implicit polynomials. If we have a parametric patch

$$\begin{aligned} x &= x(u,v) \\ y &= y(u,v) \\ z &= z(u,v), \end{aligned}$$

we can treat it as three equations in five unknowns (x, y, z, u, and

v), eliminate v from the pairs (1,2) and (2,3), then eliminate u from the resulting two equations. The answer will be a polynomial in x, y and z which will correspond with the original parametric patch. Unfortunately, it will also extend beyond the parametric boundaries of the original patch, and so may self-intersect and do all sorts of other horrid things.

Discriminants

The *discriminants* of a polynomial are the projections of its horizon curves in some direction. They are found by taking the resultant of the polynomial with its own derivative. The discriminant of a cylinder would be two straight lines (or a circle, if the projection was along the axis); that of a sphere would be a circle in any direction, and so on.

11

Distances and offsets

Distance

The idea of distance is of course a powerful one in geometry, but trebly so in computing with geometry:

> A distance may be the answer to a problem; for instance, we need to know the distance between two conductors we are designing to see whether a spark will jump between them.
>
> A problem that does not seem particularly connected with distance is actually most easily formulated in that way. An example is the *offset* to a surface (covered below); we may visualize it as the path of a ball rolling in contact with the surface, but the problem is most satisfactorily formulated in terms of distance, not tangency.
>
> The most frequent use of distance in computing with geometry is in *culling* out geometric comparisons without actually doing them. For instance, if we know that two surfaces are too far apart to intersect, we need never attempt the (difficult) intersection calculations (see the previous chapter).

We will generally be interested in Euclidean distance, that is, the usual 'as the crow flies' sort. Some problems require different *metrics*. For instance, if we wish to calculate which points will contact first on the jaws of a press which are coming together, we are interested in distance along the direction of travel. That is a simple example; distance in a given direction is easier to calculate

than Euclidean distance. At the other end of the spectrum are things like non-uniform offsets to surfaces, where the distance metric varies with direction.

The last of the above uses of distance—as a test—is so common that it biases the way we think about distance; often we cannot obtain exact values from distance computations, but guaranteed *under*estimates are often nearly as good. That is to say, if we know that something is definitely not nearer than a certain distance, then we can often build a test or algorithm around that, even if the underestimation is severe. (We would of course like it to be as good as possible, but what does that cost?) Estimates of distance that may be too great or too small are rarely of interest.

Like everything else, distance calculations start by being easy(ish) and get tougher quickly. Let's start with points.

Point-to-point distance

The simplest distance formula is that from a straight line or plane to a point. From the point (x_0, y_0, z_0) to the plane $ax+by+cz+d=0$ the distance is $|ax_0 + by_0 + cz_0 + d|$, provided that the plane is normalized (i.e. $a^2+b^2+c^2 = 1$); that's all, and that's why problems involving distance in a given direction can be relatively easy. In effect it is the same as finding distance in a single coordinate; the plane effectively rotates the axes.

The Pythagorean formula $\sqrt{x_1^2 + x_2^2 + x_3^2 + \cdots}$ is, of course, used to calculate distance between two points; it works in any number of dimensions, as the terms x_i are meant to indicate. Maybe this is the all-time most well-known formula, in geometry at least. However, the necessity for a square root is a continuing annoyance (no doubt the Greeks thought it was annoying too). Surprisingly, there are actually a few things we can do about it:

> Don't do it; if we are comparing distances, then comparing squared distances is just as good (although you'll have to square any that come out linear formula, such as point-to-plane).
>
> Approximate it; the approximation $\max|x_1-x_1'|, |x_2-x_2'| \ldots$ is quite good for everyday numbers of dimensions. (But awful for spaces of science-fiction dimensions. Compare the maxi-

mum error in two, three and a hundred dimensions; curious, isn't it?) In two dimensions, the recipe $|x - x'| + |y - y'| - \min(|x - x'|, |y - y'|)/2$ is effective (see Paeth's contribution to Glassner's *Graphics Gems*: pages 427–431). There are a number of other approximation formulae: some are over-estimates and some are approximations which may be over or under; these are less useful.

Note that the square root function on a computer is performed by means of an expansion. We can expand the function $\sqrt{x^2 + y^2}$ directly, as a Taylor series or use Halley's method (see Dubrulle's paper, or that of Moler and Morrison, both published in the *IBM Journal of Research and Development* in 1983). There are additional advantages in this approach; the loss of accuracy when x is much larger than y (or vice versa) can be reduced, and likewise the possibility of the squares producing arithmetic overflow or underflow. But unless you are writing *very* low-level code (e.g. for a graphics device), it is doubtful whether this sort of thing can be made to pay in terms of speed alone.

Distances to general curves and surfaces

Only the straight line and circle, and plane and sphere in three dimensions, have obvious exact distance equations. The circle and sphere are merely contours of distance from a point: not particularly significant. The implicit equations of other quadratic curves and quadric surfaces—even the humble ellipse—do not have linear relationship with distance that we would like. Their equations do define a mock-distance, called variously a potential function or *algebraic distance*, which was mentioned in Chapter 5.

There is a little more to be said, however. Parametric curves are not too horrible. If we have a curve

$$\begin{aligned} x &= f(t) \\ y &= g(t) \end{aligned}$$

and a point (x_0, y_0), then we can express the (squared) distance between them as

$$(x_0 - f(t))^2 + (y_0 - g(t))^2.$$

At the point on the curve closest to (x_0, y_0), the distance (squared or not) will be a minimum. Thus

$$\frac{d}{dt}[(x_0 - f(t))^2 + (y_0 - g(t))^2] = 0.$$

For quadratic f and g, we would obtain a cubic equation to solve, which does have a formula. Of course, the roots may be maxima or minima; to distinguish these the second derivative has to be examined. For higher-order curves, numerical methods are required. Replacing the curve by a surface produces not only two parameters, but two equations to be solved, as the partial derivatives in both parameters must be equated to zero; that is not so easy to solve.

Offsets

A powerful idea in geometry is to let one thing slide or roll along another and see what shape it traces out. The cycloidal curves of gear teeth are one well-known embodiment of the idea; the curves need to be how they are so that they do roll, and don't slide. Another application is in machining; in most machining processes, the cutter is not a point tool. The trajectory that the centre of the cutter takes to produce a particular curve or surface is a new curve or surface somewhat away from, or offset from, the first, so that the distance between the centre and periphery is accounted for, and the correct shape is manufactured.

This may be an elegant idea, but it produces nasty geometry:

> The algebra is horrid; we are usually forced straight away into one approximation or another.
>
> We can get pieces of curve or surface that we don't want, and we may not get some pieces that we do want.

The simplest form of offset is a curve a constant distance from another. Think of it as the curve traced out by the centre of a circle rolling along a curve: in three dimensions, a sphere rolling over a surface. Very few curves and surface types are *closed* under offsetting; that means, the offset is not a curve or surface of the same type. In fact, only straight lines and circles in two dimensions, and the

natural quadrics and torus (actually cyclides [1]) in three dimensions, are closed under offsetting.

We can get further with general parametric curves and surfaces. For a curve, we can make some progress by realizing that the centre of the circle which traces out the offset is always along the *normal* from the point where the circle touches the curve. If the curve is:

$$\begin{aligned} x &= f(t) \\ y &= g(t), \end{aligned}$$

then we know the tangent vector is $(f'(t), g'(t))$; we can obtain normal vectors by a quick trip into three dimensions. Taking the vector product of $(f'(t), g'(t), 0)$ with each of the unit vectors pointing out of the plane $(0, 0, 1)$ and $(0, 0, -1)$, we get $(g'(t), -f'(t))$ and $(-g'(t), f'(t))$. These results must be normalized (note the two uses of the word 'normal'), to give the unit normals:

$$\pm \left(\frac{g'(t)}{\sqrt{f'(t)^2 + g'(t)^2}}, -\frac{f'(t)}{\sqrt{f'(t)^2 + g'(t)^2}} \right).$$

Try it yourself for the quadratic

$$\begin{aligned} x &= a_1 + b_1 t + c_1 t^2 \\ y &= a_2 + b_2 t + c_2 t^2. \end{aligned}$$

Oh dear, the result is not even a rational polynomial, because of that wretched square root. Actually, there are some special polynomials to which the offsets are at least rational polynomials, if not simple ones. They are exotically called Pythagorean hodographs (see Farouki and Sakkalis' paper) but are not (yet) in wide practical use.

So what to do? The easiest thing is to offset the curve on a point-by-point basis, and there's plenty of numerically controlled machine tool software that does just that. You need to watch the spacing

[1] A cyclide is a distorted torus of which the diameter of the minor circle varies in such a way that the cyclide would fit between two angled planes (i.e. planes forming a wedge): as opposed to the two parallel planes between which a torus would fit.

of the points, though; even spacing on the original curve doesn't produce even spacing on the offset when the curvature is sharp.

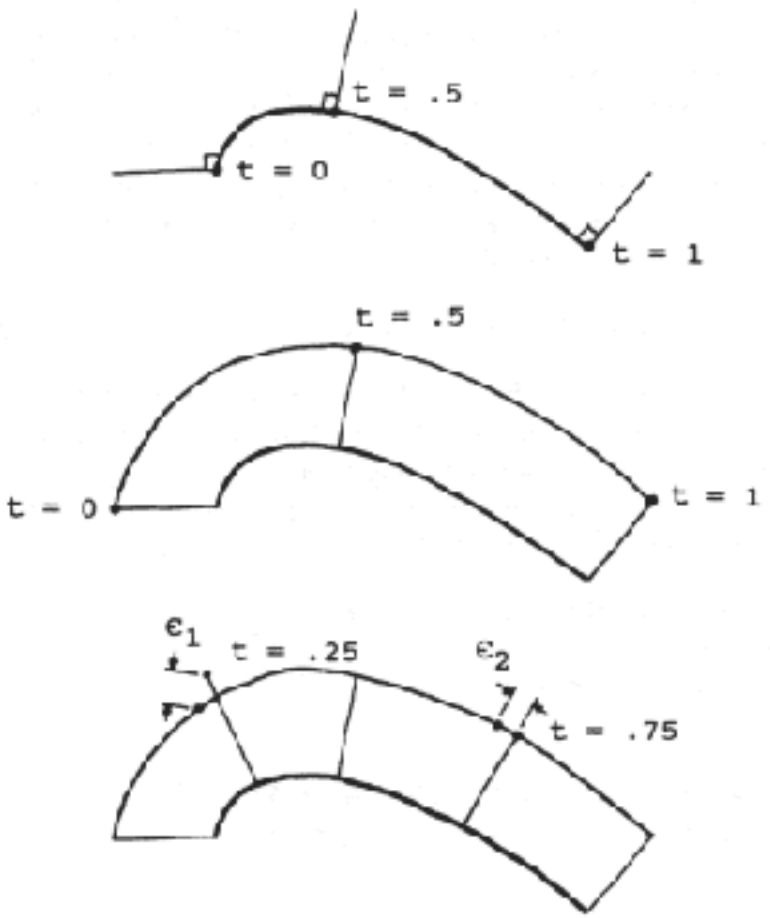
In effect, this is approximating the offset curve by a polyline, and it's likely that acceptable smoothness and accuracy can be achieved with rather less data using approximate curve segments which are at least slope-continuous. There are a number of approximate offsetting schemes in the literature, which mostly produce approximate offset curves of the same type as the original curve. They all have a similar concept:

> Offset some features of the original curve.
>
> Put a new curve through them.
>
> Attempt to measure how accurate an offset that is.
>
> If it is not adequate, subdivide the original curve and try again.

One quite well-known technique (see Klass' 1983 paper) for constructing approximate offsets to cubics is to interpolate a new cubic which matches the correct tangent directions and curvatures at the end points; but there's some numerical work needed to get the curvatures. An alternative is to offset each straight line segment in the control track of a Bézier curve, find their intersections and use that as the control track of a new curve.

Measuring and refining offsets

Measuring how good the approximation is can be done by a numerical integration process (see Farouki's 1986 paper on the subject) but in practice it is often adequate to try a few test points; even that isn't so easy. Working out where the normals from the original curve cut its offsets is not advised; you'll get equations with a L-O-T of roots. But we can't simply look at the distance between points at the same parametric position on original and offset curve; the result may appear accurate, but be rubbish because the parameterization is displaced from one curve to another, and the normal distance is actually too small. What you have to do is to compare the offset from the first curve with the corresponding parametric point on the offset curve. That may indicate an error when the offset is actually acceptable, again because of a change in parameterization, but at least it won't appear to be correct when it isn't.



11(i)—Approximate offsetting of a quadratic: three offsets are constructed; a new curve is interpolated through them; two further points on the new curve are compared with the corresponding offsets, to give a rough check on accuracy.

Just to see the sort of code that might be involved, let's make the Mickey Mouse version shown in Illustration 11(i). About the simplest thing you could do would be to take the end points and $n-1$ others on a curve of degree n, offset them all and put a new curve through them by Lagrange interpolation: too trivial to appear in 'the literature' but not too trivial to appear here.

First here's a macro to evaluate one of the parametric equations of a quadratic polynomial with coefficients a, b and c; the Horner form is employed as usual:

```
#define eval(a,b,c,t) ((a) + (t)*((b) + (t)*(c)))
```

Now we will put together another procedure to perform Lagrange interpolation of a quadratic through three points (x_1, y_1), (x_2, y_2) and (x_3, y_3), rashly assuming $t = \frac{1}{2}$ at the middle one. After solving the simple simultaneous equations, we can put together the code, again just for a single coordinate:

```
void interp(x1,x2,x3,a,b,c)
float x1,x2,x3;
float *a,*b,*c;
{
  *a = x1;
  *b = -3.0*x1 + 4.0*x3 + 4.0*x2;
  *c = 2.0*x1 - 4.0*x2 + 2.0*x3;
}
```

Next comes a procedure to work out an offset from a point:

```
#define ACCY (1.0e-6)  /* Or whatever you want
                          near-zero to be */

int offset(a1,b1,c1,a2,b2,c2,t,r,x_off,y_off)
float a1,b1,c1,a2,b2,c2,t,r;
float *x_off,*y_off;
{
  float t2,tc1,tc2,x_dash,y_dash,denom,den_inv,xn,yn;

  t2 = t + t;
```

```
     tc1 = t2*c1;
     tc2 = t2*c2;
/*
 * First, calculate the normal vector, of length r.
 */
     x_dash = b1 + tc1;
     y_dash = b2 + tc2;
     denom = x_dash*x_dash + y_dash*y_dash;

     if (denom < ACCY) return(1);

     den_inv = r/denom;
     xn = y_dash*den_inv;
     yn = -x_dash*den_inv;
/*
 * We assume a clockwise rotation between the
 * directions of tangent and normal; otherwise
 * the minus goes before y_dash.
 *
 * Now add the normal vector to the point on the curve.
 */

     *x_off = eval(a1,b1,c1,t) + xn;
     *y_off = eval(a2,b2,c2,t) + yn;
     return(0);
   }
```

Note that we're carting the coefficients of the quadratic about all over the place; rather inefficient, but keeps the code readable. So, we now have a piece of code that takes a quadratic curve, takes three normals from it, fits a new quadratic through them, and then compares offsets from the original curve at $t = \frac{1}{4}$ and $t = \frac{3}{4}$ to the parametrically corresponding points on the new curve.

```
#include <math.h>

/* Maximum value macro */

#define f_max(a,b) (((a) > (b)) ? (a) : (b))
```

```
   float a1,b1,c1,a2,b2,c2,r;
   float worst_err;

   float a1_new,b1_new,c1_new,a2_new,b2_new,c2_new;
   float dx,dy;
   float x_off_1,y_off_1,x_off_2,y_off_2,x_off_3,y_off_3;
   float x_off,y_off;
   float err1_sq,err2_sq;

   int offset();
   void interp();

     .
     .
     .

/*
 * Offset at three points.
 */
   if (offset(a1,b1,c1,a2,b2,c2,0.0,r,&x_off_1,&y_off_1))
      return(1);
   if (offset(a1,b1,c1,a2,b2,c2,0.5,r,&x_off_2,&y_off_2))
      return(2);
   if (offset(a1,b1,c1,a2,b2,c2,1.0,r,&x_off_3,&y_off_3))
      return(3);
/*
 * Interpolate in x and y to make a new curve.
 */
   interp(x_off_1,x_off_2,x_off_3,
          &a1_new,&b1_new,&c1_new);
   interp(y_off_1,y_off_2,y_off_3,
          &a2_new,&b2_new,&c2_new);
/*
 * Check the new curve against an offset at t=0.25.
 */
   if (offset(a1,b1,c1,a2,b2,c2,0.25,r,&x_off,&y_off))
      return(4);
```

```
   dx = x_off - eval(a1_new,b1_new,c1_new,0.25);
   dy = y_off - eval(a2_new,b2_new,c2_new,0.25);
   err1_sq = dx*dx + dy*dy;
/*
 * Check the new curve against an offset at t=0.75.
 */
   if (offset(a1,b1,c1,a2,b2,c2,0.75,r,&x_off,&y_off))
      return(5);
   dx = x_off - eval(a1_new,b1_new,c1_new,0.75);
   dy = y_off - eval(a2_new,b2_new,c2_new,0.75);
   err2_sq = dx*dx + dy*dy;

/*
 * Take the worst error
 * (C maths library is in doubles - *sigh*).
 */
   worst_err
   = (float)sqrt((double)(f_max(err1_sq,err2_sq)));
```

Again, that is rather dire code; but it gives some idea what is required, which isn't very much. The non-zero returns assume that this is to be built into a procedure which will flag duff input points (i.e. coincident ones). If `worst_err` was not acceptable, we could divide the original curve into two parts at $t = \frac{1}{2}$ and try again.

Even if we have an acceptable approximation to the algebra, the offset is often not what we want; Illustration 11(ii) shows a curve, its algebraic offset, and what we really wanted. These problems can be sidestepped if we avoid

Sharp corners.

Curvatures less than the offset distance.

'Global' problems where the curve approaches itself.

11(ii)—Where radius of curvature is smaller than the offset distance, the offset curve will cross itself. In practice, we need to detect such events and not attempt to offset from the offending piece of curve.

Many real systems survive on interactive aids (e.g. colouring curves or surfaces by curvature).

As regards surfaces, the offset at a point can be obtained from the vector product of two different tangents in the surface; typically we use those in the t and u directions. Patched approximations to offset surfaces have been discussed (see Farouki 1986 again), but are not widely used. As you might appreciate, the potential topological problems of the offset surface are severe. Interactive solutions are preferred.

Algebraic offsetting

It is possible, using the technique of resultants which we met in Chapter 10, to work out the implicit polynomial which is a constant distance from another. In principle, the algebra is quite easy, and also easy to understand; but, in practice, it can often get very complicated and has a tendency to make algebra systems get lost in a fugue of their own thoughts.

Consider offsetting a surface $P(x_P, y_P, z_P) = 0$.

Let $R(x_R, y_R, z_R) = 0$ be the offset surface a distance d away

from P that we're trying to find. What do we know? Well, we have the following system of equations:

$$\begin{aligned} P(x_P, y_P, z_P) &= 0 \\ (x_P - x_R)^2 + (y_P - y_R)^2 + (z_P - z_R)^2 - d^2 &= 0 \\ x_P - x_R + k\frac{\partial P}{\partial x} &= 0 \\ y_P - y_R + k\frac{\partial P}{\partial y} &= 0 \\ z_P - z_R + k\frac{\partial P}{\partial z} &= 0. \end{aligned}$$

The second equation is just Pythagoras on the distance between the point on one surface and the point on the other. The last three equations are the vector equation $(x_R, y_R, z_R) = (x_P, y_P, z_P) + k\bigtriangledown P(x_P, y_P, z_P)$, which says that the point on R must be somewhere along the normal to P. We can find the k resultant of

$$\begin{aligned} x_P - x_R + k\frac{\partial P}{\partial x} &= 0 \\ y_P - y_R + k\frac{\partial P}{\partial y} &= 0 \end{aligned}$$

and of

$$\begin{aligned} y_P - y_R + k\frac{\partial P}{\partial y} &= 0 \\ z_P - z_R + k\frac{\partial P}{\partial z} &= 0, \end{aligned}$$

giving two equations without k. We can then carry on eliminating variables from pairs of equations, until we end up with an equation in x_R, y_R, z_R, and d, which will be the offset surface $R = 0$ that we were looking for.

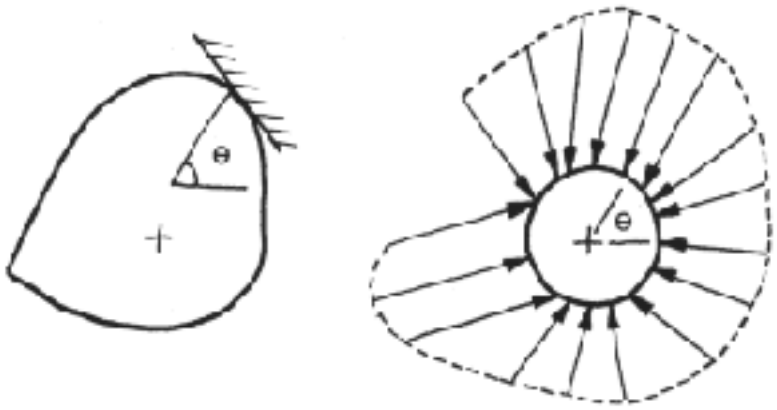
If you do all this and then plot out the results, the first thing you see is that you've got two offset surfaces, not one. They are, of course, the one on the inside and the one on the outside—the equation $(x_P - x_R)^2 + (y_P - y_R)^2 + (z_P - z_R)^2 - d^2 = 0$ didn't say anything about which direction the distance was measured in. The

other thing you'll find is that you've probably got a few other curves as well. These are additional factors that the resultant calculations incorporated in R. This confusion can be avoided by using a variable elimination technique such as Gröbner bases, but be warned—when you want to offset anything much more complicated than an ellipsoid, the algebra system will go off in a huff and never talk to you again.

There is a similar system of equations to the one above that allows you to find the surface that is equidistant between two others—the median surface. The derivation of this is left as an exercise for the more energetic reader.

Non-uniform offsets

We have assumed that the constant-distance offset curve or surface that we have been describing so far will be adequate. But there are many significant applications in which it is not: for example machining processes such as punching with non-circular punches, and milling with non-spherical cutters (there are good technological reasons for these requirements). Formulating continuous solutions to problems of this sort is not feasible; but they can be approached on a point-by-point basis, by taking the normal as before, and performing a transformation, based on its direction, which yields the nominal centre of the cutter that would be tangent to that point; see Illustration 11(iii). In effect, this is a lookup operation and, where the shape of the cutter is complicated, this can be a good way to program offsetting. The problems of avoiding gouging—eliminating parts of the offset surface that are not valid because of the proximity of another part of the surface—is more now complicated. We should check for intersections between any part of the cutter and the surface being machined. Even if we settle a simple test based on curvature being adequately large, a single value of curvature obviously cannot be used; it will be necessary to look up the curvature of the cutter at each contact point.



11(iii)—An irregular shape of cutter needs a different offset vector for each angle of contact. On the right of this figure is a lookup table from angle of contact to offset vector (both direction and length). Note that the corner on the cutter generates a sequence of angles over which the offset direction does not change. The flat portion, on the other hand, yields a particular angle at which the offset vector is not defined. These problems—and others, such as non-convex cutters—need solutions for technologies such as numerically controlled punching of sheet metal.

12

Geometric algorithms

In this chapter, we shall try to say something about geometric algorithms, as opposed to either algorithms or geometry in general. Algorithms are particularly important in geometry because:

> The impossibility of ordering data in more than one dimension leads us to use all sorts of more complicated structures, and it is not easy to ensure that these are used efficiently.
>
> Geometric applications such as graphics tend to involve data in much greater quantities than other forms of output (e.g. text); that unreasonable person the user may perceive a program to be 'slow' even though it produces a million pixels in the same time as an 'acceptable' program generates a ten-word sentence.

The study of geometric algorithms is often considered to be synonymous with *computational geometry*, which is centrally concerned with the theoretical analysis of the performance of algorithms. Algorithms can be judged:

> By their best-case, expected or worst-case run-time, and how these grows with the amount of data.
>
> By their best-case, expected or worst-case use of storage, and how this grows with the amount of data. This is less interesting, either because the extra (i.e. in addition to the required input and output data) storage required by the algorithm is trivial, or because more than enough storage is already available (i.e. is necessary anyhow for other algorithms in the same program or system).
>
> By their understandability, and hence ease of implementation, and maintainability.

> By their flexibility—extending algorithms over a range of related problems, or even re-using code for different types of data.

Let us guess that 90% of results in computational geometry are concerned with the theoretical *worst-case* time performance of algorithms against problem size. The discipline is correspondingly best-developed in the world of problems concerning sets of points, usually in two dimensions; Preparata and Shamos' book led the way here. It is now common to see concepts of order of algorithm being applied to more complicated geometric situations, but this can be less successful because:

> It is not always obvious what the size of the problem is; if we are working out the probability of intersections between, say, spheres, their radii (compared to their average separation) as well as the number of them is relevant.
>
> The worst-case complexity may be laughably far from the actual complexity. For instance, an algorithm that exploits parallel faces in models of engineering components might do quite well in practice. But, in the worst case, no planes are parallel; is this a good reason to judge the algorithm worthless?

A further caveat, which Preparata and Shamos are most careful to make, is that performance against problem size is only of interest in a world of perpetually growing problems. Although there are geometric applications where sets of data grow as fast as workstations can deal with them, in other areas that is not the case; for instance ship hulls do not become perpetually more complicated. Then other considerations, such as ease of implementation, may become dominant. Further, even if we are concerned with growing data sets, we can often use an algorithm with a poor performance on large data sets as a *part* of a program. For instance, we might want to generate the intersections between a large set of elements by generating the unbounded intersections, and then bounding them. This sort of approach is not efficient (it is $O(n^3)$) as a global strategy, but can be and is used within a limited region of space, where data set size is in effect constant. Poor-order algorithms may be much quicker than theoretically better ones on a limited amount of data. You can nest one algorithm inside another in this way, and tune

the resulting program by controlling the size of problem which one algorithm passes to the other. Don't leave yourself more than a few constants to tune, however; remember you are setting yourself a multi-dimensional optimization problem!

Point-set algorithms

Point sets give theorist of algorithms a field day [1]. Firstly, algorithmic properties can be studied without the overhead of too much algebraic geometry. Secondly, the results predicted for random point sets more nearly correspond with real problems; random curves and surface are scarcely credible testbeds for anything. There are two sorts of computational problems that involve point sets. The first is the computation of a *property* of the set, such as:

> The shortest distance between any two points.
>
> The longest ditto.
>
> The tightest polygon or polyhedron around the points: the convex hull.

The second sort of computation requires a *comparison* between the set and some other entity, for instance:

> The point nearest to a given point.
>
> All the points in a given polygon or polyhedron.

When only points are involved in the formulation of a problem (e.g. closest pair), then algorithms are typically extensible without difficulty from two to three dimensions. Where other structures, such as polygons and polyhedra, are involved, three-dimensional algorithms are typically a lot more complicated. As usual, these difficulties are actually caused by a change from one to two dimensions, but in structures embedded in the space (e.g. from an orderable polygon to a polyhedron which cannot be sorted).

However, perhaps the most important factor in point-set algorithms is that of data structure; whether we have to start with:

> Just a list of points.

[1] Or at least a job.

Points organized in a useful way but one not tailored for the problem in hand.

Points organized in a structure of our choosing.

Suppose, for example, that we want to find the point nearest to a given point, in two dimensions.

If we have no structure, we need to search through all the points in the data set; that will clearly be an $O(n)$ algorithm.

If we have a partially suitable structure, such as an ordering in one coordinate, then we can do a binary search in that coordinate, $O(\log n)$, but we'd expect to have to search through $\sqrt{n}$, because in effect we're looking at a slice through the space. This gets worse in three dimensions, where we'd expect to look through $n^{2/3}$ points.

If we were allowed to choose a structure, we could use a tessellation (see Chapter 6) grid. That allows us to access a cell in constant time. Or we might go for a quad-tree, and expect $O(\log n)$ performance, or for a Dirichlet tessellation, and expect $O\sqrt{n}$.

To decide what to do in a particular case we have to consider

How often we will be making a particular query.

Whether an existing structure is suitable—enough.

How quick another structure will be to build.

How quick it will be to update.

What it will cost in storage.

And how easy the whole system will be to hack up.

Given the difficulties of making these choices with point sets, no wonder the writing of algorithms for more complicated geometry is an art requiring cultivation.

13

Geometric programming

Once we have got some algebra and an algorithm, we can think about code. At this level, solving geometric problems is not that different from other programming tasks. This chapter concentrates specifically on two aspects:

> Accuracy—the lack of which can have a devastating effect on geometry.
>
> The use of storage—speeding up processing through code expansion and the use of lookup tables.

The tools of the trade—languages, packages and machines—remain much as in other areas, but a few remarks are in order.

The choice of a language is often Hobsonian; compatibility with existing code, availability of a compiler, and skills of programmers can all be overriding. Which language is fastest, especially if interaction is required? FORTRAN is the Ancient, and C the Modern answer to this question; C allows you to get closer to the machine, but RISC[1] architectures make this an increasingly questionable aim; there are some blindingly well-optimized FORTRAN compilers, and this language is still the standard in many varieties of scientific programming. FORTRAN retains its hallowed but curious syntax in places, and the low level of C can make it difficult to debug; and they boast inflexible and zero typing respectively. C++ *may* be the obvious solution to this problem. The ability to define data types and operations—corresponding to geometric entities like vectors, and operations like vector products—is an obvious advantage.

[1] Reduced Instruction-Set Computer

But it is not clear that code re-use, which is after all stated to be a major goal of object-orientation, is in general feasible with geometric data. Other readily extensible languages, such as PROLOG (which we use for some of the examples) are excellent for prototyping; PROLOG itself also shines at interfacing with databases (e.g. of molecular structures) and, to some extent, for handling algebraic expressions. But PROLOG, like the less structured AI language LISP, is quirky. Trying to achieve a goal in many different ways is fine, until an arithmetic error prompts your code to 'retry' the previous and perfectly adequate arithmetic statements rather than issue an (admittedly boring) error message!

The choice of a language may also be affected by the availability of packages which are callable from it. Packages are ways of avoiding writing some code, and we can try to avoid writing at a number of different levels, conveniently the four we identified in the Introduction: symbolic, analytic, numerical and approximate.

> At the *symbolic* level we will normally need to use an algebra system; this functionality is not commonly available as a package[2].
>
> At the *analytic* level, we can obtain packages which will deal with analytic geometry for us; there are, for instance, parametric geometry packages, which handle intersections and so on, available to computer-aided design system developers.
>
> The *numerical* level is rich with packages, but they are not necessarily readily applicable to geometric problems.
>
> *Approximations* are also catered for by numerical algorithm packages, but in a specifically geometric domain we might alternatively look for a package to deal, say, with quad- and oct-trees. A few of these do exist, but even if we have to write our own it can be shared between a group and provides a useful conceptual simplification.

However, the most common sort of package called from geometric code must be the *graphics package*; but arguably it's either nothing to do with geometry *per se* (yes, just a debugging tool) or it's another sort of approximation. The nature of a graphics package

[2] As mentioned in Chapter 1, NAG offer one.

can be ignored if it is just being used for output *and* it provides all the output needed. As soon we need to supplement the package (e.g. add hidden-surface capability to a wire-frame package) or we need to use it for input, then it often becomes a liability. The guts of a graphics package—the transforms, the data structures and so on—are often wholly or partially concealed (allegedly for the benefit of the more nervous sort of user, but in practice to avoid internal details becoming unalterable parts of the interface. But it is exactly this sort of partially transformed geometry, and screen-space coordinates etc., that the package developer doesn't want you to get at—but you have to, if you want to supplement the package's function. Conceptually at least, it may be easier to abandon the package and resort to steam methods of driving a device, but then compatibility and portability become casualties.

Similar constraints to those of a graphics package can also be applied by the requirement to conform with standards for the exchange of geometrical data. At the moment, these are only prevalent in some areas of computer-aided design and manufacture areas, but we may look to see them spreading into topics such as scientific data visualization. While they are an essential part of a maturing discipline, like a standard voltage on the mains, they take choices—and thus also opportunities for innovation and improvement—away from the software developer, and can also involve a considerable amount of complexity (cf. X-windows).

As far as machines themselves are concerned, there would be a lot to say if we were to consider all the special geometric hardware that has been produced—mostly in prototype form—over the last ten years or so; fascinating though these developments are, these chips are only in wide use within graphics displays, and if you are programming them you probably know how to do it already. They often involve integer versions of algorithms, exploiting the discrete nature of a raster screen (e.g. Bresenham's algorithm). Integer algorithms appear often enough in the literature, but are increasingly of specialist use, because floating-point operations are now as fast as integers on many processors (and both are now dominated by storage access times). Further, the rise of the RISC machine is a move away from attempting to exploit the peculiarities of the hardware at all. The speed of these machines can only practically be accessed

through a compiler; there is little you can do to exploit the machine architecture yourself. There are small exceptions, for instance recent machines of *superscalar* design do allow a small number of instructions to be executed in parallel; commonly an integer with a floating-point, or (as in the IBM RISC workstations, a floating-point multiplication with an addition). If you place two together in your C code, you can probably rely on the compiler to make sure they are performed together. For instance, a scalar product $x_a x_b + y_a y_b + z_a z_b$ will go nicely as follows:

```
sum = xa*xb + ya*yb + za*zb;
```

We could expect it to be done in the same time as three multiplications alone; and anyway, why should anyone code it as follows?

```
sum_1 = xa*xb;
sum_2 = ya*yb;
sum_3 = za*zb;
sum = sum_1 + sum_2 + sum_3;
```

But things like Horner forms $a_0 + t(a_1 + t(a_2 + \ldots$ are tiresome, because the multiplications and additions must be done sequentially. Of course, all this fast hardware is increasingly dependent on having data close to hand: in registers, or a cache at the least. If we have to fetch data from main memory (let alone a *disk* then the advantage of such architectures quite disappears.

Today's workstations with limited parallelism seem forever to keep ahead of large-scale parallel computers, such as the Inmos Transputer arrays, as practical general-purpose machines. Where problems with intrinsic parallelism, such as some signal-processing applications, can be found, a very fast special-purpose parallel code can certainly be written. There have, of course, been many efforts to write parallel geometric algorithms, but it is not today a mainstream subject. It is interesting to speculate whether, in the future, parallel machines will:

Remain a special-purpose tool.

Become a general-purpose tool, but require the use of languages which support parallel constructs explicitly.

Become a general-purpose tool, supported by systems to extract parallelism from algorithms.

The last prospect is distant.

Accuracy

The accuracy of geometric computations determines not merely the accuracy of the answer, in a numerical sense, but often its whole quality. In a graphics application, for instance, accuracy problems could make a thin object vanish: and ruin, or at least drastically change, the whole of a picture. You are probably familiar with this sort of problem. Remarkably few graphics programs are free of 'buggy pixels' even with run-of-the-mill input, let alone data deliberately designed to break the program.

Here are five different approaches to the accuracy question, with their advantages and disadvantages.

Do what you can algebraically

There's no point in fiddling with the code if the problem is not as well-conditioned as possible. We have already covered a lot of this ground. For instance:

Are equations normalized?

Are you using Horner or Bernstein bases where appropriate?

Are you doing the computations as near the origin as possible?

Maximize the raw accuracy available

The use of double precision is the obvious example. Quadruple precision has been tried in some sculptured-surface problems (e.g. intersections); the perpetrators of these outrages are in hiding. The advantage of increased precision is:

It can often be done at the flick of a compiler option.

The disadvantages are:

It makes everything slower.

> It is usually a counsel of desperation; double precision solves very few problems completely, but may reduce the incidence of trouble.

Do the computations exactly

Some graphics algorithms that work in screen space show how this can be done; a less restricted technique is use of arbitrary-precision rational arithmetic, common in algebra systems but not often seen in practice in geometric programs. The advantages of this approach are:

> Results are as exact as the input.
>
> In the correct context, integer arithmetic is fast.

Its disadvantages are:

> The range of computations that can be handled is extremely limited. Even square roots, and thus distance calculations, cannot be supported by integer or rational arithmetic (some clever people are starting to do exact arithmetic with roots; but then what about trig functions?).
>
> Arbitrary-precision rational arithmetic is slow.

Do the computations repeatably

Accuracy problems can sometimes be avoided if great care is taken always to do the same things to the same data and in the same order. Thus, two points that are transformed using an identical sequence of floating-point operations will end up in the same place. The points' positions may be 'wrong', but at least there isn't, for instance, a gap between the two polygons of which each point is a vertex. Advantage:

> Small run-time overhead.

Disadvantage:

> Can't be used in too many situations.
>
> Tricky to program (and to port—watch for compiler optimizations).

Allow for the inaccuracy in the computations

This is where we all get to in the end. Sophisticated schemes try to track inaccuracies using interval arithmetic and so on. Much more common is the use of 'fudge factors' i.e. accuracy constants that are compared against results. Advantages:

> Widely usable (and widely used).
>
> Programs can be improved by tuning fudge factors after implementation.

Disadvantages:

> The values of factors are not determined solely by allowing a generous bound on the floating-point accuracy. They must be chosen to match the well-conditioned, or ill-conditioned, nature of the calculations. Worse, factors must often have dimensionality, and are susceptible to scale effect.
>
> Difficult to get working.
>
> Rarely fully effective in practice: impossible to guarantee.

Fudge factors: a cautionary example

It is not difficult to see that we may need more than one fudge factor in a program. For instance, we might want to know when points are *on* planes and when vectors are *perpendicular* to them. In the first case, the linear scale of the data with which we are working is clearly important: rescale all our dimensions from metres to millimetres and *some* programs stop working. In the second case, we can fix a accuracy constant for scalar products that defines a small angle, and will not need to be changed for bigger or smaller objects (but maybe will if the application changes).

Another way in which fudge factors may proliferate is that we sometimes need coarser factors to determine what elements take part in a subsequent calculation, to which a finer factor applies. For instance, supposing we are generating the vertices of some polyhedra from the plane equations of their faces, and these polyhedra are known to share faces. If more than three planes meet at a vertex, their mutual intersection points are most unlikely to coincide exactly. However, only a nitwit wants to complicate the edges of the

polyhedra with a lot of tiny bogus edges around the intended vertices. Therefore the vertex 'estimates' resulting from the three-plane intersection calculations must be coalesced. If the plane equations have floating-point coefficients (i.e. nothing exotic) the only way to control that coalescing is by a fudge factor, specifying a distance. If a group of points is closer together than that distance, it is replaced by the centroid of the group.

To deal with a large number of points efficiently, we will need a spatial structure. Suppose we go for an oct-tree; what happens then? Suppose that the plan is to create the oct-tree as the points are generated, and then to coalesce the groups found in each 'oct'. But what happens if a cloud of points spans the wall of an oct? We will get two very close vertices and a bijou edge; just what we didn't want. Now, when we visit each oct, we *could* also look at the points in adjoining octs. But neighbour-finding in an oct-tree is rather complicated, especially if it is stored in a space-efficient manner. Further, we will be examining a great deal of the data nine times over.

So, what about making the octs just a little bit bigger when we sort the points between them? That will make sure that we've got all the points we need in each oct, which can then be considered only once. Now, we might imagine that we could expand each oct by the same factor that we're going to use to coalesce the points. But it's too risky; we might just fail to include a point in a box that would still be acceptable for the coalescing process: much more sensible to define a much wider fudge factor to add to the box, and be sure we catch everything of interest.

Of course, you'll be saying that this method won't work anyway, because coalesced points near the boundary of an oct will appear in both octs; in fact, near a corner, coalesced points could appear eight times. And to hunt them down, you say we'd need to find the neighbours, which we swore we wouldn't do.

What about this solution? Assume each point is numbered (this may just be an array index). Using this numbering, ensure that the calculation of the centre of gravity of a group of points is done in a uniform fashion. The comparison of that centroid with each (exact, no fudge factor) wall of an oct is easy to do consistently, because the octs are axially aligned, and only a subtraction is required. Using

this method, the coalesced vertices can be uniquely assigned to an oct.

Problems with this method only occur when two real vertices are within fudge-factor range of each other. We cannot guarantee success here. However, the wide second fudge factor around the octs will help to ensure that near vertices are treated consistently in adjacent boxes. If the bound around the octs is ten times the coalescing factor, it will need a chain of ten pessimally close vertices to upset the process, provided that vertex coalescing is consistent. Coalescing using a *greedy* algorithm, starting from the closest points, will improve consistency of the coalescing between octs.

Using storage effectively

While operating systems miraculously expand to fill any and all disks available, scientific algorithms seem to remain just the same size. That is to say, books about programming and algorithms aren't ten times thicker than they were in 1980, and certainly not 100 times thicker than they were in 1970, although there may be more books out there.

Put it another way. Performance and memory capacity of 'average' computers has increased in remarkable synchronization over the last twenty years. If you use the same code as you did twenty years ago it will run faster, and you can have more data, but your algorithms may not be as efficient as they could be were they more compact. This is not a plea for sloppy code (OS writers please note), but it is a suggestion that more efficient and even simpler code can result from an imaginative use of space. We suggest two areas which strongly relate to geometric code.

Expanding loops and recursive calls

Geometric code is notorious for having tight loops in which the same code is executed many times. Just putting more code into loops is not of course a good idea, but there are cases where loops can be expanded to good effect. A simple example is some code to evaluate a parametric polynomial in the Horner scheme. We *could* have two loops here: one for the coefficients (`a`) and one for the coordinates

of the result (`q`). Here is an execrable example:

```
for(dim = 0; dim < n_dim; dim++)
{
     q[dim] = a[n_coeff,dim];
     for (coeff = n_coeff-2; coeff >= 0; coeff--)
          q[dim] = t*q[dim] + a[coeff,dim];
}
```

Well that's quite nasty. But note:

> The algorithm is basically an efficient one.
>
> The code is typographically compact.
>
> It is academically impressive, to the extent that it permits a general degree of polynomial and a general number of dimensions, useless as both of these will often be.

Similar things happen with recursion. Again, this makes elegant code, but there may be an overhead imposed by the recursion itself, and the recursion may fail to permit us to take obvious short cuts. As an example, we offer three algorithms to calculate B-spline curves:

> The recursive de Casteljau construction: an $O(n^2)$ algorithm, but wonderfully elegant.
>
> The Horner scheme, roughly as given by Farin in his book.
>
> Our version of Horner with the loop expanded (for cubics in this case).

Here's the procedure corresponding to the de Casteljau construction in PROLOG. This version works for a single coordinate only, and the coordinate values start off in *data* predicates. The notation follows the algebra in Chapter 4.

```
decast(1,0,*,X1).
decast(1,1,*,X2).
```

... And so on. Now some action.

```
decast(R,I,T,B)
<- RN := R - 1
 & IP1 := I + 1
 & decast(RN,I,T,B1)
 & decast(RN,IP1,T,B2)
 & B := B1 * (1.0 - T) + B2 * T.
```

Yessir/ma'am, that's it; pretty it is, efficient it is not (in PROLOG or otherwise).

Secondly, in C this time, here is a nearly straightforward Horner scheme, except that the Bernstein coefficients are obtained from a lookup table rather than calculating them on the fly. We use an array for this, because the table is actually quite short (only 36 elements, if we go up to degree 10), and we don't want to incur the overhead of a function call every iteration. Note that:

> The array `p` contains one of the sets of coordinates; we would need to call the routine twice in two dimensions, three times in three.
>
> The routine does work—rather inefficiently—in the linear case (`n` = 1) as we can rely on the compiler to ignore the loop. In that case the value `l_off[0]` is ignored, as `look_up` is never used.

```
/*
 * Bernstein coefficients.
 */
static float look_up[] = {
             2,
             3, 3,
             4, 6, 4,
             5, 10, 10, 5,
             6, 15, 20, 15, 6,
             7, 21, 35, 35, 21, 7,
             8, 28, 56, 70, 56, 28, 8,
             9, 36, 84, 126, 126, 84, 36, 9,
             10, 45, 120, 210, 252, 210, 120, 45, 10
             };
/*
```

```
 * Offset into look_up for each degree (goes up to
 * degree 10).   * The entry l_off[0] is a dummy (see
 * the text below).
 */
static int l_off[] = {0, 0, 1, 3, 6, 10, 15, 21, 28, 36};

/*
 * Function to return coordinate at t.
 */
float bern(n,t,p)

int n;     /* Number of control track points minus 1 */
float t;   /* Parameter value */
float p[]; /* Control-track coordinates
               (for 1 dimension) */
{
    int l;
    float s,b,t_power;

    l = l_off[n];
    s = 1.0 - t;
    b = p[0]*s;
    t_power = t;

    for (i = 1; i < n; i++)
    {
         b = s*(b + t_power*look_up[l++]*p[i]);
         t_power = t*t_power;
    }

    b = b + t_power*p[n];
    return(b);
}
```

The array `look_up` contains the Bernstein coefficients of the inner terms (only), while `l_off` provides the offset for the particular degree of equation we're evaluating. The first line (`2`) is for quadratics, the second line (`3, 3`) is for cubics, and so on.

The first value in `l_off` (`0`) is a dummy, corresponding to the

case `n` = 1. It is there to avoid having to subtract 1 from `n` when `l_off` is accessed. Each subsequent entry is a value of $n(n-1)/2$; they are used to index the lines of data in `look_up`.

Finally, here is a 'written-out' routine for the Horner form. This applies to cubics only, but on the other hand it *does* deal with all the coordinates at once to save re-calculating powers of t *and* it incorporates the Bernstein coefficients in-line:

```
s = 1.0 - t;
t_x_3 = t*3.0;
t_sq_x_3 = t_times_3*t;
t_cu = t*t*t;

bx = ((s*x0 + t_x_3*x1)*s + t_sq_x_3*x2)*s + t_cu*x3
by = ((s*y0 + t_x_3*y1)*s + t_sq_x_3*y2)*s + t_cu*y3
bz = ((s*z0 + t_x_3*z1)*s + t_sq_x_3*z2)*s + t_cu*z3
```

This has not been made a function, because in practice it would probably make sense to couple it closely with plotting routines etc. It would be no problem to box the code up to suit. Generating a few more of these to handle other orders of curve is not a big problem; we could even generate them automatically. The programming time required is probably about the same either way, but automatically generated code may contain fewer errors (i.e. typos).

More lookup tables

Having seen lookup tables used to store Bernstein coefficients (a relaxingly trivial example), we will go on to see some more geometric uses of that splendid programming device. They can be divided into two types. In the first type, we approximate a continuum with a set of discrete values, and interpolate between them. In the second type, a situation has a genuine integer number of outcomes.

When lookup tables are used to approximate a continuous function, getting adequate resolution from a reasonable size of table is often a problem. Beware of situations where multi-dimensional lookup tables are required: for instance, looking up the distance between two points in space is not normally considered feasible; this is a six-dimensional problem, and so storage requirement varies as

the sixth power of resolution! That distance problem has another aspect; even if we could afford the storage, we would need to know the order of magnitude of the distances involved in order to build the table with appropriate values. But a real system might be dealing with a spectrum of distance of $1 : 10^6$ and more. We can only get around this problem by some normalization step, which neatly uses up the time the table is meant to be saving.

However, in graphics, limited screen resolution ameliorates this problem. One interesting use of *continuous* lookup tables is to provide a repeatable set of related pseudo-random numbers for solid texture. This is much faster than recursive-division type pseudo-fractal algorithms. In general, however, we will do better to stick to using lookup tables for dimensionless quantities, like angle; a complicated trig function that was in frequent use would be a candidate, for example. Even so, it often impossible to make a continuous lookup table—that you think is ingenious—pay in practice.

The case for lookup tables where there are distinct outcomes is much stronger. This opportunity may arise from geometric considerations. The classic examples are clipping of the Cohen-Sutherland kind (see Foley and van Dam's book), in which straight-line segments are classified against the area of a screen by a lookup table based on the regions in which the lines' end-points lie. This can be extended to mutual intersections of straight-line and arc segments (see *A Programmer's Geometry*). The distance between two axially-aligned boxes is a further extension, and this is obviously useful in pre-processing for more detailed distance calculations.

Rather than look for cases based on the position of two elements, we can look up how points or elements are classified. For instance, we can classify coloured quads in a quad-tree in this way. If two or three adjacent quads share the same colour—or other data value—we need only write it out once (for an extreme example, see Woodwark's paper "Compressed quad trees"). A more recent example along the same lines is the classification of cubic regions based on whether their corners are inside or outside a surface of interest. This is a very quick way to decide how to approximate the surface within that region, and is used in volume visualization (Lorensen and Cline's marching cubes paper of 1987 is widely cited).

References and Bibliography

The references from the text are here intermingled with a bibliography, and the whole informally annotated. It's not a very standard list, but we hope it provides some insights, if only into the diseased brains of the compilers. We make no apology for referring to a good selection of our own efforts; if we don't advertise them, who will?

Useful proceedings

Proceedings of the ACM SIGGRAPH Conferences , ACM, annually.

The first choice of publication medium for many authors of papers on graphics. It has recently become divided into shorter and more diverse sections; see the comments in the Introduction.

Proceedings of the ACM Symposia on Computational Geometry , ACM, 1985 and annually thereafter.

An excellent source of snapshots of a range of topics in computing with geometry, although it inclines to the 'orders of algorithms' end of the market.

Proceedings of the International Symposia on Spatial Data Handling , Geographical Union, 1984 and every two years since.

Representing geographical geometry using computers is always a hard problem. This series provides a good summary of the progress that cartographers and geographers have made in recent years. The papers tend not to be very mathematical, and sometimes lack algorithmic and computational insight, but give a view of the field as seen by the practitioners.

Books and papers

J.H. Ahlberg, E.N. Nilson and A.J. Stein, *The Theory of Splines and their Applications,* Academic Press, 1967.

A difficult book, but worth a glance into, to see where splines came from.

F. Ayres, *Projective Geometry,* McGraw-Hill, 1967.

One of McGraw-Hill's Schaum Outline texts, intended for the student pocket. Good value, and covers the basics.

T.F. Banchoff, *Beyond the Third Dimension*, Freeman, 1990.

A valiant attempt to show that you can visualize more than three spatial dimensions: we've read the book, and still say you can't.

R.E. Barnhill, G.E. Farin, M. Jordan and B.R. Piper, "Surface/surface intersection", *Computer Aided Geometric Design* **4**,1–2 (3–16), 1987.

About the classic patch-patch intersection problem.

B.A. Barsky and T.D. DeRose, "Geometric continuity of parametric curves: three equivalent characterizations", *IEEE Computer Graphics and Applications* **9**,6 (60–68), November 1989 and "Geometric continuity of parametric curves: construction of geometrically continuous splines", *IEEE Computer Graphics and Applications* **10**,1 (60–68), January 1990.

These two papers constitute a tutorial on geometric continuity.

C. Bell, B. Landi and M.A. Sabin, "The programming and use of numerical control to machine sculptured surfaces", Proceedings of the 14th International Machine Design and Research Conference, Manchester (233-238), 1973.

Not a recent paper, but one of the few about the important practical problem of non-spherical offsets.

C.B. Besant and C.W.K. Lui, *Computer-Aided Design and Manufacture*, Ellis Horwood, 1986.

There is rather too much about how computers and graphics devices work, including glossy photographs of kit, but the book is a reasonable introduction to computer-aided design and manufacture.

J.F. Blinn, "A generalization of algebraic surface drawing", *ACM Transactions on Graphics*, **1**,3 (235–256), July 1982.

Blinn's implicit blends for molecular graphics.

W. Böhm, G.E. Farin and J. Kahmann, "A survey of curve and surface methods in CAGD", *Computer Aided Geometric Design* **1**,1 (1–60), 1984.

The first article ever published in this journal; a bit out-of-date but succinct.

K.M. Bolton, "Biarc curves", *Computer-Aided Design* **7**,2 (89–92), April 1975.

Spline curves made out of biarc segments.

A. Bowyer, "Computing Dirichlet tessellations", *Computer Journal* **24**,2 (162–166), 1981.

A. Bowyer and J.R. Woodwark, *A Programmer's Geometry*, Butterworths, 1983, 2nd edition 1988.

A how-to-do-it book of code and explanation on some of the simpler geometrical constructions.

H. Chiyokura, *Solid Modeling with Designbase*, Addison-Wesley, 1988.

Describes in considerable detail how Chiyokura's Designbase boundary modeller works. It is sufficiently detailed that, after reading it, you could in principle write a similar modeller yourself.

W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981.

Maybe not the latest Prolog book, but a very well-known one; we've tried to be more-or-less consistent with its notation in the few Prolog examples we provide.

S.A. Coons and B. Herzog, "Surfaces for computer-aided aircraft design" Proceedings of the AIAA 4th Annual Meeting and Technical Display, Anaheim, AIAA Paper 67–895, 1967.

Not too easy to obtain, but included just to show how long Coons patches have been around. There's something about them in most books (e.g. Woodwark 1986).

J.H. Davenport, Y. Siret and E. Tournier, *Computer Algebra*, Academic Press, 1989.

This is the best book on computer algebra that we have seen. It starts very gradually, but gets tough later on, and the reader is sometimes foxed. Try to understand the method of resultants from this book, for instance; we couldn't.

A.A. Dubrulle, "A class of numerical methods for the computation of Pythagorean sums", *IBM Journal of Research and Development* **27**,6 (582–589), 1983.

R.A. Earnshaw (ed.), *Fundamental Algorithms for Computer Graphics*, Springer-Verlag, 1985.

The proceedings of a NATO Advanced Study Institute on graphics algorithms; the list of contributors is a Who's Who of the field, and most aspects of graphics are addressed, often in some depth.

H. Eves, *A Survey of Geometry*, Allyn and Bacon, 1972.

A very Euclidean perspective on the subject: not meaning that it deals with Euclidean space, but that it comprises a modern Euclid's elements. It starts with an interesting historical chapter at the beginning. Worth consulting for links into other parts of mathematics, how about (e.g.) the relationship between Lennes Polyhedra (polyhedra that can be split into tetrahedra using only the original polyhedron vertices) and Cauchy's theorem?

G.E. Farin, *Curves and Surfaces for Computer Aided Geometric Design* (2nd ed.), Academic Press, 1990.

A recent work which deals with parametric curves and surfaces from a strongly 'Bernsteinian' viewpoint. This book is subtitled A Practical Guide*, and this promise is fulfilled by the inclusion of a number of routines in the C language as codas at the end of each chapter. However, the overall approach is quite theoretical. The interpolated chapters on "Differential geometry" by Böhm seem heavy going to us. The organization of the final chapter "Evaluation of some methods" inspired our list of advantages and disadvantages of the Bernstein basis in Chapter 4.*

R.T. Farouki, "The approximation of non-degenerate offset surfaces", *Computer Aided Geometric Design*, **3**,1 (15–43), May 1986.

R.T. Farouki and V. Rajan, "On the numerical condition of polynomials in Bernstein form", *Computer Aided Geometric Design* **4**,3 (191–216), 1987.

R.T. Farouki and V. Rajan, "Algorithms for polynomials in Bernstein form", *Computer Aided Geometric Design* **5**,1 (1–26), 1988.

R.T. Farouki and T. Sakkalis, "Pythagorean hodographs", IBM Thomas J. Watson Research Center Report RC 15223, 1989.

I.D. Faux and M.J. Pratt *Computational Geometry for Design and Manufacture,* Ellis Horwood, 1979.

This book has worn well despite the lack of a second edition. Obviously many things have changed since 1979, but some readers may prefer this less Bernstein-oriented text. The level of coverage is a little uneven. For instance, there is an introduction to Cartesian geometry that includes stick figures at various angles. That's rather quaint even for 1979; nevertheless, it's probably more use to most people than the 'introductory' work which starts "Consider an arbitrary mapping $\mathbf{R}^n \Rightarrow \mathbf{R}^m \ldots$*".*

J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes, *Computer Graphics: Principles and Practice* (2nd ed.), Addison-Wesley, 1990.

New edition of 'Foley and van Dam' which ousted 'Newman and Sproull' as the best-known graphics book.

P.C. Gasson, *Geometry of Spatial Forms*, Ellis Horwood, 1983.

Rather like an incomplete butterfly collection: there are many beautiful and interesting things in it, but also some things that are missing completely. For instance, there is a section on crystal geometry, but nothing on parametric surfaces: worth looking at, but not a 'standard' text.

A.S. Glassner, "Spacetime ray tracing for animation", *IEEE Computer Graphics and Applications* **8**,3 (60–70), March 1988.

Not the first attempt to treat animation as a 'four-dimensional model' with time as one of the dimensions. Intuitively attractive, maybe: but read the article and notice how little advantage (from 'frame-to-frame coherence') was actually obtained (see Woodwark 1988).

A.S. Glassner, ed., *Graphics Gems*, J. Arvo *Graphics Gems II*, D. Kirk *Graphics Gems III*, Academic Press, 1990, 1992 and 1993.

There a lot of geometry in these books, although they are primarily about graphics. The sections were submitted by various authors, and the level and coverage is correspondingly diverse: perhaps increasingly so as the series progresses. There are *some gems, but paste too. It was a good idea to define a uniform pseudocode to link the contributions, but it seems an unusually opaque one.*

P.J. Green and R. Sibson, "Computing Dirichlet tessellations in the plane", *The Computer Journal* **21**,2 (168–173), 1978.

B. Guenter and R. Parent "Computing the arc length of parametric curves", *IEEE Computer Graphics and Applications* **10**,3 (72–78), May 1990.

M.J. Haigh, *An Introduction to Computer-Aided Design and Manufacture*, Blackwell, 1985.

Rather similar to Besant and Lui's book.

D. Harper, C. Wooff and D. Hodgkinson *A Guide to Computer Algebra Systems*, Wiley, 1991.

This is like a Which? guide to algebra systems—it says what all the commonly available ones can and can't do using helpful tables, and also has a section of example problem solutions at the back, computed using various of the systems reviewed. If you're going to start using algebra systems for the first time, read this book before doing so.

F.R.A. Hopgood, D.A. Duce, J.R. Gallop and D.C. Sutcliffe, *Introduction to the Graphics Kernel System* , Academic Press, 1983.

Graphics standards are dull, and GKS is no exception. However, it is sometimes necessary to refer to them, and this book makes the exercise comparatively painless.

R. Klass, "An offset spline approximation for plane cubics", *Computer-Aided Design* **15**,5 (296–299), 1983.

D.E. Knuth, *The Art of Computer Programming* (2nd ed.), Addison-Wesley 1973.

All computer life is here. This is the best-known work—and rightly so—on everything about computer programs.

J.J. Koenderink, *Solid Shape*, MIT Press, 1990.

Like Gasson's book, Solid Shape *contains many interesting observations, in an individualistic and 'descriptive' treatment. The focus is on differential geometry, and the slant is towards vision, rather than graphics. For once, this book can be recommended as a 'read'.*

A.E. Lord and C.B. Wilson, *The Mathematical Description of Shape and Form*, Ellis Horwood, 1984.

An interesting mixture of geometric and topological approaches to the description of form. It covers both 'conventional' topics, such as transforms, projection, curve and surface fitting, and also deals with self-similarity, fractals, Penrose tessellations and other such topics.

W.E. Lorensen and H.E. Cline, "Marching cubes: a high resolution 3D construction algorithm", *Computer Graphics* **21**,4 (Proceedings of SIGGRAPH 87) (163–169), July 1987.

A.M. MacBeath, *Elementary Vector Algebra*, Oxford University Press, 1964.

What we said in A Programmer's Geometry *still holds: a good introduction to vector algebra, with particular emphasis on applications in three-dimensional geometry.*

M. Mäntylä, *Solid Modeling*, Computer Science Press, 1988.

Not dissimilar to Chiyokura's book: very strongly orientated towards boundary models and specifically reports the details of the GWB (Geometric WorkBench) modeller.

G. Markowsky and M.A. Wesley, "Fleshing out wire frames", IBM Thomas J. Watson Research Center Report RC 8124, 1980.

P.S. Milne, "On the algorithms and implementation of a geometric algebra system", University of Bath Computer Science Technical Report 90–40, 1990 (*e-mail:* tech-report uk.ac.bath.maths)

One of our research students' PhD theses. It describes how to design a symbolic system to do geometry; it also has a good section on interval arithmetic, and a lot of detail on how to find the zeros of polynomial expressions describing curves and surfaces.

C. Moler and D. Morrison, "Replacing square roots by Pythagorean sums", *IBM Journal of Research and Development* **27**,6 (577–581), 1983.

R.E. Moore, *Methods and Applications of Interval Analysis* , SIAM, 1979.

The prime use of intervals is in bounding approximate (e.g. floating-point) arithmetic calculations. Not much of this book is very relevant to geometry, but there doesn't seem to be a better text about.

M.E. Mortenson *Geometric Modeling*, Wiley, 1985.

This book is remarkable for its lack of special pleading; it is probably the only one in this field in which the author does not cite a single paper of his own. Unfortunately, we have not often found it insightful.

N.M. Patrikalakis and G.A. Kriezis, "Representation of piecewise continuous algebraic surfaces in terms of B-splines", *The Visual Computer* **5**,6 (360–374), 1989.

N.M. Patrikalakis and P.V. Prakash, "Surface intersections for geometric modeling", *Transactions of the ASME: Journal of Mechanical Design*, **112** (100–107), March 1990.

An accessible reference on this subject; many other results are in PhD theses etc.

T. Pavlidis, *Algorithms for Graphics and Image Processing,* Springer-Verlag, 1982.

Good coverage of the basic common ground between graphics and vision, twin subjects which often seem to have that well-known property of parallel lines—that they never meet.

K. Perlin, "An image synthesizer", *Computer Graphics,* **19**,3 (Proceedings of SIGGRAPH 85) (287–296), 1985.

The first publication about 'solid texture', but not the paper to read if you actually want to implement it.

L. Piegl, "Key developments in computer-aided geometric design", *Computer-Aided Design* **21**,5 (262–274), June 1989.

A short history of parametric methods.

L. Piegl "On NURBS: a survey", *IEEE Computer Graphics and Applications* **11**,1 (55–71), January 1991.

Just what it says.

R.A. Plastock and G. Kalley, *Computer Graphics,* McGraw-Hill, 1986.

Like 'Ayres', a Schaum Outline: good value, but graphics dates much more quickly that 'pure' geometry.

D. Pletinckx, "Quaternion calculus as a basic tool in computer graphics", *The Visual Computer* **5** (2–13), March 1989.

Quaternions provide a more compact alternative to matrices for rotation transforms; worth looking at to put matrices in perspective (last bad pun).

F.P. Preparata and M.I. Shamos, *Computational Geometry: an Introduction*, Springer-Verlag, 1985.

At its publication date, a snapshot of the subject in considerable detail. Like the ACM Symposia Proceedings, it tends to be over-concerned with worst-case performance, as against more practical considerations.

W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling *Numerical Recipes* and *Numerical Recipes in C*, Cambridge University Press, 1986 and 1988.

Subtitled The Art of Scientific Computing*, these are highly successful compilations of snippets of numerical code, with enough explanation so that (you think) you know what you're doing when you use them. The first book has FORTRAN and (in an Appendix) Pascal code; the second book has the code in—of course—C. There are no explicit geometric sections, but many sections are useful in geometric applications all the same.*

A. Ricci, "A constructive geometry for computer graphics", *Computer Journal* **16**,2 (157–160), 1973.

The famous Ricci blend; note the date.

A.P. Rockwood and J.C. Owen, "Blending surfaces in solid modelling", Proceedings of a SIAM Conference on Geometric Modeling and Robotics, (G. Farin, ed.), Albany N.Y., 1985.

D.F. Rogers and J.A. Adams, *Mathematical Elements for Computer Graphics*, McGraw-Hill, 1976.

Not a new book, but one of the few to provide lots of detailed worked examples, for practical people such as ourselves who like that sort of thing. We have heard reports of some typographical errors amongst the wealth of mathematics presented, so do convince yourself of the correctness of anything you use. (That of course applies to any book on geometric computing, and especially the present volume.)

H. Samet, *The Design and Analysis of Spatial Data Structures* and *Applications of Spatial Data Structures* , Addison-Wesley, 1989.

These two books undoubtedly comprise the most complete work on quad- and oct-trees etc. available. They are good for reference: being, as one reviewer said, extended survey papers. However, in our opinion they follow the minutiae of the original authors' approaches too closely to be a completely successful exposition of the subject.

T.W. Sederberg and R.N. Goldman, "Algebraic geometry for computer-aided geometric design", *IEEE Computer Graphics and Applications* **6**,6 (52–59), June 1986.

Some comprehensible words on difficult topics such as intersections and implicitization.

T.W. Sederberg, S.C. White and A.K. Zundel, "Fat arcs: a bounding region with cubic convergence", Brigham Young University, Engineering Computer Graphics Laboratory Report ECGL-88-1, 1988.

M. Tamminen, "The extendible cell method for fast geometric access", Helsinki University of Technology Report HTKK-TKO-A20, 1980.

W.C. Thibault and B.F. Naylor, "Set operations on polyhedra using binary space partitioning trees", *Computer Graphics* **21**,2 (Proceedings of SIGGRAPH 87) (153–162), July 1987.

M. Watkins and A. Worsey "Degree reduction for Bézier curves", *Computer-Aided Design* **13**,4 (398–405), 1988.

J.R. Woodwark "Compressed quad trees", *Computer Journal* **27**,3 (225–229), August 1984.

J.R. Woodwark, *Computing Shape*, Butterworths, 1986.

This short book provides an introduction to the shape models used in computer-aided design in a single volume. Not superseded, but in need of revision, especially the list of references.

J.R. Woodwark, "Blends in geometric modelling", in *The Mathematics of Surfaces* (R.R. Martin, ed.) (Proceedings of the 2nd IMA Conference on the Mathematics of Surfaces, Cardiff, 1986) (255–297), Oxford University Press, 1987.

This was quite comprehensive in its day; blends need a new review paper.

J.R. Woodwark "Spacetime ray tracing", Letter to the editor, *IEEE Computer Graphics and Applications* **8,**5 (8), September 1988.

Comment on Glassner's article "Spacetime ray tracing for animation".

G. Wyvill, C. McPheeters and B.L.M. Wyvill "Soft objects", *Advanced Computer Graphics* (Proceedings of Computer Graphics 86, Tokyo) (113-128), 1986.

Around the first of a number of publications about the 'soft object' approach.